International Journal of Reconfigurable Computing

Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)

Guest Editors: Lionel Torres and Viktor K. Prasanna

Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)

International Journal of Reconfigurable Computing

Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)

Guest Editors: Lionel Torres and Viktor K. Prasanna

Editorial Board

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 679484, 2 pages
doi:10.1155/2010/679484

Editorial

Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)

Lionel Torres[1] and Viktor K. Prasanna[2]

[1] *Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), UMR CNRS 5506, Université de Montpellier 2, 34090 Montpellier, France*
[2] *University of Southern California, Los Angeles, CA 90033, USA*

Correspondence should be addressed to Lionel Torres, lionel.torres@lirmm.fr

Received 31 December 2010; Accepted 31 December 2010

The fifth edition of the International Conference on Reconfigurable Computing and FPGAs (ReConFig 2009) was held in Cancun, Mexico, from December 9 to 11, 2009. ReConFig is a leading edge forum for researchers and engineers across the world to present their latest research and to discuss future research and applications. The conference seeks to promote the use of reconfigurable computing and FPGA technology for research, industry, education, and applications.

This special issue covers actual and future trends on reconfigurable computing given by academic and industrial specialists from all over the world. Papers presented in this special issue were selected from all ReConFig 2009 submissions and were peer reviewed for the final publication in this journal with the breadth and depth needed for readers involved in the reconfigurable computing field.

There are a total of 15 articles in this issue. We begin the special issue with two papers that extend across the digital signal processing domain, and especially Software Define Radio. The paper by J. A. Surís et al. "*RapidRadio: A Domain-Specific Productivity Enhancing Framework*" addresses a framework enhancing tool reducing the required knowledge base for implementing a receiver on an FPGA-based SDR platform. In "*Space-Based FPGA Radio Receiver Design, Debug, and Development of a Radiation-Tolerant Computing System,*" Z. K. Baker et al. proposed a processing capability which enables very advanced algorithms such as wideband RF compression scheme to operate at the source, allowing bandwidth-constrained applications to deliver previously unattainable performance.

Three papers are within the area of arithmetic applied to digital signal processing. The paper by M. Baesler et al. presents a new parallel decimal fixed-point multiplier designed to exploit the features of Virtex-5 FPGA in order to provide important operation with least possible rounding error.

In the next paper, "*Partial Reconfigurable FIR Filtering System Using Distributed Arithmetic,*" D. Llamocca et al. propose two efficient dynamic partial reconfiguration systems allowing to implement a wide range of 1D FIR filters. For both systems, the required partial reconfiguration region is kept small by using distributed arithmetic implementations. R. A. Arce-Nazario et al., in "*Reconfigurable Hardware Implementation of a Multivariate Polynomial Interpolation Algorithm,*" address a new methodology based on Lagrange interpolation. The generalized algorithm can be efficiently mapped to a systolic array in which each processing cell implements a pair of binary operations between an incoming and a stored value. The FPGA implementation of the reduction operations and the complete application achieved speedups of up to 172× and 67×, respectively, as compared to software implementations run on a contemporary CPU, with moderate resource utilization.

Five papers are within the broad area of automated design and multiprocessors systems on chips. As discussed in "*Reconfigurable Multiprocessor Systems: a Review*" by T. Dorta et al., run-time reconfigurability uses the dynamic reconfiguration feature of FPGAs to obtain a new degree of freedom in the design of multiprocessor systems, making these systems more flexible to target different applications using the same hardware. L. Kirischian et al. propose in "*Mechanism of Resource Virtualization in RCS for Multitask Stream Applications*" a mechanism for the virtualization

of computing resources for multitask and multi-mode stream applications. The comparative analysis made on the Multitask Adaptive Reconfigurable System demonstrated negligible hardware overhead offset by sufficient gains in the main performance parameters. Memory access is a key point for modern design. In "*Traversal Caches: A Framework for FPGA Acceleration of Pointer Data Structures*," J. Coole and G. Stitt introduce an original traversal cache framework that enables efficient FPGA execution of applications with irregular memory access patterns. Such approach greatly improves effective memory bandwidth for repeated traversals. The next paper describes the way to make the reconfigurable resources available for all applications in the embedded systems domain, particularly mobile devices or networked devices where the requirements change. S. Döbrich and C. Hochberger show that the AMIDAR concept allows providing a general-purpose processor that can automatically take advantage of reconfigurable resources. The last paper of this section, "*3D Network-on-Chip Architectures using Homogeneous Meshes and Heterogeneous Floorplans*," by V. de Paulo and C. Ababei proposed original 3D 2-layer and 3-layer NoC architectures using homogeneous networks on a separate layer and heterogeneous floorplans on different layers. A design methodology that consists of floorplanning, routers assignment, and cycle-accurate NoC simulation was implemented and utilized to investigate new architectures.

One domain addressed by this special issue is how security should be considered into FPGA architecture. For this reason 3 papers are dealing with this exciting domain. The paper of L. Sauvage et al., "*Exploiting Dual-Output Programmable Blocks to Balance Secure Dual-Rail Logics*," proposes a new logic, dedicated to FPGA, robust against side channel attacks. The presented approach gave evidence that differential placement and routing of an FPGA implementation can be done with a granularity fine enough to improve the security gain. One major issue in security concerns the way to generate true random number. The paper "*True-Randomness and Pseudo-Randomness in Ring Oscillator-Based True Random Number Generators*," by N. Bochard et al. deals with true random number generators employing oscillator rings. A mathematical analysis, a simulation model, and FPGA implementations are discussed in depth. The last paper concerning security "*Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification*" by S. Drzevitzky et al. elaborates on the discussion of proof-carrying hardware (PCH) as a novel approach to reconfigurable system security. PCH takes a key concept from software security, known as proof-carrying code, into the reconfigurable hardware domain.

The last two papers show how FPGA could be efficiently used for robotics applications. The first paper proposed by J. Kalomiros and J. Lygouras "*Robotic Mapping and Localization with Real-Time Dense Stereo on Reconfigurable Hardware*" discusses about a new reconfigurable architecture for dense stereo and an observation framework for a real-time implementation of the simultaneous localization and mapping problem in robotics. The second paper "*A Reconfigurable System Approach to the Direct Kinematics of a 5 D.o.f Robotic Manipulator*" by D. F. Sánchez et al. describes an FPGA implementation of kinematics of a spherical robot manipulator using floating-point units operators. The proposed architecture was designed using a Time-Constrained Scheduling. Synthesis results show that the proposed hardware architecture for direct kinematics of robots is feasible in modern FPGAs families.

We hope you enjoy reading these papers, related to reconfigurable computing, and will serve the community to share new results in this field of research.

We sincerely thank authors for their valuable contributions and all reviewers for their help to ensure the quality of this special issue. We hope that you enjoy the articles in the ReConFig 2009 special issue and find its contents useful and give readers a good idea of where researchers have been focusing, both on long-studied problems still needing more work and on newer challenges.

Please stay tuned for the coming issues of the International Journal on Reconfigurable Computing and FPGAs.

Lionel Torres
Viktor K. Prasanna

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 492560, 15 pages
doi:10.1155/2010/492560

Research Article

RapidRadio: A Domain-Specific Productivity Enhancing Framework

Jorge A. Surís,[1] Adolfo Recio,[2] and Peter Athanas[2]

[1] *Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*
[2] *Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic and State University, Blacksburg, VA 24067, USA*

Correspondence should be addressed to Jorge A. Surís, jasuris@gmail.com

Received 7 March 2010; Revised 5 July 2010; Accepted 30 November 2010

Academic Editor: Viktor K. Prasanna

The RapidRadio framework for signal classification and receiver deployment is discussed. The framework is a productivity enhancing tool that reduces the required knowledge-base for implementing a receiver on an FPGA-based SDR platform. The ultimate objective of this framework is to identify unknown signals and to build FPGA-based receivers capable of receiving them. The architecture of the receiver deployed by the framework and its implementation are discussed. The framework's capacity to classify a signal and deploy a functional receiver is validated with over-the-air experiments.

1. Introduction

With the ever increasing processing power of general purpose processors and FPGAs, radio platforms are no longer tied to using static hardware. Using more flexible hardware platforms allows the radio to adapt to its environment, modifying the modulation scheme, symbol rate, or other link properties to maximize the efficiency of the communications link. This added flexibility then results in a volatile communications environment in which a participant may not know in advanced the properties of the link. This is often the case in signal intelligence applications where the user is attempting to become a silent third-party participant in a private communication. In this case, the user must first determine the link properties (modulation type, carrier frequency, symbol rate, etc.) and then create a receiver with the desired properties. Given the complexity of both of these tasks, it is desirable to automate the process.

Determining the properties of the physical communications link is a difficult problem and has been the focus of a large number of research efforts [1]. Although great strides have been made in this area, large assumptions about the environment or the signal are made that limit the applicability of the proposed techniques. Additionally implementation of a receiver for the newly classified signal is rarely addressed. When addressed, mostly software-based solutions are offered. The flexibility provided by FPGAs makes them good candidates for the implementation of the receivers, but the complexity associated with the design and development of FPGA-based solutions has relegated them to be second-class citizens. Instead of exploiting the full potential of the FPGA, radio designers implement only what is absolutely necessary on the FPGA. This strategy has worked so far because signal processing is performed on high-end processors on a PC. For embedded systems, however, the power requirements of these processors are unacceptable.

In this paper, the RapidRadio framework for signal classification and receiver deployment is discussed. RapidRadio is a domain-specific productivity enhancement tool that aims to reduce the knowledge base required for the development of radio receivers. Many productivity enhancing tools provide a design environment where the user can specify a series of high-level blocks that compose the system [2, 3]. Although these tools abstract away part of the design process, the user must still understand the block's interfaces and how they are interconnected. By narrowing the scope of the tool suite to a specific task, such as receiver deployment, RapidRadio can abstract most of the system architecture away. The user

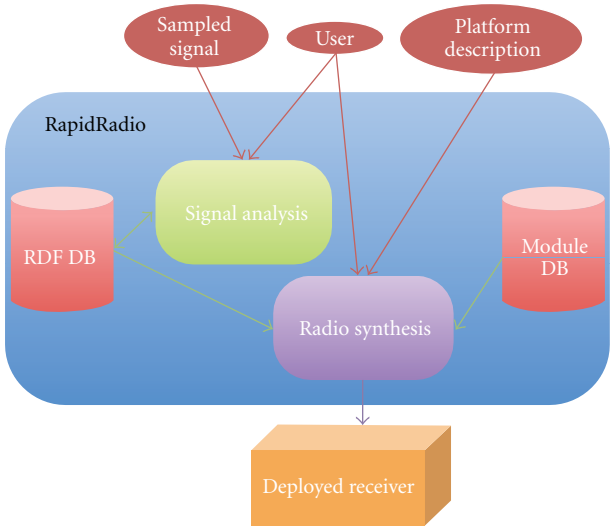


Figure 1: RapidRadio system architecture.

need only be concerned with signal analysis activities, and the framework will automatically identify the parameters of the physical link layer and build a receiver for it.

2. Rapid Radio Framework

RapidRadio divides the process of radio creation into two phases: the analysis phase and radio synthesis phase. The analysis phase guides the user through the process of classifying an unknown signal and determining its modulation scheme and parameters, cast in the domain of a signal intelligence analyst. Various transforms are provided to the user and the results of these transforms are presented. The term "transform" here is used in the abstract sense and refers to the process of applying various operations on continuous signal sets to accentuate certain features. Using high-level transforms shield the user from the underlying operations. The user, for example, may need to determine if a signal is synchronized or not but does not need to know how the synchronization is being performed. In addition to presenting the results of the transforms to the user, an expert system analyzes the results and suggests a possible course of action to the user. The output of the analysis phase is an abstract representation of the architecture of the receiver called the Radio Description File (RDF). The receiver synthesis phase transforms the platform independent RDF into a platform-specific description of the receiver. Figure 1 shows the high-level architecture of the RapidRadio framework and how these two phases interact.

Signal detection and classification is an inherently difficult problem. Before any classification is attempted, the signal desired must be identified. Most classification systems depend on some artificial intelligence structures [4, 5] to make decisions. The problem with this approach is that the artificial intelligence structures need to be trained a priori. This makes it hard to expand the system to classify new signal types. The RapidRadio framework uses the "human in the loop" approach to address this problem. "human in the loop" implies that the user is prompted to validate the decisions made by the framework. All the data used to make a decision is presented to the user. If the user disagrees with the decision made by the framework, then the framework's decision can be overridden. The RapidRadio framework utilizes the "plug-in" concept to coarsely describe various modulation and synchronization strategies that it can use in its search process. For example, constellations tested are specified in the form of an XML file. Adding a new constellation for consideration is easily accomplished by creating an XML file for the constellation. The interaction between the user and the framework is shown in Figure 1. The signal analysis part of the framework is discussed in Section 3 followed by a discussion of the receiver deployment phase in Section 4. Results from simulations and over-the-air experiments are presented in Section 6.

3. Signal Analysis

Signal analysis is the process through which the modulation scheme used for a signal transmission is identified. A large body of work has concentrated on Automatic Modulation Classification (AMC), but typically many assumptions are made about the signal that simplify the classification process [6, 7]. In many cases, the system has a priori knowledge about the signal being classified [8, 9]. In such cases, making assumptions such as perfect synchronization is reasonable. When working with the emergency bands, for example, it may be acceptable to assume that the receiver has a priori knowledge of the signals because the number of users is limited to the police, fire-fighters, and rescue personnel.

The RapidRadio framework is designed to perform blind signal classification of single-carrier, linearly modulated signals. As mentioned above, many contemporary signal classification systems assume perfect synchronization, which is difficult to obtain if the modulation scheme is unknown. RapidRadio follows a different path to signal classification that allows it to avoid such assumptions. Figure 2 shows a high-level description of the signal analysis process. First, the spectrum is sampled and displayed. The user then selects the area of interest in the spectrum. The selected signal is brought down to baseband and a spectral fitting is done to obtain the basic modulation parameters. For each hypothesized constellation, synchronization is attempted. Lastly, a set of metrics are obtained to determine the correctness of each hypothesis based on the synchronized signal. The user can evaluate the metrics and determine which hypothesis is correct. A Matlab front-end, with a CLIPS [10] back-end, is used to allow the user to interface with the RapidRadio framework. The steps involved in the analysis phase are discussed in more detail in subsequent sections.

3.1. Signal Selection and Isolation Interface. Upon initialization of the framework, the user is presented with a graphical user interface that allows him or her to obtain a sample of the spectrum, isolate a small portion of the spectrum, and

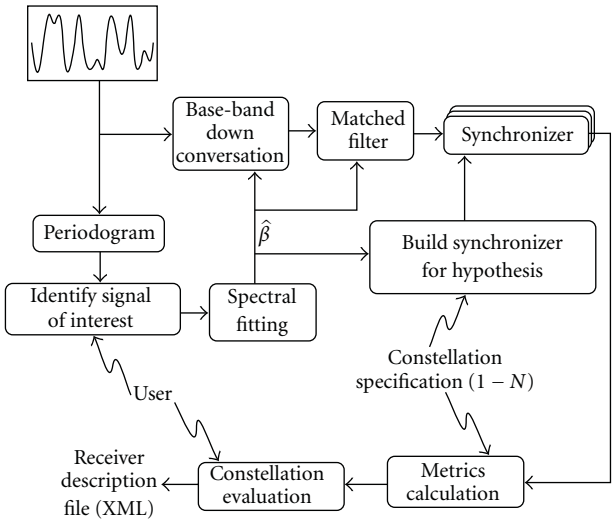


Figure 2: High-level flow of the analysis phase.

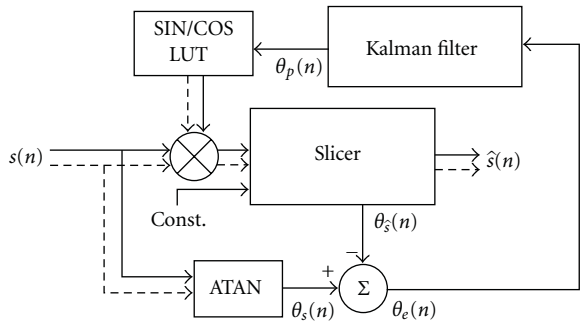


Figure 3: RapidRadio derotator architecture.

estimate the modulation parameters. Additionally, the CLIPS engine is initialized. CLIPS is a rule-based expert systems that makes inferences based on a set on known "facts" and rules. Facts represent what the expert system knows or has observed. The rules are used to interpret to indicate how facts should be interpreted. Initializing the framework eliminates transient facts developed during a previous run from affecting future runs. A set of default facts stored in CLIPS, such as the location of the constellation description files, are used to initialize the framework. The framework then loads the constellations into memory and adds them to the CLIPS fact set.

The operating environment is then added to the CLIPS fact-set. Using predefined rules, CLIPS defines the probability of each constellation based on the in-use environment. The RapidRadio framework includes three predefined operating environments.

(i) *Unknown*: there is no knowledge about the environment. All constellations are assumed to be equiprobable.

(ii) *Urban*: a high-noise environment is assumed. Constellations with less than four bits per symbol are assumed to have a higher probability than those with four or more bits.

(iii) *Microwave*: a low-noise environment is assumed. Constellations with four or more bits per symbol are assumed to have a higher probability than those with three or less bits.

These predefined environments were defined for testing and validation purposes only and may not accurately represent reality. All results shown in this paper were obtained using the Unknown environment. Environments can be easily added or altered by modifying the CLIPS rule set. This permits the user to taylor the environment to his or her operational conditions by reflecting a priori probabilities of certain constellations. New rules are automatically used, because the rule-sets are always reloaded during initialization.

3.2. Spectrum Sampling. Obtaining a sample of the spectrum is also done at the end of system initialization. The platform configuration used to sample the spectrum and the sampling frequency are obtained from the CLIPS fact-set. The Matlab front-end establishes a TCP/IP link with a daemon running on the target embedded platform. Through this link, the front-end configures the radio platform and extracts a sample of the signal. The signal is then converted to the frequency domain to allow examination of the spectrum.

The framework's GUI presents the periodogram, using the Bartlett method [11], of the captured signal. The user can then select a portion of the spectrum to analyze, by zooming into the area of the periodogram containing the signal. The area selected is used to obtain a crude estimate of the carrier frequency ($\hat{f}_c$).

3.3. Modulation Parameter Estimation. Throughout the analysis phase, only two assumptions about the signal are made. The first is that the signal is linearly modulated. Although this is a limiting factor, it is not an unreasonable assumption because linear modulations have wide-spread use. The second assumption made is that a root of raised cosine filter is used at the transmitter to limit bandwidth. This is a very common filter because it limits the bandwidth of the signal and reduces intersymbol interference. Because the ideal shape of the signal is known, an estimate of the carrier frequency ($\hat{f}_c$), symbol rate ($\hat{f}_s$), the roll-off factor ($\hat{\alpha}$), and signal-to-noise ratio (SNR) can be obtained by fitting the spectral shape of the incoming signal to the model. Finding the best fit is then a nonlinear Least Square optimization problem. This process is discussed in detail in [12]. The initial estimates for the unknowns are obtained from the user's specification of the signal to analyze.

3.4. Symbol Timing Recovery. Symbol timing recovery circuits attempt to determine the optimal moment in which to sample the incoming signal to extract the transmitted symbol. Many symbol timing recovery circuits, with varying levels of flexibility, have been proposed in the literature.

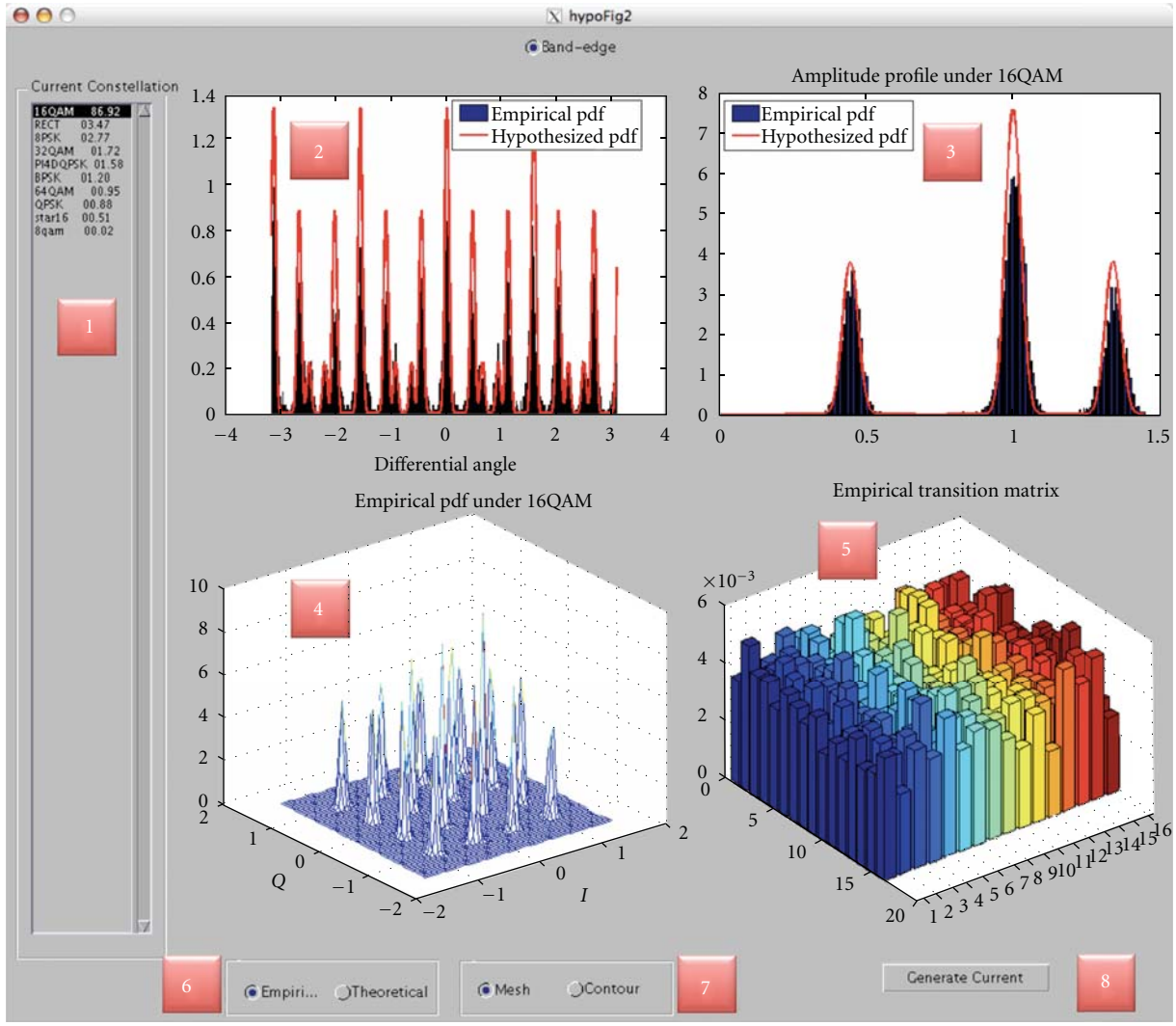


Figure 4: RapidRadio hypothesis evaluation interface showing the deployment of an inferred 8QAM receiver.

In [13], an early/late synchronization algorithm is presented, capable of operating a various symbol rates, but a training sequence is required. An efficient phase-independent synchronizer is presented in [14], but only phase modulated signals are supported.

To provide maximum flexibility, the chosen timing recovery architecture should work with little to no change for all linear modulation schemes. The architecture must also be rotation agnostic because carrier recovery is done after symbol synchronization. For RapidRadio, a modified passband timing recovery synchronizer with an oversampling rate of four is used [15, 16]. This architecture was chosen because it extracts the timing information from a spectral component in the signal, making it independent of the actual modulation scheme being used. A spectral line generator is used to extract the symbol timing from the received signal. The spectral generator produces a sinusoidal signal with a frequency equal to the actual symbol frequency. A local, smoother copy of the signal is produced using a phased-locked loop (PLL). Using the accumulator of the PLL, the correct timing instant is calculated. A Lagrange interpolator is used to obtain the value of the signal at the correct instant.

3.5. Carrier Recovery. The output of the timing recovery circuit is a stream of symbols. Each symbol is represented by the value of two signals; the in-phase (I) and the quadrature (Q) signals. They are represented in a two-dimensional plane where the x axis represents the magnitude of the in-phase signal and the y axis represents the magnitude of the quadrature signal. In the presence of carrier frequency error, symbols rotate on the I/Q plane and need to be derotated. An efficient derotation architecture is presented for 16QAM in [17, 18]. The RapidRadio architecture expands on this work to increase the derotator's flexibility and tolerance to frequency errors. The derotator architecture can be seen in Figure 3.

The incoming symbol ($s(n)$) is derotated using the predicted phase error ($\theta_p(n)$). A constellation-specific slicer is used to determine the closest constellation symbol ($\hat{s}(n)$). The slicer also produces the phase of $\hat{s}(n)$ ($\theta_{\hat{s}}(n)$). The

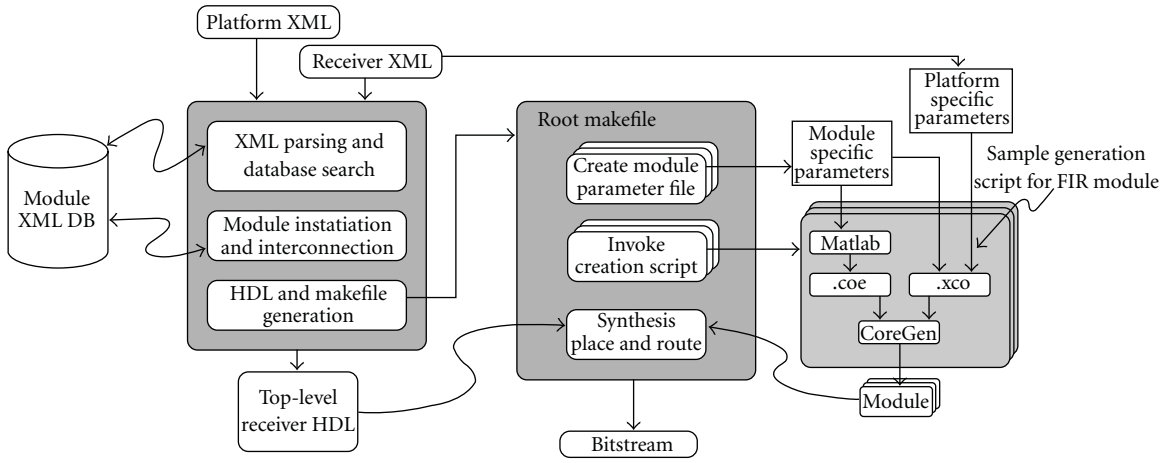


Figure 5: Radio synthesis flow.

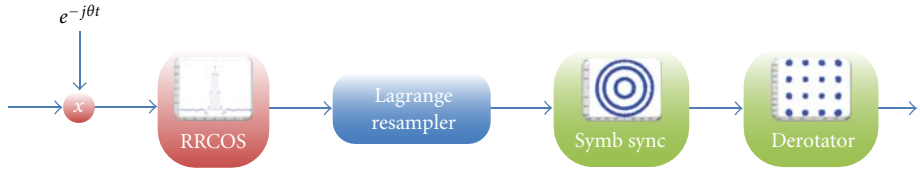


Figure 6: RapidRadio receiver architecture.

measured phase error ($\theta_e(n)$) is the difference between the phase of the $s(n)$ ($\theta_s(n)$) and $\theta_{\hat{s}}(n)$. The traditional PLL is replaced with a Kalman filter that tracks both the phase error and the frequency (f_d).

3.6. Hypothesis Fitness Evaluation. For each hypothesized constellation, the output of both the symbol synchronizer and the derotator is evaluated to determine the correctness of the demodulation. The evaluation is based on four metrics: the phase profile (pp) which measures the change in phase between two consecutive symbols, the amplitude profile (ap) which measures the magnitude of the received symbols, the symbol distribution (sd) which measures how the symbols are distributed in the I/Q plane, and the final metric is the symbol transition matrix (tm) which measures how often a transition between two specific symbols is observed.

All four metrics are calculated by comparing the theoretical probability distribution function (PDF) of the data with the empirical PDF. The empirical PDF is obtained by grouping the received data points in a histogram. Bin sizes for the histogram are calculated using Scott's formula for optimal bin size [19]. The count for each bin is then divided by the total number of data points obtained. The theoretical PDFs are obtained from models adjusted for the level of noise observed. The two PDFs are compared using the Hellinger distance (H) [20, 21]. The Hellinger distance is a measure of how similar two PDFs are. It takes values in the range of $(0, 1)$, where a zero indicates the PDFs are identical and a value of one indicates they are completely different.

The hypothesis evaluation GUI, shown in Figure 4, presents both the theoretical and empirical PDFs for the metrics in graphical form for evaluation by the user. The panel on the left marked with a one displays all the hypothesized constellations along with their scores. The hypotheses are ordered based on their likelihood, with the most likely constellation on the top. By selecting a constellation in this panel, the user can view the data used by the framework to formulate its decision. Navigating through the different hypotheses allows the user to make an informed decision on whether the framework chose the correct constellation or not. Areas marked two through five show the data used for the phase profile, the amplitude profile, the symbol distribution and the transition matrix, respectively. Both the theoretical and the empirical values are shown. Some datasets need to be presented in 3 dimensions making it impossible to present the theoretical and empirical values in the same chart, so buttons are provided to allow the user to switch between the two (Areas six and seven). Lastly a button to indicate to the framework to generate the radio for the provided constellation is shown in area eight.

3.7. Phase Profile. The phase profile examines the change in phase (θ) between consecutive received symbols. The theoretical PDF is obtained using the valid transitions for the hypothesized constellation. This does not account for errors due to signal noise however. Given that an estimate of the noise is obtained by the parameter estimator (see Section 3.3), the theoretical PDF can then be adjusted to reflect the phase variance due to the variance in the symbols.

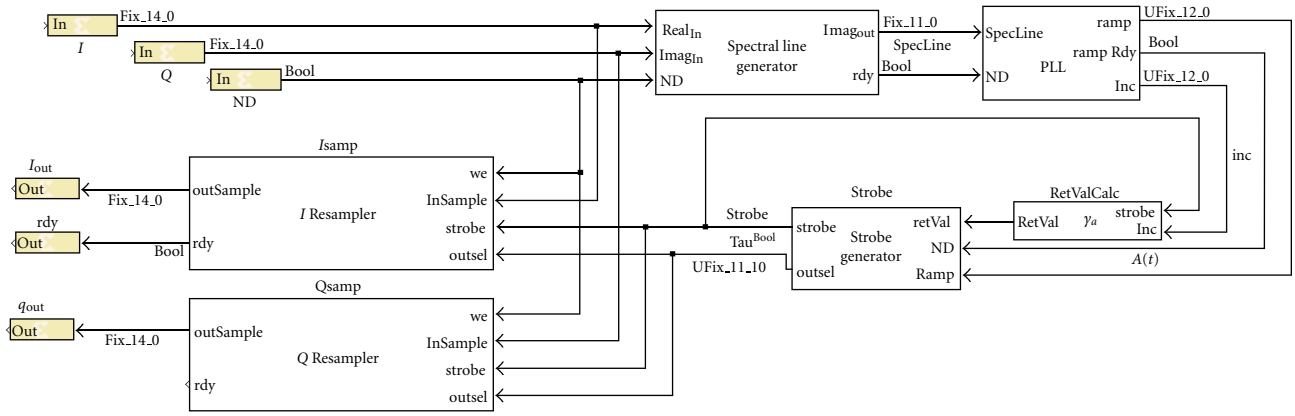


Figure 7: Symbol synchronizer implementation.

The PDF of the phase difference between two vectors perturbed by Gaussian noise is given as [22]

$$f(\psi) = \frac{1}{2\pi}\int_0^{\pi/2} e^{-\rho(1-\cos(\psi)\cos(\theta))}\left(1+\rho+\rho\cos(\psi)\cos(\theta)\right)\cos(\theta)\mathrm{d}\theta, \tag{1}$$

where $\rho = 1/\sigma^2$, σ^2 is the variance of the Gaussian noise, and $\psi = \theta(t) - \theta(t-1)$ with zero mean. Let ζ equal the expected phase difference for a symbol transition and ψ_n the phase difference with mean n; then

$$f(\psi_\zeta) = f(\psi_0 - \zeta). \tag{2}$$

The theoretical PDF is then

$$f_{\mathrm{pp}|c}(x) = \sum_{i=0}^{N-1} P(\theta = \zeta_i) f(\psi_0 - \zeta_i), \tag{3}$$

where N is the total number of expected phase changes for a given constellation c.

This metric is obtained prior to derotation because the derotator modifies the symbols phase according to the hypothesized constellation. The rotation, however, can be expressed as a constant phase error (θ_e) and (3) can be restated as

$$f_{\mathrm{pp}|c}(x) = \sum_{i=0}^{N-1} p(\theta = \zeta_i) f(\psi_0 - \zeta_i - \theta_e). \tag{4}$$

The value of θ_e that results in the lowest Hellinger distance is used to calculate the value of the phase profile metric. The Hellinger distance for the phase profiles is then expressed as

$$H_{\mathrm{pp}|c} = H\left(h_{\mathrm{pp}}(x), f_{\mathrm{pp}|c}(x)\right), \tag{5}$$

where $h_{pp}(x)$ is the the empirical histogram.

3.8. Amplitude Profile. The Amplitude profile examines the distribution of the magnitudes for the received symbols and compares it to the theoretical values. As with the phase profile, the measured magnitudes are grouped up into bins to form a histogram (h_{ap}). The theoretical PDF is obtained from the constellation description and is assumed to have a Rice distribution:

$$f(x,\nu) = \frac{x}{\sigma^2}\exp\left(\frac{x^2+\nu^2}{2\sigma^2}\right) I_0\left(\frac{x\nu}{\sigma^2}\right), \tag{6}$$

where ν is the amplitude, σ^2 is the symbol variance, and I_0 is a Bessel function with order 0. The theoretical PDF is then given as

$$f_{\mathrm{ap}|c}(c) = \sum_{i=0}^{N-1} p(\nu = \nu_i) f(x,\nu), \tag{7}$$

where ν_i is the ith amplitude and N is the number of possible amplitudes for constellation c. The amplitude profile metric is the Hellinger distance between the actual distribution and the theoretical distribution and can be expressed as

$$H_{\mathrm{ap}|c} = H\left(h_{\mathrm{ap}}(x), f_{\mathrm{ap}|c}(x)\right). \tag{8}$$

3.9. Symbol Variance and Distribution. Symbol distribution examines the number of received symbols for each constellation point. Symbol variance is a measure of the distance between the received symbol and the theoretical location of the symbol. These two metrics are combined to form a two-dimensional PDF of the received symbols. Assuming that in-phase and quadrature components of the symbol are independent random variables with Gaussian noise, the joint PDF can be expressed as

$$f_j(i,q) = \frac{1}{2\pi\sigma^2}\exp\left(\frac{\left(i-I_j\right)^2 + \left(q-Q_j\right)^2}{\sigma^2}\right), \tag{9}$$

where I_j and Q_j are the expected in-phase and quadrature values for the jth symbol and σ^2 is variance of the noise. Assuming that the all symbols in a constellation are equally probable, the two-dimensional theoretical PDF is then

$$f_{\mathrm{sd}|c}(i,q) = \frac{1}{N}\sum_{i=0}^{N-1} f_i(i,q), \tag{10}$$

where N is the total number of symbols in constellation c.

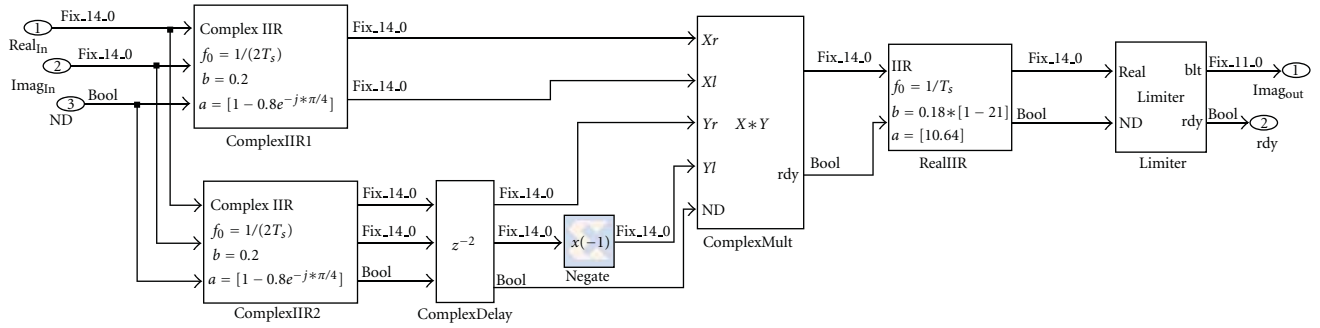


Figure 8: Spectral Line generator Diagram.

The empirical PDF is obtained using a two-dimensional histogram (h_{sd}) to group the received symbols into bins. The Hellinger distance can be used to measure the difference between the empirical and theoretical PDFs and is expressed as

$$H_{\text{sd}|c} = H\left(h_{\text{sd}|c}(x,y), f_{\text{ap}|c}(x,y)\right). \tag{11}$$

3.10. Symbol Transitions. This metric compares the distribution of symbol transitions to the theoretical values. An $M \times M$ matrix is populated with all the transitions observed, where the rows of the matrix represent the ith and the columns represent the ith+1 symbol. The matrix is then divided by the total number of observed transitions to obtain the empirical PDF. The theoretical PDF is obtained from the constellation's description file. For each symbol in the constellation, the description file has a list or range of valid symbols a given symbol can make a transition to. As with other metrics, the Hellinger distance is used

$$H_{\text{tm}|c} = H\left(h_{\text{tm}|c}(x,y), f_{\text{tm}|c}(x,y)\right), \tag{12}$$

where $h_{\text{tm}|c}(x,y)$ is the empirical transition matrix and $f_{\text{tm}|c}(x,y)$ is the theoretical transition matrix.

3.11. Combining the Metrics. All of the metrics discussed above provide information on the fitness of a hypothesis, but cannot on their own identify the correct hypothesis. A mechanism for combining the metrics in a manner that results in a single value that can be used to rank the hypothesized constellations is needed. Additionally, the mechanism must allow for easily modifying the probability of occurrence for any constellation. Artificial intelligence blocks such as neural networks were considered as a method for combining the metrics, but because they have to be trained, they lack the flexibility desired for the RapidRadio framework. A Bayesian network on the other hand does not require *a priori* training and has a natural mechanism for integrating the probability of occurrence of constellations.

Scoring is done in two stages. First the *a priori* probabilities of occurrence of all the constellations are pushed down to the processing node. At the same time the probability of each metric given a hypothesis is passed to the processing node. The new probability of each constellation is then calculated according to (13) below:

$$P(h_i \mid \text{pp}, \text{ap}, \text{sd}, \text{tm}) = \frac{P(\text{pp}, \text{ap}, \text{sd}, \text{tm} \mid h_i) P(h_i)}{\sum_{i=0}^{N-1} \left[P(\text{pp}, \text{ap}, \text{sd}, \text{tm} \mid h_i) P(h_i)\right]}. \tag{13}$$

Including the *a priori* probability of the hypothesized constellations in (13) ensures that constellations that are known to more likely have a higher probability of being chosen. pp, ap, sd, and tm are conditionally independent which allows the dividend of (13) to be defined as

$$\begin{aligned} &P(\text{pp}, \text{ap}, \text{sd}, \text{tm} \mid h_i) \\ &\quad = P(\text{pp} \mid h_i) P(\text{ap} \mid h_i) P(\text{sd} \mid h_i) P(\text{tm} \mid h_i). \end{aligned} \tag{14}$$

The probabilities for each metric are approximated as follows:

$$\begin{aligned} P(\text{pp} \mid h) &= 1 - H_{\text{pp}|h}, \\ P(\text{ap} \mid h) &= 1 - H_{\text{ap}|h}, \\ P(\text{sd} \mid h) &= 1 - H_{\text{sd}|h}, \\ P(\text{tm} \mid h) &= 1 - H_{\text{tm}|h}. \end{aligned} \tag{15}$$

This approximation assigns higher probabilities to hypothesis with empirical PDFs that closely resemble the theoretical PDFs. Using the Hellinger distance for all metrics ensures that they are equally weighed. The result of the Bayesian network is a set of probabilities that indicate the likelihood of all constellations. Situational awareness and knowledge from past experiences can be inserted into the model by modifying the *a priori* probabilities of each constellation. From (13), it can be seen that the joint probability of the metrics given the hypothesized constellation is normalized by the sum of the joint probabilities of the metrics for all hypothesized constellations. This normalization allows the addition of new constellations with little to no change to the classification algorithm.

4. Radio Deployment Phase

The radio deployment phase creates an FPGA-based receiver for the classified signal that produces a stream of symbols. Using a TCP/IP link, the framework starts the build on a server that hosts the vendor tools. When the build process has

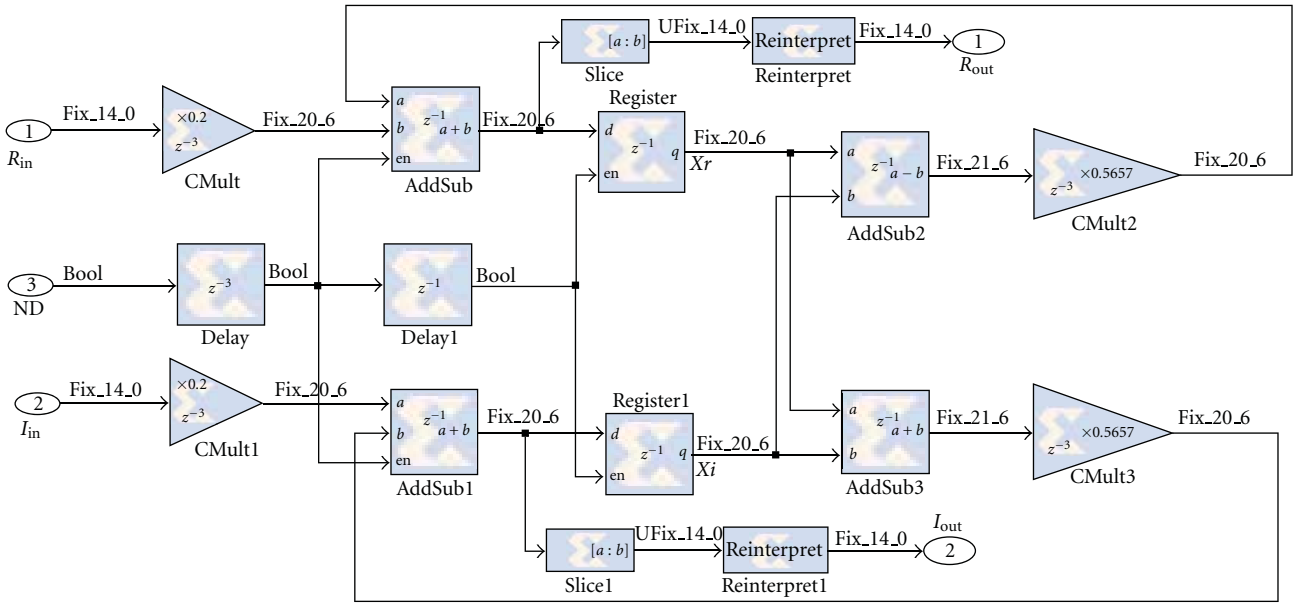


Figure 9: Complex IIR.

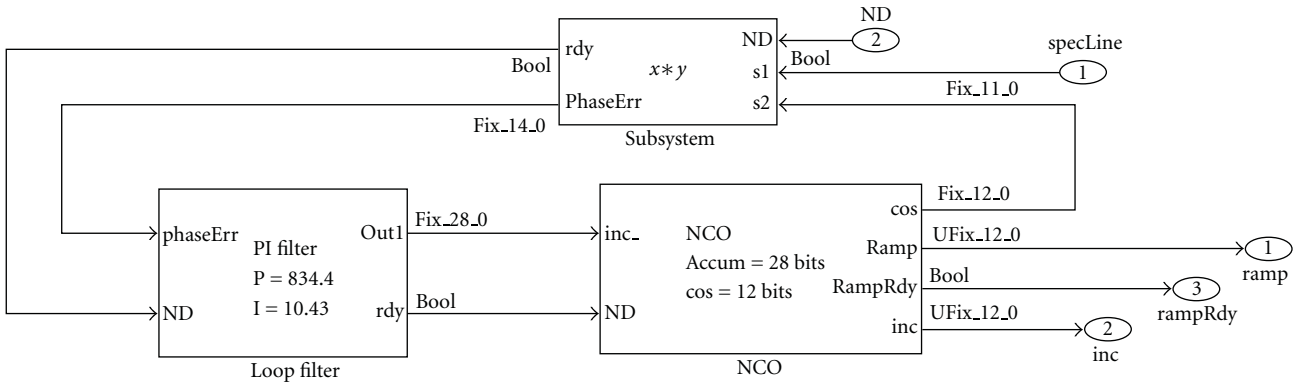


Figure 10: PLL Implementation.

completed, the new configuration bitstream is loaded into the target platform. A set of 81,000 noncontiguous symbols is then extracted from the platform and displayed on the user interface. A high-level description of the radio synthesis phase can be seen in Figure 5. The synthesis tool developed is written in C++ and builds on-top of other tools such as Makefiles, Xerces-C, Matlab, and Xilinx's Core Generator. The following sections discuss the major pieces of the radio synthesis phase.

4.1. Platform Description File. A goal of the RapidRadio framework is to reduce the amount of FPGA knowledge necessary to create a system. There are many parameters, however that are platform unique, such as ADC initialization, intercomponent communications, and output pin usage to name a few. To hide this information from the user, the framework could be made platform specific, building all the knowledge about a given platform into the system. This approach however is not very attractive because it would make the framework hard to modify to target a different platform.

The RapidRadio framework solves this problem by assuming that a top level design was previously created. This top level design serves as a host to the receiver and takes care of all the initialization and communication infrastructure. The framework therefore produces a *receiver* module which accepts an input data stream and produces an output data stream. This approach requires a one-time setup when a new platform is chosen to serve as a target, but the cost of designing and testing the top level design is relatively small when compared to the cost of building an entire new design every time.

4.2. Module Selection and Generation. The module database shown in Figure 5 does not contain the modules themselves, but module description files written in XML. The file can represent prebuilt modules or a set of rules dictating how to create a module. Definitions of the interfaces for the module

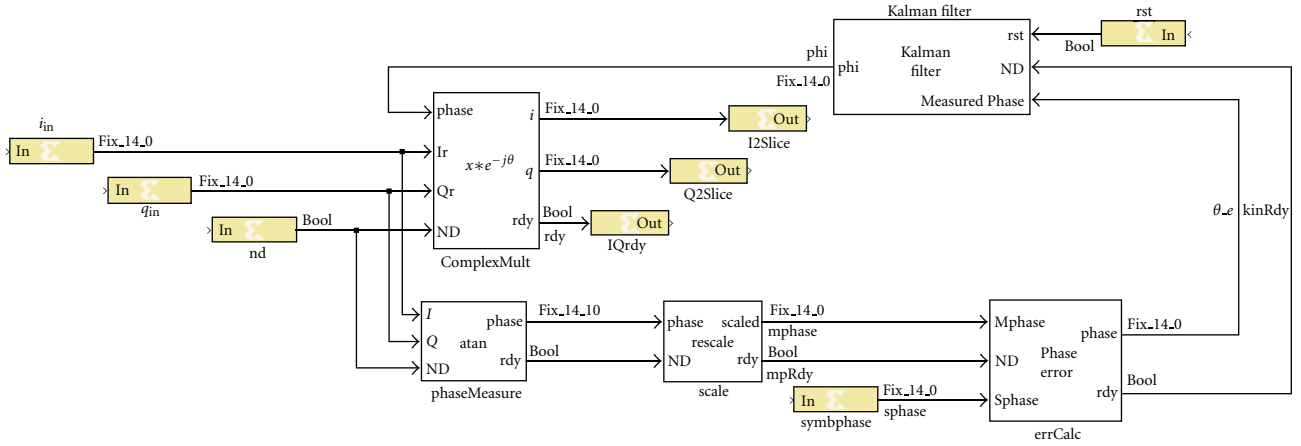


Figure 11: Derotator FPGA implementation.

and a list of all the ports associated with each interface are also contained. If the implementation is not device specific or the module has not yet been implemented, the description file will indicate that it can match any device. Otherwise the device type information is listed. Lastly all module parameters are listed in the file. The module description file uses some of the tags previously defined for the RDF and the platform description file.

When a module is encountered in the RDF, a database is searched for a matching module. Matches are based on the modules type/subtype, parameters, and the target device. Parameters that are hard-coded into the modules implementation are shown as having a specific value. If the database module does not have the same value as that requested in the RDF, then it is not considered a match. If the database module has a value of "configurable" for the parameter, then this parameter is ignored during the match process because the module can be configured to use the value provided in the RDF.

5. Receiver Architecture and Implementation

Receivers deployed by the RapidRadio framework share the general architecture shown in Figure 6. The signals sampling rate at each block is indicated by the blocks color. A complex mixer is used to bring the signal down to baseband. Decimating root-raised cosine filters are used for matched filtering and sampling rate reductions. A resampler circuit is used to recondition the signal, reducing the sampling rate to four times the symbol rate. Synchronization is then performed to produce a stream of symbols. The following sections describe in detail the FPGA implementation of these blocks. All blocks were designed for 14-bit interfaces. The bit-width of internal registers was determined at design time in such a manner as to prevent overflow with full-strength signals.

5.1. Downconverter. Using an automatically generated multiplier block and a 14-bit NCO, the signal is brought down to baseband. The multiplier is created using Xilinx's Core Generator. The output width of the multiplier (w_o) is inherited from the platform description file. The width of the multiplicands can be different because one is the output of the NCO which has a fixed output width of 14-bits and the other is the input data which has a width equal to the input width (w_{in}). The output width of the multiplier is then

$$w_o = 14 + w_{\text{in}}. \tag{16}$$

The effective output of the multiplier is $w_o - 1$ because both inputs are signed. To avoid bit growth, the output of the multiplier is truncated, to keep it the same size as the input. The MSB of the multiplier block is

$$\text{MSB} = w_o - 2, \tag{17}$$

and the LSB is

$$\text{LSB} = \text{MSB} - w_{\text{in}} + 1. \tag{18}$$

5.2. Matched Filtering. Matched filters are also created using the Core Generator. The roll-off factor is obtained from the RDF. The filter order is dependent on the number of samples per symbol at the filter's input and on the roll-off factor. The roll-off factor is used to determine how many symbols the filter should contain to minimize ISI. The length of the filter in symbols (l_s) is defined as

$$f_l = 4 * \left\lceil \sqrt{\frac{2}{\alpha}} \right\rceil. \tag{19}$$

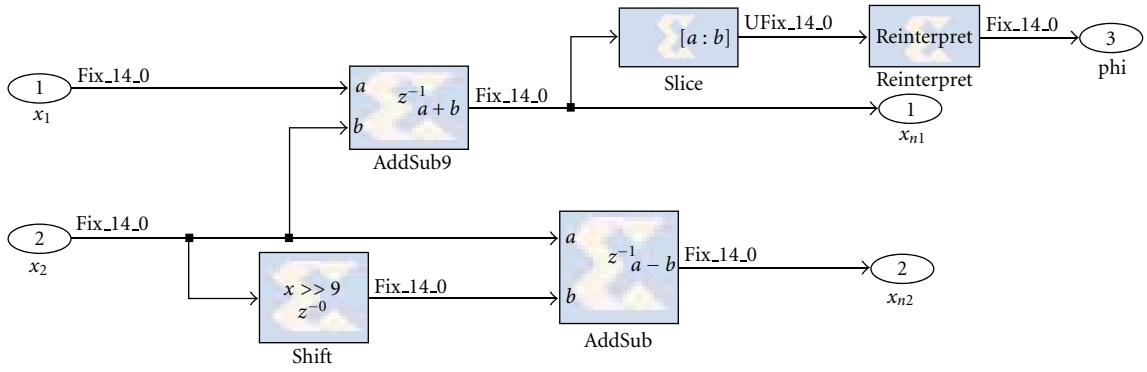
The number of samples per symbol at the input is:

$$s_s = \left\lfloor \frac{F_s}{f_s} \right\rfloor. \tag{20}$$

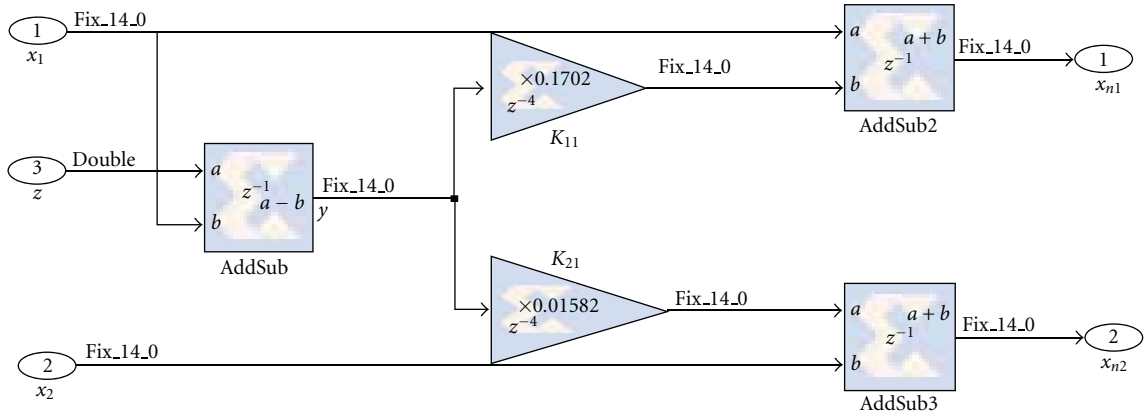
The filter order is then defined as

$$\max(64, s_s \cdot f_l). \tag{21}$$

The filter's decimation factor (η) is a function of the filter's input sampling rate (F_s) and the minimum required



(a) Prediction stage



(b) Update stage

Figure 12: Kalman filter FPGA implementation.

input sampling rate of the resampler (F_{sr}). η is calculated by the CLIPS back-end. Filter coefficients are obtained from Matlab using the sampling rate, symbol rate, and roll-off factor. Coefficients are then scaled and stored into a ".coe" file.

A filter of order N has $N + 1$ multiplications and N additions. Assuming that the filter coefficients are the same size as the input data, the width of the output (w_o) is approximately

$$w_o = 2w_{\text{in}} + \left\lceil \log_2\left(\frac{N+1}{\eta}\right)\right\rceil, \tag{22}$$

where w_{in} is the input width. As with the downconverter, the output of the filter is truncated to avoid bit-growth. Selecting which bits of the output will be used however is not an obvious choice. Selecting the w_{in} MSB will guarantee that the output never overflows, but may result in underflow. To obtain an estimate of which bits to use, the scaled coefficients are used to process the sampled data obtained in the spectrum sampling phase (see Section 3.2). The magnitude of the filtered signal is then used to determine the MSB:

$$\text{MSB} = \left\lceil \log_2(\max(x_s))\right\rceil, \tag{23}$$

where x_s is the filtered signal. This methodology reduces the possibility of overflow at the output of the filter, as long as the signal level is similar to that encountered during the analysis phase.

5.3. Resampler. The RapidRadio framework assumes that it does not control the sampling rate of the input data stream. The synchronization circuit however, requires an input sampling rate of $4f_s$. To obtain the sampling frequency required by the synchronizer, a resampler circuit is used. The resampler uses 16-bit accumulator and a Lagrange interpolator to produce a copy of the input signal at a lower sampling rate. Aliasing is avoided because the matched filters eliminated higher frequency components.

The accumulator is incremented by a fixed value κ for every input sample received. A new output sample is desired when the value of the accumulator is zero. The increment is calculated as follows:

$$\kappa = \frac{2^{16} f_{\text{out}}}{f_{\text{in}}}, \tag{24}$$

where f_{in} is the input sampling rate and f_{out} is the output sampling rate. When the accumulator wraps, the interpolator uses the last three samples received and the value of the

accumulator ($A(t_n)$) to create the new sample. The interpolation is then performed over the sample set $\{t_{n-2}, t_{n-1}, t_n\}$. Assigning time indexes -1, 0, and 1, respectively, to the samples gives the interpolation polynomials in (25).

$$\begin{aligned} \ell_0 &= \frac{x(t_n) + x(t_{n-2})}{2} - x(t_{n-1}), \\ \ell_1 &= \frac{x(t_n) - x(t_{n-2})}{2}, \\ \ell_2 &= x(t_{n-1}), \\ \widetilde{x}(\tau) &= \tau^2 \ell_0 + \tau \ell_1 + \ell_2, \end{aligned} \tag{25}$$

where $x(t)$ is the value of the original signal at time t and $\widetilde{x}$ is the resampled signal. For the resampler, the desired sampling instant (τ) is the distance between the sample at time t_{n-1} and the maximum value of the accumulator (A_{mx}) normalized to κ. τ is then expressed as

$$\tau = \frac{A_{\text{mx}} - A(t_{n-1})}{\kappa}. \tag{26}$$

5.4. Timing Recovery Implementation. The implemented architecture of the symbol timing recovery module is shown in Figure 7. The spectral line generator and one of the IIR filters are shown in Figures 8 and 9, respectively. Internally the IIR filter uses six bits for decimal representation in addition to 14 bits for integer representation. By only truncating to 14 bits at the output of the filter, the truncation error is reduced.

Notice that only the imaginary part of the complex multiplication is required for the spectral generation circuit. This permits the optimization of the multiplication to only require two multiplications and one addition. Given that the inputs to the multiplier are four 14 bit numbers, the output could require up to 29 bits of precision. Matlab simulations, however, showed that only 25 bits are required when the input signal to the synchronizer is fully scaled to $\pm 2^{13}$. To maintain a signal of 14 bits, the output of the multiplier is rescaled by shifting it right 11 bits. The output of the third filter is then passed through a limiter circuit that saturates at ± 725 to eliminate any amplitude modulation.

In the PLL, shown in Figure 10, the phase difference between the signal generated by the spectral generator and a cosine generated by a local NCO is calculated. A multiplier is used to measure the phase error. A PI filter is used for the PLL's loop filter. Due to the small size of the loop filters coefficients, they cannot be properly represented using a reasonable number of bits. Obtaining representable values requires that the normalization factor (μ), that converts the phase error into an NCO accumulator offset, be moved into the filter. An additional factor of 2^3 is used to upscale the filter values because multiplying by μ does not result in sufficiently large values. Truncating the three least significant bits of the output of the filter provides the increment correction to the NCO. The NCO's increment and accumulator are then used to calculate the perfect timing instant (γ_a), the timing error (τ), and the strobe signal. Table 1 shows some of the values used for the timing recovery module.

Table 1: Symbol synchronizer implementation values.

Variable	Value
Loop filter proportional constant (α)	2.44×10^{-6}
Loop filter integrating constant (β)	3.02×10^{-8}
Phase error size	14 bits
NCO input size	12 bits
NCO accumulator size	28 bits
NCO ramp output size	12 bits
μ	$2^{28}/2\pi$
τ size	11 bits

Table 2: Derotator implementation parameters.

Variable	Value
θ_p size	14 bits
sin/cos size	12 bits
ATAN processing elements	9
Phase error scaling	$(2^{13} - 1)/\pi$
Kalman filter register size	14 bits
Kalman gain $K_{1,1}$	0.1702
Kalman gain $K_{2,1}$	0.01582

Table 3: Modulation parameters obtained from RapidRadio framework for test over-the-air signals. The first row shows the nominal values used by the transmitter. The second row shows the values used for receiver deployment.

Modulation	E_sN_0 (db)	Value type	α	f_c (Hz)	f_s (Hz)
32QAM	9.18	Nominal	0.65	2000000	100000
		Deployed	0.54	1999314	100010
16QAM	7.36	Nominal	0.65	2000000	250000
		Deployed	0.64	1999130	250039
8QAM	8.50	Nominal	0.75	2000000	350000
		Refined	0.74	1999806	350046
QPSK	12.56	Nominal	0.65	2000000	250000
		Deployed	0.68	1999211	250044

5.5. Derotator Implementation. The derotator is implemented in two blocks. The first is comprised of the static portion of the architecture that is not dependent upon the constellation. This block, shown in Figure 11, was created in System Generator. The complex mixer derotates the input signal using a predicted phase error and a sin/cos lookup table. The phase of the input symbol is calculated using Xilinx's CORDIC atan core. The output of the atan block, which is in the range of $[-\pi, \pi]$ is normalized to a power of 2 in order to facilitate further operations, such as wrapping. The phase output of the slicer is then used to calculate the phase error of the input symbol. The measured phase error is in the equivalent range of $[-2\pi, 2\pi]$ because it is calculated using a subtraction. The error is re-normalized to $[-\pi, \pi]$ by adding 2π if the value is less than $-\pi$ or adding -2π if the value is greater than π. Table 2 shows the implementation parameters for the derotator.

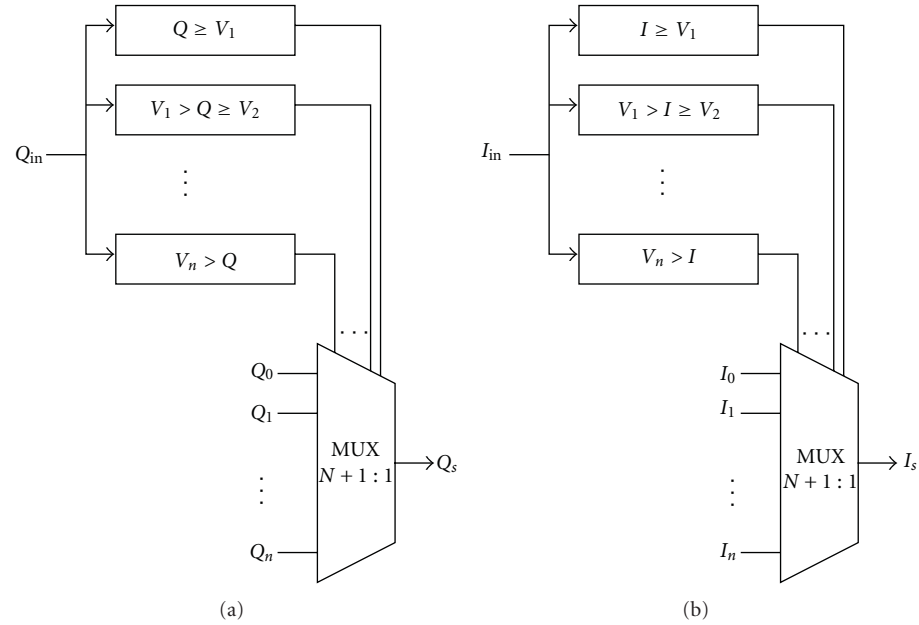


Figure 13: Grid-based slicer architecture.

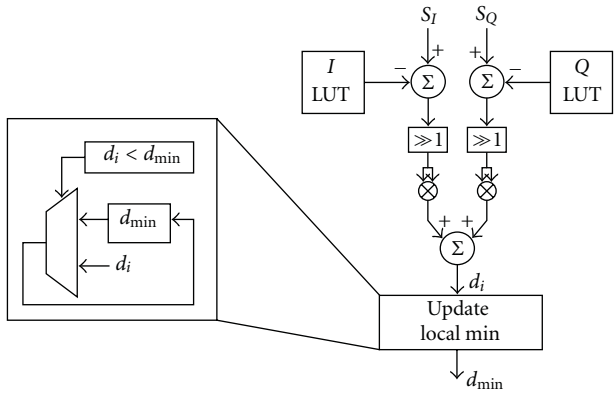


Figure 14: Distance block.

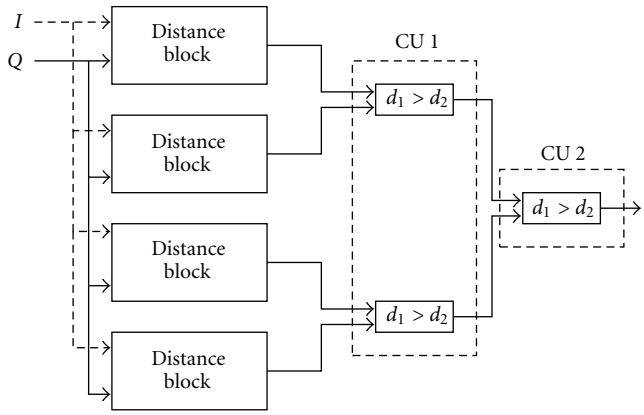


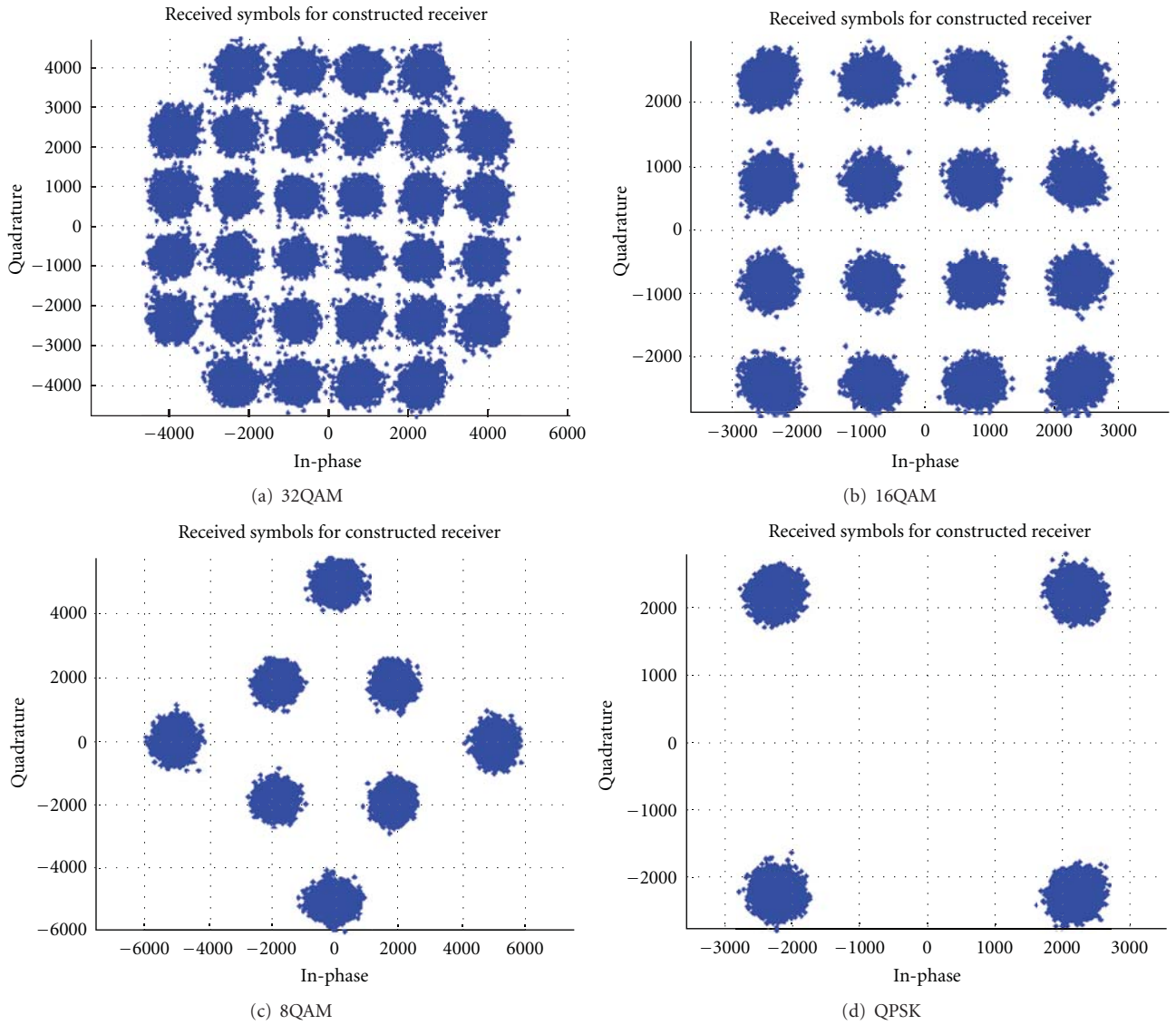
Figure 15: Distance-based slicer architecture.

As mentioned in Section 3.5, the Kalman gains are static and precalculated at design time. Their values are shown in Table 2. Figure 12 shows the FPGA implementation of both the prediction and the update stage of the Kalman filter.

Because the slicer block is constellation unique, it is generated at implementation time using a custom C++ HDL generator. A constellation name and the average signal power are provided as input. A Matlab engine is opened by the generator and the constellation description is loaded. All constellations are stored in XML files and contain a list of the constellation points. The magnitude of the constellation points has been normalized to obtain an average power of one. A Matlab script is then used to determine the slicer architecture to be used. Constellation points are scaled by the average power to ensure proper slicing.

The grid-based architecture is shown in Figure 13. Values V_1 through V_n are the decision boundaries. The input values are compared with the boundaries and the correct symbol is chosen using two $n + 1 : 1$ muxes, where n is the number of boundaries. This architecture is fairly efficient as it only requires $2n$ comparators and four lookup tables. All comparisons run in parallel, which provides a slicing latency of one clock cycle.

A distance-based slicer works for any constellation but is computationally expensive. For every point in the constellation, two subtractions, two multiplications, and one addition must be done. Additionally, $\log_2(M)$ comparisons are needed to determine the shortest distance, where M is the number of points in the constellation. Ideally every constellation point would be processed in parallel, but this would not take advantage of the pipelining built into the multiplier blocks. The resource requirements would also be excessive. For 64QAM, for example, 128 multipliers would



(a) 32QAM (b) 16QAM

(c) 8QAM (d) QPSK

Figure 16: Sample symbols extracted from the deployed receivers. Clustering of symbols indicates that synchronization was properly achieved.

Table 4: Resource utilization for deployed receivers on a Xilinx Virtex 4 XC4VLX60 FPGA, using a sampling rate of 8 MHz and an operating clock rate of 192 MHz.

			RRCOS filter		
Constellation	Order	MSB	DSP48	RAMB16	Slices
32QAM	608	31	23 (35%)	27 (16%)	7242 (27%)
16QAM	248	30	25 (39%)	31 (19%)	4348 (16%)
8QAM	176	29	25 (39%)	21 (13%)	5189 (19%)
QPSK	256	30	27 (42%)	33 (20%)	4403 (16%)

be required. A simple solution is to multiplex access to the multipliers.

The RapidRadio distance-based slicer uses a distance block for every four constellation points. Each block has two lookup tables containing the I and the Q values for four symbols. Calculating the distance between the input and each of the symbols in the table is accomplished using the circuit in Figure 14. Shifting the output of the subtractor right one bit keeps the signal 14 bits wide. The sum of the squares is then calculated (d_i). Each distance is compared to the previous minimum $d_{\min}$. If it is lower, then the minimum distance is updated to the current value. This process is

shown on the left side of Figure 14. The output of the distance block is the smallest distance and the location and phase of the corresponding symbol (the latter is not shown in Figure 14). Multipliers have a pipeline depth of four, and the addition and subtraction circuits have a latency of one cycle. With the the comparison and control logic, the distance block has a total latency of 10 cycles.

Slicing constellations with more that four symbols is accomplished by using more than one distance unit. For a given constellation, a total of $U = M/4$ distance blocks are required. A set of cascading comparison units is used to merge the output of the distance blocks to obtain the constellation point. Each comparison unit divides the L inputs into $L/2$ pairs. For each pair, a relational operator is used to eliminate the largest distance. A total of $\log_2(U)$ comparison units are necessary to obtain the results because each comparison unit reduces the number of candidate symbols by a factor of two. Given that each comparison units has a latency of one clock cycle, the distance-based slicer then has a total latency of $10+\log_2(U)$. Figure 15 shows the overall architecture of the slicer.

6. Results

This section discusses the performance of the RapidRadio framework prototype. Three signals were generated and transmitted over the air at 2.05 GHz. The framework was used to classify the signal, determine the modulation parameters, and synthesize the radio. The modulation schemes used for the tests were QPSK, 16QAM, and 32QAM. Both the transmitter and the receiver were implemented in the Harris SDR SIP package which contains four Xilinx Virtex 4 XC4VLX60 FPGAs. Separate systems and clock sources were used for the transmitter and receiver to ensure that synchronization is not achieved simply because the same clock is used for both systems.

For each signal, the constellation chosen by the framework was validated and the receiver deployment phase was started. The synthesis phase was executed on linux machine, using the standard vendor tools. A TCP/IP daemon running on the ARM processor in the development board then loads the newly created receiver bitstream onto the FPGA connected to the receiver RF chain. Figure 16 shows the extracted symbols for the test constellations. It can be observed that for all constellations, synchronization was properly recovered because the extracted symbols are clustered appropriately.

Table 3 shows the modulation parameters for all test signals. The first row shows the nominal values used by the transmitter. The second row contains the values obtained by the framework, which are used to deploy the receiver. It can be observed that the parameters obtained by the framework are very similar to the nominal values.

Table 4 shows the resource utilization for the deployed systems as well as the instantiation parameters for the matched filters. As expected the signals with the lowest data rate require high-order filters because of the high number of samples per symbol at the input sampling rate. The highest system utilization is observed by the constellations that require a distance-based slicer (32QAM and 8QAM).

7. Conclusion

In this paper, the RapidRadio framework for signal classification and receiver deployment was discussed. The framework guides the user through the process of determining the modulation type of an unknown signal and building an FPGA-based receiver capable or demodulating the signal. Reducing the scope of the framework narrows down the possible implementations and allows the hiding of implementation details.

Unknown signals were classified using the four metrics: the phase profile, the amplitude profile, the symbol distribution, and the symbol transition matrix. Using a Bayesian network, the metrics were combined to assign each hypothesis a probability. Using a Bayesian network provides added flexibility to the framework by ensuring that it can automatically adjust itself to new constellations.

The frameworks functionality was verified by capturing off-the-air signals. All signals were properly classified and functional receivers were deployed on a Virtex-4 FPGA.

Acknowledgment

The authors would like to thank the Harris Corporation, Government Communications Division for supporting this research.

References

[1] O. A. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, "Survey of automatic modulation classification techniques: classical approaches and new trends," *IET Communications*, vol. 1, no. 2, pp. 137–156, 2007.

[2] Xilinx Inc., *System Generator for DSP: Getting Started Guide*, Xilinx Inc., 2007.

[3] G. J. Minden, J. B. Evans, L. Searl et al., "KUAR: a flexible software-defined radio development platform," in *Proceedings of the 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN '07)*, pp. 428–439, Dublin, Ireland, April 2007.

[4] K. Kim, I. A. Akbar, K. K. Bae, J.-S. Um, C. M. Spooner, and J. H. Reed, "Cyclostationary approaches to signal detection and classification incognitive radio," in *Proceedings of the 2nd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN '07)*, pp. 212–215, Dublin, Ireland, April 2007.

[5] A. Fehske, J. Gaeddert, and J. H. Reed, "A new approach to signal classification using spectral correlation and neural networks," *Proceedings of the 1st IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN '05)*, pp. 144–150, November 2005.

[6] J. Y. J. Chen and R. Morelos-Zaragoza, "Design and implementation ofan algorithm for modulation identification of analog and digital signals," in *Proceedings of the Software Defined Radio Forum Technical Conference (SDR '06)*, SDR Forum, Orlando, Fla, USA, 2006.

[7] L. Liu and J. Xu, "A novel modulation classification method based on high order cumulants," in *Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM '06)*, pp. 1–5, September 2006.

[8] B. Le, F. A. G. Rodriguez, Q. Chen et al., "A public safety cognitive radio node," in *Proceedings of the Software Defined Radio Forum Technical Conference (SDR '07)*, SDR Forum, Denver, Colo, USA, November 2007.

[9] B. Le, T. W. Rondeau, D. Maldonado, D. Scaperoth, and C. W. Bostian, "Signal recognition for cogntive radios," in *Proceedings of the Software Defined Radio Forum Technical Conference (SDR '06)*, SDR Forum, Orlando, Fla, USA, 2006.

[10] "*CLIPS Reference Manual: Volume I Basic Programming Guide*," December 2007.

[11] P. Stoic and R. Moses, *Spectral Analysis of Signals*, chapter 2, Prentice Hall, New York, NY, USA, 2005.

[12] A. Recio, J. Surís, and P. Athanas, "Blind signal parameter estimation for the rapid radio framework," in *Proceedings of the Military Communications Conference (MILCOM '09)*, IEEE, October 2009.

[13] Y. Tachwali, W. J. Barnes, and H. Refai, "Configurable symbol synchronizers for software-defined radio applications," *Journal of Network and Computer Applications*, vol. 32, no. 3, pp. 607–615, 2009.

[14] P. Zicari, E. Sciagura, S. Perri, and P. Corsonello, "A programmable carrier phase independent symbol timing recovery circuit for QPSK/OQPSK signals," *Microprocessors and Microsystems*, vol. 32, no. 8, pp. 437–446, 2008.

[15] D. N. Godard, "Passband timing recovery in an all-digital modem receiver," *IEEE Transactions on Communications*, vol. 26, no. 5, pp. 517–523, 1978.

[16] M. Simon, "Noncoherent symbol synchronization techniques," in *Proceedings of the Military Communications Conference (MILCOM '05)*, vol. 5, pp. 3321–3327, IEEE, October 2005.

[17] C. Dick, F. Harris, and M. Rice, "FPGA implementation of carrier synchronization for QAM receivers," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 57–71, 2004.

[18] C. Dick and F. Harris, "FPGA QAM demodulator design," in *Proceedings of the 12th International Conference of Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream (FPL '02)*, pp. 279–287, 2002.

[19] D. W. Scott, "On optimal and data-based histograms," *Biometrika*, vol. 66, no. 3, pp. 605–610, 1979.

[20] X. Huo and D. Donoho, "Simple and robust modulation classification method via counting," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '98)*, vol. 6, pp. 3289–3292, May 1998.

[21] K. Umebayashi, J. Lehtomäki, and K. Ruotsalainen, "Analysis of minimum hellinger distance identification for digital phase modulation," in *Proceedings of the IEEE International Conference on Communications (ICC '06)*, pp. 2952–2956, July 2006.

[22] R. F. Pawula, S. O. Rice, and J. H. Roberts, "Distribution of the phase angle between two vectors perturbed by gaussian noise," *IEEE Transactions on Communications*, vol. 30, no. 8, pp. 1828–1841, 1982.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 546217, 12 pages
doi:10.1155/2010/546217

Research Article

Space-Based FPGA Radio Receiver Design, Debug, and Development of a Radiation-Tolerant Computing System

Zachary K. Baker, Mark E. Dunham, Keith Morgan, Michael Pigue, Matthew Stettler, Paul Graham, Eric N. Schmierer, and John Power

Los Alamos National Laboratory, Los Alamos, NM 87545, USA

Correspondence should be addressed to Zachary K. Baker, zbaker@lanl.gov

Received 5 March 2010; Revised 28 July 2010; Accepted 14 September 2010

Academic Editor: Lionel Torres

Los Alamos has recently completed the latest in a series of Reconfigurable Software Radios, which incorporates several key innovations in both hardware design and algorithms. Due to our focus on satellite applications, each design must extract the best size, weight, and power performance possible from the ensemble of Commodity Off-the-Shelf (COTS) parts available at the time of design. A large component of our work lies in determining if a given part will survive in space and how it will fail under various space radiation conditions. Using two Xilinx Virtex 4 FPGAs, we have achieved 1 TeraOps/second signal processing on a 1920 Megabit/second datastream. This processing capability enables very advanced algorithms such as our wideband RF compression scheme to operate at the source, allowing bandwidth-constrained applications to deliver previously unattainable performance. This paper will discuss the design of the payload, making electronics survivable in the radiation of space, and techniques for debug.

1. Introduction and Background

Since the original Adaptive Computing Systems Program at DARPA in 1994, Los Alamos (LANL) has been developing RF processing systems with FPGA devices, now known in the literature as Software Defined Radios (SDR). Originally designed with Altera and Xilinx first generation logic arrays, these receivers have grown with the chip families underlying them [1].

The other components of an SDR have gained in performance as well. In particular, modern ADC technology now enables Direct Down-Conversion analog architectures due to the GHz input bandwidths available in 12 to 16 bit converters. Meanwhile point-of-load power conversion, high speed SRAM, and very large NonVolatile RAM technology all support extreme performance from the processing side. The net result is SDRs which are truly special-purpose supercomputers, and which can execute complex algorithms in real-time, over bandwidths meeting or exceeding those required for modern communications systems.

We focus on satellite-capable SDR designs, since the need for reconfiguration is greatest in these systems which are not accessible after launch. Space deployment requires a robust research effort in COTS Radiation Tolerance (COTS-RT). Identifying promising COTS-RT parts is key to our success, since, for example, the Xilinx Virtex 4 is 1000 times more capable than the best RadHard microprocessors available [2].

In the last decade radiation tolerance has advanced to the point that 400 kRad Total Ionizing Dose (TID) is routinely seen in commercial parts, and destructive latchup is disappearing in many families. These improvements are the result of fabrication process changes intended to improve electrical performance, but the radiation tolerance is a direct result as well. Remaining to be dealt with beyond TID and destructive latchup is Single Event Effects (SEEs) performance which in general is degrading as device feature size shrinks [3–5]. The LANL Rad Tolerance program therefore devotes much of its R&D to methods for mitigating SEE in modern electronics.

No RF receiver is complete without algorithms, which are often point designs to satisfy a specific communications

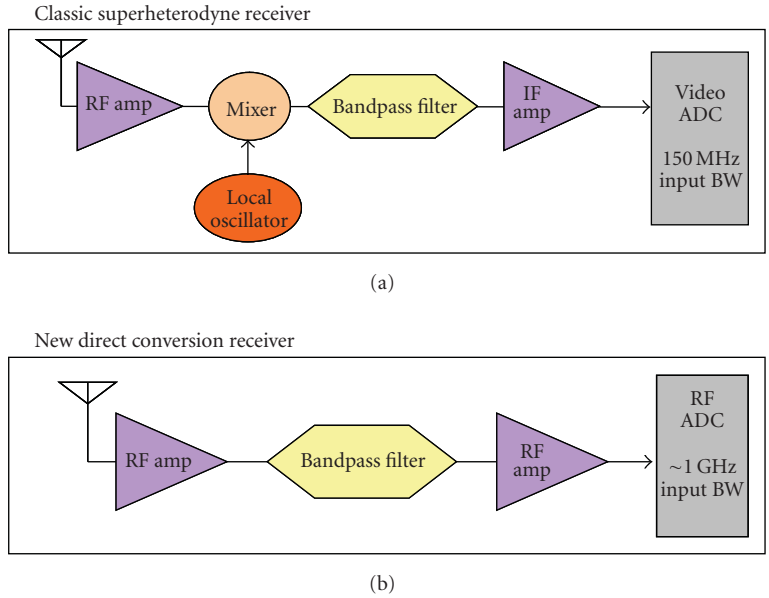


Figure 1: Comparing direct conversion to super-heterodyne architectures. The direct conversion approach does not require an external local oscillator. Rather, the FPGA synthesizes a local oscillator and computationally tunes the desired RF channel.

need. LANL is more interested in general scientific applications that are flexible for a wide range of signal classes. Therefore we describe two such applications in some detail here, an incoherent wideband signal detector, and a fully coherent wideband transform and compression engine. Such algorithms form the building blocks of subsequent user-specific applications, or perform as general purpose science data gatherers. LANL has a well-known VHF lightning research program, based around global monitoring of severe storms via the GPS spacecraft. Radio astronomy also forms another area of interest, especially radio bursts such as pulsars. Our radio is solely used as a receiver.

It is both a challenge and a blessing to the space electronics community that COTS electronics evolves at a rate of at least 2.5 generations/launch. In other words, by the time a new satellite design is delivered to the launchpad, the COTS technology will have advanced by at least 2.5 more generations. This paper describes a delivered receiver that is now two generations behind FPGA state-of-the-art, but we also discuss work with current generation parts in the laboratory.

2. Direct Conversion Principle of Operation

A key innovation of the LANL TeraOps SDR (T-SDR) is the use of direct conversion radio architecture. This feature is diagrammed in Figure 1, where it is seen that higher dynamic range is achieved concurrently with lower power, as long as analog tunability is sacrificed. In direct conversion, tuning is moved into the digital domain, while the analog section becomes a wideband band-selection filter. This simplifies design and manufacture, compared to traditional mixer-IF designs. Most importantly, significant flexibility is acheived by replacing fixed analog components with reconfigurable computation. Our receiver architecture utilizes a 60 MHz digital bandwidth (120 MegaSamples/Second) 16 bit ADC.

As Figure 1 shows, Nyquist's original theory specifies the bandwidth supported by a particular sample rate, but not the actual band, which can exist between any integer multiple of the Nyquist rate which is within the input bandwidth of the analog system (Other terms for this process are used in the literature include Nyquist conversion and aliasing [6]). The ADC does digitally downconvert the input RF signal to baseband, since the output can be interpreted as any of the bands shown in the inset of Figure 1, entitled Nyquist Downconversion (Aliasing). In essence, the digital output stream always occupies the lowest Band 1. The input may be positioned in any of the higher bands, aliasing to Band 1 (a 130 MHz signal would alias through the 120 MHz sample rate to 10 MHz). In the case of the Linear Technology LT2208 ADC, there are at least 18 possible Nyquist bands one can use, albeit with increasing degradation of effective bits.

Our experimental receiver (Figure 2) explores four separate Nyquist bands of the 18 possible, each exploiting a different portion of the ADC performance. Even at the highest bands (>1 GHz), we still obtain greater than 10 effective bits (Section 6). One common theme found while working with our filter suppliers was that we needed 10 MHz transition zones for 50 MHz bandwidth and reasonable size, so we relaxed attenuation at the Nyquist boundaries to 40 dB, as long as we were 80 dB down within another 5 MHz.

3. Detailed Architecture

In order to clearly outline the design aspects enabled by modern chip capabilities, we now describe each system component shown on Figure 3 in some detail.

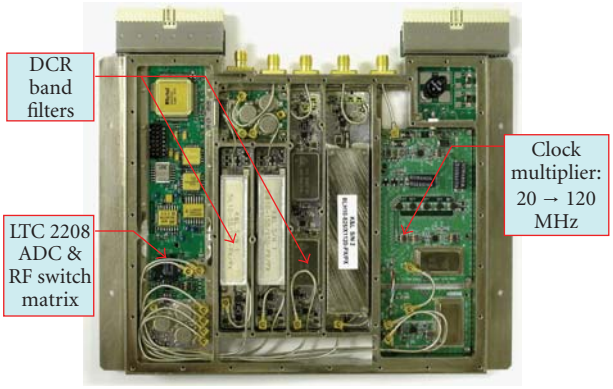


Figure 2: Direct conversion Receiver (DCR) System including analog bandpass filters and analog-to-digital conversion.

3.1. Input Antenna and Direct Conversion Receiver. A 16 bit, 120 Megasample/second ADC converts selected 50 MHz RF bands into Digital Intermediate Frequency (DIF) data streams, downconverting them at the same time it digitizes, in a process known as Direct Conversion. Direct Conversion is a major advancement suitable for HF, VHF, and UHF receivers, and offers gains in both performance and power consumption. The output stream from our Direct Conversion Tuner-Digitizer (DCR) is a 16 bit-wide parallel stream of samples at the 120 MHz clock rate, leading to an aggregate data rate of 1.92 Gigabits/second. Our sample clock remains at one frequency, but there is no reason not to change sample clock rate when input bands change, if desired.

One factor confounding good signal reception at HF and VHF frequencies is that well-matched antennas must subtend one-half wavelength in some dimension. However, at 50 MHz, the wavelength is already 3 meters, and keeps growing inversely with frequency. Hence, T-SDR uses active antennas (Figure 4) which obtain the same pattern as full sized elements, but are less than 1/20 wavelength at the lower end of range. Use of such a short antenna requires nonconventional matching amplifiers to deal with the very high impedances present. But in the HF and low VHF the spectrum is totally dominated by excess ambient noise, for instance at 10 MHz the total noise is ~900,000 K, or 3000 times more than at frequencies higher than 200 MHz [7]. Thus, the system noise figure will be dominated by this 34 dB ambient value, as long as coupling efficiency is adequate and LNA temperature is much lower than ambient.

3.2. The Real-Time Signal Processor (RTSP). The digitized IF output of the DCR is fed to a RTSP, containing two Xilinx Virtex 4 Platform FPGAs and their associated Quad-Data Rate SRAM memories.

The RTSP board and heatsink is shown in Figure 5. Benchmarks of these devices have already shown the ability to process 10 Gigabits/second of data, far higher than our 2 Gigabit/second stream. For a discussion of the partial reconfiguration scrubbing used to mitigate SEE in these devices, see [1, 3].

In the space allocated to the RTSP on cPCI, we have been able to accommodate seven synchronous Quad Data-rate Static RAM (QDR-SRAM) memories, each containing 32 Megabits of data organized 36 bits wide by 1 Megaword deep. ECC through Hamming codes is implemented in the FPGA. Processed data exits the RTSP via a PCI bus interface managed by an ACTEL RTAX Radiation-Hardened PLA. Our requirement is that ~10 Mbytes/second be transferred over the PCI bus. The RTAX is a key part because it is one of the few devices built for use in space, as opposed to COTS hardware repurposed for space. The space-qualified capability is important because the satellite has stringent Do-No-Harm requirements, and the RTAX isolates the satellite from the nonspace-qualified COTS hardware.

A final element in the design of the RTSP are new Point-of-Load power converters from Enpirion, which have been tested by Yale University and LANL to be extremely Rad Tolerant [8]. These are used as the primary supplies for both FPGAs, and for the QDR-SRAM. They operate at >75% efficiency.

3.3. SVIM. Following the RTSP is a Satellite Vehicle Interface & Memory module (SVIM), based on a SPARC 32 bit integer processor. The SVIM card (Figure 6) receives the PCI data, and temporarily stores it in the main 1 Gb memory via DMA. The SPARC is a European Space Agency standard design, and is radiation hardened for LEO and Geosynchronous orbit conditions.

The SVIM contains the Atmel AT697 SPARC controller, spacecraft bus interfaces, and also provides the radiation hardened and nonvolatile storage of all software for the T-SDR payload. The new space processor is rated best-in-class at 100 Mips/Watt, operating at a 80 MHz core clock. Memory supporting the microprocessor is organized as follows.

(i) 2048 Mbit of Start-Up ROM (SUROM) are provided to the SPARC alone, so that critical OS level functions are stored in RadHard memory. All elements of software required to decompress and load FPGA bitstreams from telemetry are contained in this SUROM.

(ii) A bank of blank Xilinx PROM 48 Megabits deep, with hard ECC. These are one-time programmable devices, but programmable on-orbit. We also store the initial applications in this bank of PROM. Compression is used in order to reduce bitstream image size by 4X–16X, which is bitstream dependent. Typical FPGA bitstreams are ~40 Mbits in size.

(iii) A 64 Mb bank of new BAE Chalcogenide-RAM, with hard ECC, which can be reprogrammed at will, including on-orbit. This is the target memory for most reprogramming after launch. It also must contain the SPARC application code associated with the FPGA application.

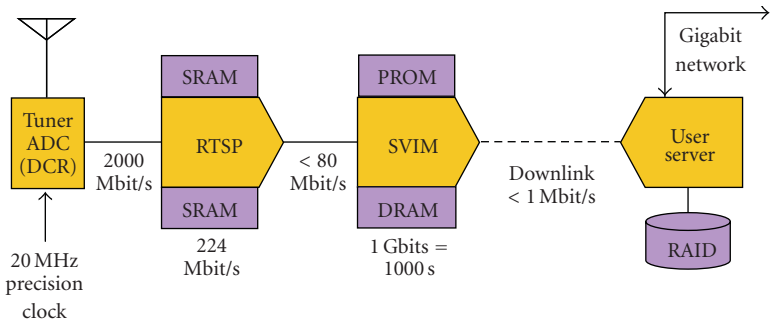


Figure 3: System architecture for satellite receiver system. The ADC accepts a 20 MHz clock and produces a 120 Msps 16 bit data stream. The FPGA-based RTSP processes the RF stream into full-bandwidth snapshots and data products. The radiation-hardened flight computer (SVIM) provides command and control as well as packaging data for transmission to the ground.

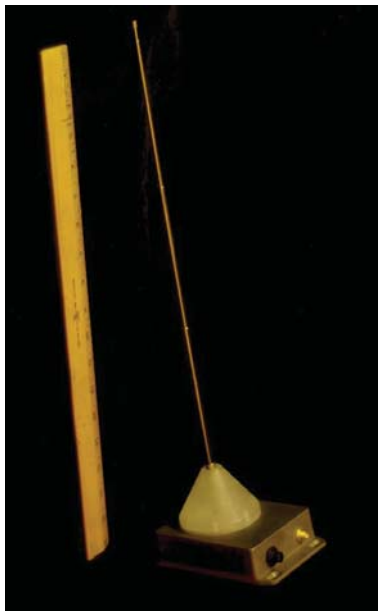


Figure 4: Active antenna assembly with 18″ ruler for size comparison.

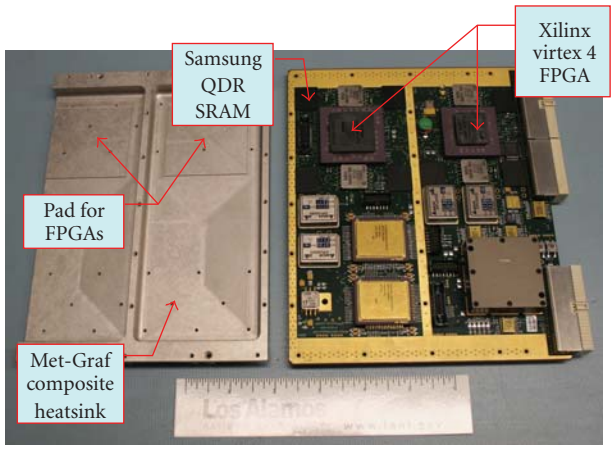


Figure 5: Real-Time Signal Processor (RTSP) board with Xilinx Virtex 4 LX200 and SX55, seven banks of Quad Data-Rate (QDR) memory, and radiation-hardened scrubbing and interface circuitry.

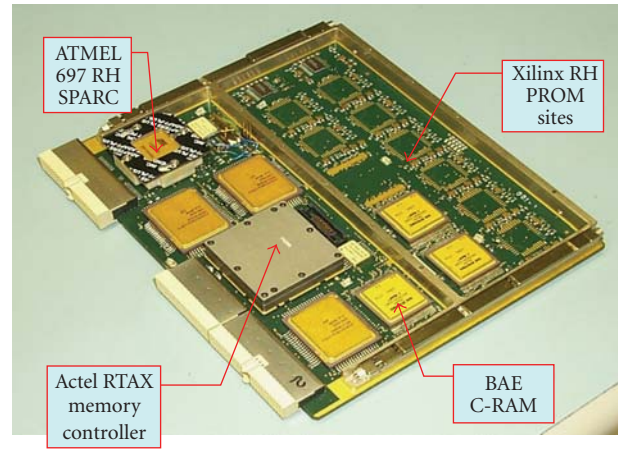


Figure 6: Flight computer board with radiation-hardened processor, SRAM, and Xilinx PROM programming circuits.

The SVIM is tasked with 3 major functions: responding to and transmitting telemetry, retrieving configuration data dynamically from C-RAM memory to support SEU mitigation, and operating the downlink with any associated data compression or thinning operations. The SPARC is capable, but will have to assign interrupt priority to these tasks in normal operations, so user applications reside primarily in the RTSP, with the SVIM supplying custom I/O functions.

4. FPGA Fault Injection Testing

As the FPGAs in our system are only radiation tolerant (and not radiation *hardened*) it is necessary to determine the impact of single-event functional interrupts (SEFIs) and single-event upsets (SEUs) on application availability. An FPGA application's availability is a function of its application error rate and the "down time" caused by application errors.

An application error is very simply an event in which the FPGA application's data output is corrupted. By definition all SEFIs cause application errors. All SEUs, however, do not cause an application error in an FPGA since a given application only uses a subset of the FPGA's available

configuration bits. As a result, the impact of SEFIs is independent of application and can be measured once for a given FPGA device. The impact of SEUs, however, is application dependent and must be measured for each application.

The duration of the data corruption caused by the application error is the "down time". The "down time" caused by a SEFI is system-dependent and can be measured once for a system. The "down time" caused by an SEU-induced application error is application-specific and therefore must be measured.

It is possible to measure an application's error rate with ground-based accelerators, but it is expensive and the radiation destroys hardware. A much less expensive nondestructive method with high fidelity is SEU emulation via fault injection.

4.1. Fault Injection Mechanism. More information about the mechanics of FPGA fault-injection can be found in [9, 10]. By emulating all possible SEUs in the FPGA's configuration memory, it is possible to determine the percentage of SEUs that cause an application error. Also, for each bit in the configuration bitstream, the following important information can be gathered during fault injection regarding an SEU of that bit.

(i) *Criticality*: If an SEU injected into a configuration bit causes an output error, the application error for the bit is classified as "critical" otherwise it is classified as "noncritical" (These are sometimes referred to in the literature as "sensitive" and "nonsensitive.").

(ii) *Recovery Type*: Each critical application error is subclassified as "self-recoverable" or "unrecoverable (These are sometimes referred to in the literature as "persistent" and "nonpersistent" [11])." An application error is "self-recoverable" if the application automatically recovers to an error-free state once the SEU was repaired. An application error is "unrecoverable" if the application requires a reset to return to an error-free state.

(iii) *Recovery Time*: For critical, self-recoverable application errors, the number of clock cycles with output errors is recorded. For critical, unrecoverable application errors, the number of clock cycles with output errors is system dependent.

Once fault-injection has been performed and the preceding information collected it is possible to predict the application's availability. System availability is calculated with (1) which is simply the ratio of "up time" to "total time," where "up time" is the "total time" minus the "down time" as follows:

$$\text{Availability} = \frac{\text{uptime}}{\text{totaltime}}. \tag{1}$$

To use (1) to measure an application's availability it is first necessary to calculate the "down time" caused by each type of application error. The "down time" caused by self-recoverable errors varies with each SEU. It is recorded during fault-injection for each application error as the recovery time, or number of clock cycles with output errors. The expected value of the "down time" due to a self-recoverable error for a given time period is simply the product of the average number of cycles in error, the clock period and the number of self-recoverable errors expected (dependent on the orbit SEU rate) during the time period. Mathematically this can be written as follows:

$$\begin{aligned} &E(\text{downtime}_{\text{self-recoverable}}) \\ &\quad = E(\text{cyclecount}_{\text{self-recoverable}}) \\ &\qquad \times t_{\text{clockperiod}} \times E(\text{events}_{\text{self-recoverable}}). \end{aligned} \tag{2}$$

The "down time" caused by an unrecoverable error varies with the application duty cycle and when the error begins within the duty cycle. The distribution of "down time" caused by unrecoverable errors is assumed to be uniform. As a result, the expected value for the down time for a given period of time due to an unrecoverable error is simply the product of one-half of the duty cycle and the number of unrecoverable errors expected (dependent on the orbit SEU rate) for the given time period. Mathematically, this can be written as follows:

$$E(\text{downtime}_{\text{unrecoverable}}) = \frac{t_{\text{dutycycle}}}{2} \times E(\text{events}_{\text{unrecoverable}}). \tag{3}$$

The "down time" caused by each SEFI is not measured by fault-injection but rather is a measure of the system's average response time for detecting and recovering from a SEFI. In our system, for example, the time to detect and recover from a SEFI was measured at approximately 5 seconds. In other words, the contribution of SEFIs to down time for a given period is the product of the SEFI response time and the number of SEFIs expected for the given time period. Mathematically, this can be written as follows:

$$E(\text{downtime}_{\text{SEFI}}) = t_{\text{SEFI-response}} \times E(\text{events}_{\text{SEFI}}). \tag{4}$$

The overall estimated "down time" is the sum of the "down time" for self-recoverable errors, unrecoverable errors, and SEFIs for a given time period. Once this is known, (1) can be applied as in (5):

$$\text{Availability} = \frac{\text{totaltime} - (E(\text{downtime}_{\text{self-recoverable}}) + E(\text{downtime}_{\text{unrecoverable}}) + E(\text{downtime}_{\text{SEFI}}))}{\text{totaltime}}. \tag{5}$$

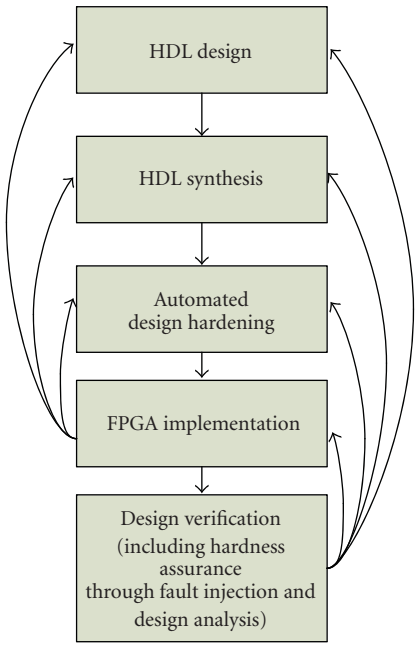


Figure 7: Recommended FPGA design flow for space (only some feedback paths shown).

4.2. Codesign for Robustness. In addition to its utility for developing a dynamic SEU cross-section and availability model for a particular device and application, fault injection provides some additional insight into the robustness of Xilinx Virtex-4 FPGAs and SEU-mitigated designs in space. A few of those beneficial insights will be described in the next several paragraphs.

First, fault injection can provide feedback to designers regarding the actual robustness of the design techniques applied to a particular design, providing an opportunity to gain further insight into design practices for space. For instance, it can provide some feedback on how to effectively apply triple-modular redundancy (TMR) to a FPGA design intended for space. Though the concept is quite simple, the effective execution of applying TMR to a particular design and the resources it uses on a particular FPGA has proven to be quite challenging due to the complexity of the FPGAs themselves and the fact that design software for FPGAs is completely unaware of reliability- and SEU-related issues. In fact, design tools frequently try to optimize designs in ways that actually unintentionally reduce application reliability. Additional understanding about the robustness of TMR and other reliability techniques used will be developed through fault injection.

As an example of this first benefit, for the Radiation Tolerance Test application (described later), we performed fault injection on our final design and noticed several anomalies in its operation. In cases where TMR should have masked errors, application output errors were being observed. By looking at the design and the design build reports, we observed that we were incorrectly using the automated design hardening tool (BLTMR) for handling our QDR memory interfaces, which could not be triplicated. After identifying the problem, we had to return to the automated design hardening stage of the design flow and rerun the BLTMR tool on our design with different mitigation options to resolve the problem. After running the design through the complete design implementation flow, we were able to perform fault injection again and verify that TMR was working properly before burning the final design into PROMs on the SVIM.

Second, fault injection can provide designers with resource-specific vulnerability information. Due to LANLs intimate understanding of Virtex-4 FPGAs and their programming data, we can effectively quantify not only the number of upsets that can affect the operation of a FPGA design but also what types of resources are affected by SEUs, providing insight into the cause of particular errors in an FPGA design. Another key aspect of fault injection is its function in a "best practices" FPGA design flow for space. As illustrated in Figure 7, the LANL approach to FPGA design for space includes a step for automated hardening of the design for space as well as hardness assurance efforts in the design verification step. By developing a fault injection test setup for MRM-LANL, we can provide better design verification tools for designers to verify that their designs meet mission hardness requirements, a key part of the design verification step of the FPGA design flow.

Third, fault injection can provide system developers a validation that their SEU scrubbing software and hardware are working properly. Both in this system and other FPGA-based space systems with which we are familiar, fault injection has been a key test for ensuring that the SEU scrubbing and error reporting subsystems are working properly. In the case of the T-SDR, we were able to use fault injection to identify a data alignment issue in the SEU scrubbing hardware and modify the associated software to compensate for the issue before final delivery of the T-SDR system. We are aware of at least one other FPGA-based system that would have benefited from using fault injection for validating their SEU scrubbing hardware and software. Since actual fault injection on the FPGA hardware was not used during system test, an error in the SEU scrubbing subsystem was not caught until the hardware was on orbit. This was later fixed, but at significant expense relative to if the developers had caught the issue during development.

For the SoftwareDefined Radio application (described later), we applied partial TMR quite judiciously. The clock rate was quite aggressive for the Virtex 4 at 120 MHz. It was clear early on that applying replication and voters would push the maximum clock rate significantly below the target. We reduced the TMR to minimal replication on the LX200. None of the seven QDR interfaces were replicated, as the most trying timing error was in the QDR-SRAM timing. The QDR was inconsistent before TMR was applied. Even though the FPGA nominally met timing, the QDR was plainly mistimed. We utilized the QDR placements from a working nonTMRed build, and applied them as constraints to the TMRed version, resulting in a functioning system.

5. Introduction to In Situ Debug

Visibility in a design is a priceless commodity during debug. Providing that visibility outside the normal channels of a register system and application behavior can be the difference between unexplained failures and mission success. At Los Alamos National Laboratory aggressive launch schedules force engineers to solve problems quickly, thus requiring unique tools. We have recently developed tools that have quickly been found to be invaluable. These JTAG (Joint Test Action Group) tools provide in situ debug, providing an increased amount of debug control and visibility than would otherwise be available. This is possible because we can read and write an arbitrary address space through the JTAG controller. This allows us to set the system to a particular state, set parameters, trigger actions, as well as provide programmed input/output (PIO) to allow for access to a secondary, indirect address space. This is particularly useful for debugging large data-driven applications. For instance, through PIO, we provide access to the entire seven banks of QDR-SRAM in our system from the Linux command line. Because it is meant for interactive debug, the slow JTAG debug connection is not a problem—the system runs more than fast enough for human use.

The JTAG standard is a common standard for running boundary scan tests, running debug tools, and programming memories and FPGAs. Innumerable commercial tools, as well as open-source efforts [12, 13], support the standard. A JTAG connection is one of the several methods to program Xilinx FPGAs [14]. The JTAG state machine is controlled by two pins; TMS (mode select), TCLK (the clock pin, which can be run to near-DC rates). The mode of the state machine determines what happens with the other two JTAG pins, TDI and TDO (data in and data out, resp.,). The TAP state machine is standardized across all platforms, but the registers that are connected to it and that extend into the fabric of the particular chip are determined entirely by the hardware designer. Because Xilinx has provided hooks into FPGA user fabric [14], they have provided the opportunity for a wide variety of powerful tools. For instance, the Picoblaze soft processor can use this approach to load user programs [15] and the Chipscope internal logic analyzer provides cross-hierarchy visibility.

In order to provide in situ access to the JTAG chain, we implement a JTAG host in an embedded microcontroller. In practice, we assume this is a radiation-tolerant processor providing reliable operation, although in our test system we use a Microchip PIC 18f2550 [16]. The controller implements software routines to run the JTAG I/O pins. The software is based on Xilinx Application Note #058 [17], which demonstrates how to implement a SVF file player (Serial Vector Format). This can be used to query ID codes or program the FPGA using SVF files produced through the Xilinx Impact program.

We have extended the functionality to support extended user-mode JTAG commands required to support the debug core. The original source code only plays back previously generated SVF files, whereas our modified version generates TAP controller sequences on the fly. Because the set of extended commands is built within the normal TAP controller architecture, it is a minimal extension to the original source (see Figure 8).

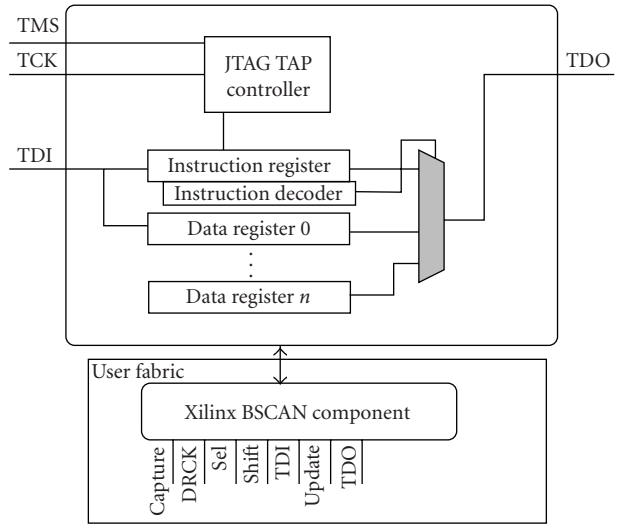


Figure 8: Xilinx internal boundary scan state machine for Virtex 4 [14].

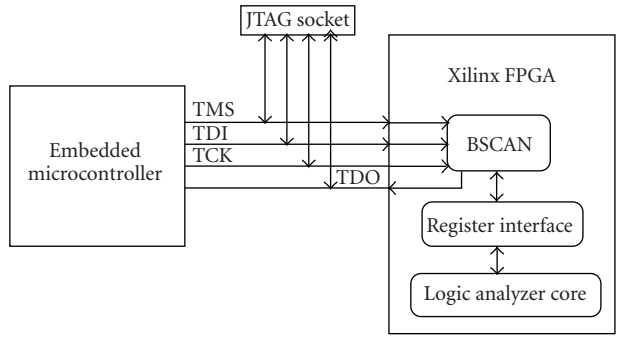


Figure 9: High-level FPGA connection to JTAG controller.

5.1. Implementation Details. The internal debug core resides in user logic. Thus, the FPGA must be programmed for the internal debug system to function, however, IDCODE, FPGA programming, and boundary scan can be accomplished through the JTAG controller without a loaded configuration bitstream. The register debug core is based on the Xilinx BSCAN primitive, which gives user logic access to the boundary scan. The initial code implementing this functionality was based on the S3 Gnat application note [18], but extended significantly to include read/write register access as well as the more elaborate triggering and snapshot system required for the internal digital logic analyzer. The Gnat application note includes firmware examples that we found to be somewhat unreliable on our system. This code was extended to improve its reliability using error correction among other techniques.

In all recent Xilinx Spartan and Virtex devices, there are at least two BSCAN components. We use the second component in the chain, such that the first is available for other debug tools. For instance, Xilinx Chipscope [19] uses

the first component, thus, we can simultaneously have both Chipscope and our tools in the same bitstream (While both components can be instantiated, only one host controller can own the JTAG chain at any time. Thus, the debug cores cannot be operated simultaneously.). It is sensible to instantiate both cores in a system to cover difficult debugging chores, as Chipscope is excellent for logic analyzer chores and our tools are excellent for user-interactive or script-driven data movement.

Our debug core is based on a simple triggering system, which includes a trigger pattern, a trigger mask, and an arm bit. The snapshot system is based on a Block-RAM along with a double-registered input.

5.2. Integrating Microcontroller with FPGA Device. In Figure 9, the connections of the JTAG controller to the JTAG pins of the FPGA slave is illustrated. TMS and TDI have internal pull-ups in Xilinx FPGAs [14]. However, TCLK does not and must be driven high. Thus, in either the open-collector or high-Z option is used, TCLK must be arranged so that it can be cut off to allow an external controller to drive the chain.

The logic analyzer core is based on the register functionality handled directly from the BSCAN primitive. This can be used for generalized register reads and writes in addition to the logic analyzer. We allocated a block of register addresses for replicating access to the control system normally handled through the rad-hard flight computer. The logic analyzer is configured, read, and controlled via the register interface, but its connection to user logic is purely through VHDL port assignment. This is less flexible that Xilinx's postsynthesis black box insertion approach, but it is sufficiently to demonstrate the capabilities of our custom JTAG interface.

This is highly useful in a debug scenario as it can be used independently of the normal communication control channel. This allows in situ system monitoring while normal activities proceed.

5.3. Case Studies. We have already have some success using the tools for development before launch. In particular, the simple register interface has proven to be an irreplaceable tool in solving difficult debug problems. Because the JTAG register system is a simple extension to the normal register command interface (only a few dozen slices are required), it can be included in all builds. We will present three case studies wherein the tools have proven useful.

Case 1. In our first debug challenge, we were experiencing intermittent register write failures. It was rare enough to disregard any obvious problem, and, at 30 MHz and easily meeting the clock constraints it was fairly unlikely the problem was in timing. We have seen some interesting problems where the language construct used to address the array of std_logic vectors has resulted in measureably different behavior in certain conditions (Specifically, when a *for loop* is used to produce a variable input to *conv_std_logic_vector*, the output behaves randomly in some situations. This block of legacy code was still using the nonstandard ieee.std_logic_unsigned library in lieu of the recommended ieee.numeric_std library.), and thus were not certain where the host or slave gate array was to blame.

Using the tools, we were able to connect to the system via JTAG, then write a set of test registers and read them back. This was working fine, so we moved to reading and writing in combination with the failing normal register system. This allows us to determine that the problem lay solely in the mechanical connection between the two chips. The problem was solved through the replacement of the grid array interposer. This is a pad matching the footprint of the part, with a small "spring" for each pin to ensure a stable connection over thermal cycling. They have a limited number of installation cycles before they begin to fail.

Case 2. In our second problematic debug challenge, we struggled to explain the behavior of an application that worked fine when the FPGA was programmed via JTAG but certain application functions failed when programmed via the select MAP interface. The select MAP programming was successful, but it was clear something was being corrupted in the programming sequence. At this point, PROMs for both the FPGA bitfiles as well as the software PROM had been burned, toward an early deadline. This was important because it forced us to try to minimize the changes to either PROM. The situation was made even more intriguing by the fact that any attempt to dump state for debugging by using a new software load would cause the failure to disappear. Thus, normal softwarecontrolled register reads for debug were useless.

However, by connecting to the system through the JTAG interface, we were able to dump the register space and determine that one of the registers was corrupted between the start of FPGA programming and the application being armed. We then forced an application break and dumped the memory location that was the source of the register write. This memory was corrupted as well. Eventually, we determined that the decompression routine was corrupting a single byte outside of its allocated space, which eventually was written to the FPGA. Because the JTAG interface provided access to FPGA application space, we were able to track down a software corruption on the other end of the PCI bus.

Case 3. In another project, an airborne persistent surveillance platform, we wished to develop multiple hardware components in parallel, namely, a high speed serial connection between a microprocessor and the FPGA, and the QDR-SRAM interfaces. Without the microprocessor connection, we had no way to communicate with the FPGA to test the SRAM interfaces. By connecting to the system via JTAG, we were able to move test images in and out of the SRAMs without use of the eventual flight communication path. While the JTAG connection is much slower, it allowed us to make progress where it otherwise would be impossible.

Because the tools operate in a Linux environment, an operator can connect to the server and operate an FPGA

Figure 10: T-SDR modules loaded in a compact-PCI test chassis. The use of commodity compact-PCI chassis allow for cheap and efficient development cycles. Shown right to left are RTSP, DCR, and SVIM in COTS cPCI test chassis.

board remotely from the command line, without having a direct board interface. This is useful not just in a laboratory or commercial setting, but also in educational settings where communicating with a development board is nontrivial. For instance, common Spartan 3 boards for university classes often have no way to talk to the FPGA except buttons and LED readouts. These tools can allow sufficiently fast communication to open a wide variety of application development opportunities.

The functional prototype system operates through a USB connection. This makes it very similiar to a commercial JTAG cable. However, our initial system is just a demonstration for the final system, which will be implemented in a rad-hard microcontroller and connected via SpaceWire to the satellite ground interface. At that point, the interface to the JTAG controller will be via SpaceWire packets, which will be translated from the satellite's real-time command and telemetry interface.

6. Results and Performance Testing

Here we present a few of the applications we have developed for the T-SDR. Our system has been under test for two years as we prepare for flight delivery, and is seen in finished form in Figure 11. A COTS form factor was also chosen, Compact PCI, to to enhance testability and to allow use of low-cost test resources. Figure 10 shows a compact PCI test chassis housing the three system cards. We can replace the RadHard Controller module with commodity PowerPC control modules, enabling device development without requiring emulation of the satellite interfaces.

When inserted in the 3 liter chassis shown in Figure 11, the entire T-SDR weighs 5.5 kg, and has a maximum power consumption of 53 W. The heat must be dissipated through the case. Even with 60% efficient 25 VDC bulk power converters, this enables the 16 k FFT cores in the third

Figure 11: Completed, sealed T-SDR system with satellite interfaces.

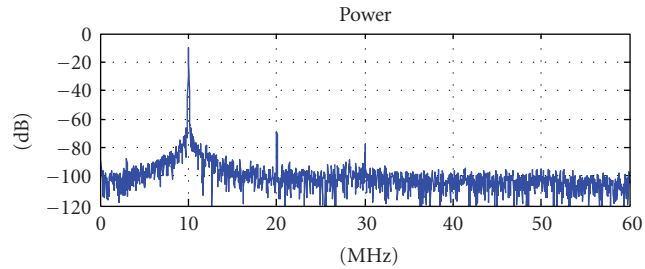


Figure 12: Spectrum output from 130 MHz sine wave input illustrating the low-noise floor of the DCR. Due to the fixed 120 MHz sample rate, the bandpassed signal is aliased to 10 MHz.

application to function at the 120 Msps input rate. The next system architecture (described in Section 7) will reduce the power conversion inefficiency.

The system provides the capability of loading new applications after delivery and launch. This is accomplished through the uploading of new FPGA bitstreams and support software. The system was delivered with two applications, a "parts test" application that will test various individual components (including FPGAs and memory systems), and a SoftwareDefined Radio (SDR) application that will be the main RF data collection system.

6.1. Radiation Tolerance Test Application. As the demands for on-orbit processing increase, the need to insert improved computing technology into space systems also increases. The challenge, of course, is how to carefully adopt and insert new hardware technologies into operational systems without putting the system's missions at risk. One approach is to perform a considerable amount of ground-based testing and analysis. Another more conservative approach leverages ground-based testing but also demonstrates the hardware on orbit in noncritical systems (as experiments or otherwise) to provide flight heritage for the new hardware. Our approach is an example of the latter, combining both ground test data and the data from actually flying hardware to demonstrate the utility of new technology on orbit.

The main goal of the Technology Readiness Level (TRL) verification application is to demonstrate that the new

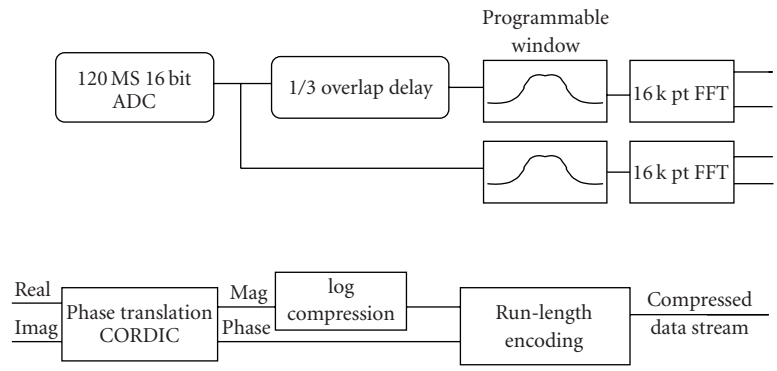


Figure 13: Application architecture for Gabor transform-based wideband compression system.

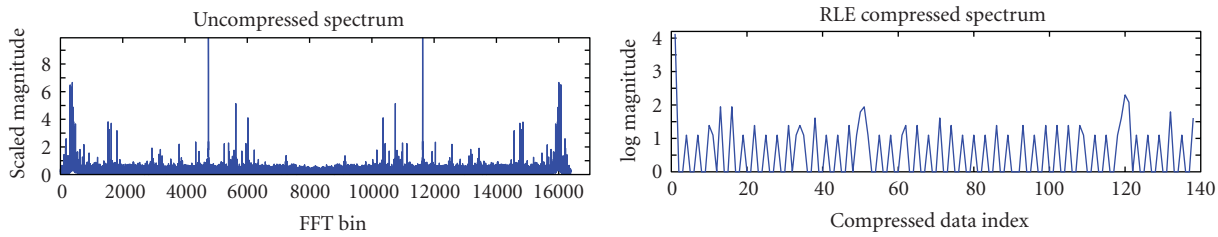


Figure 14: (a) Output from 16 k FFT of data captured from the FORTE satellite. (b) The same data compressed using the Gabor compression scheme with Run-length Encoding.

devices being flown in this system are useful for other government space missions. More specifically, we want to show that the Xilinx Virtex-4 FPGAs, the BAE C-RAM, the Atmel LEON-based microprocessor, QDR-SRAMs, and other components should be considered for future missions. As well, we will demonstrate how they can be used by mitigating risk. Since a significant amount of ground-based radiation testing has gone into most of the new parts used, probably the most challenging issue is to demonstrate that these components can operate reliably as components in a larger system and how to accomplish this. For the Xilinx FPGAs and most of the other new parts, this involves demonstrating that user applications can operate reliably in the presence of SEUs since the Total Ionizing Dose (TID) and single-event latchup characteristics of these devices have been demonstrated through ground-based radiation testing.

Our primary parts test routine is an FFT, with choice of on-board digital sine wave test sequence, or actual antenna input. Qualification test data showing actual dynamic range achieved by the T-SDR with 10 MHz input to the RF port is shown in Figure 12.

6.2. General-Purpose Recorder and Receiver. While the FPGA-based RTSP can implement any functionality, in the satellite module it is well suited to implementing a SoftwareDefined Radio and RF capture functionality. The system provides a variety of modes for capture of radio signals. Radio data can be captured in full-bandwidth snapshots directly off the Analog-to-Digital Converter (ADC), or from several taps, including subband tuned outputs and demodulated outputs.

While performing experiments scientists are often required to sift through large amounts of uninteresting data waiting for a brief event. The data stream is constantly monitored and a snapshot recorder is activated when the trigger conditions are met. For instance, a basic ADC level trigger checks for high RF signal levels at a very coarse level. A more sensitive trigger is implemented using a frequency trigger that detects energy in desired frequency bands. The frequency trigger is very effective as it is able to detect and activate on small signals far below the noise floor.

In addition to these data capture triggers based on simple energy presence, the T-SDR can also trigger based on communications parameters such as turbo decoder fram acquisition or demodulator lock.

6.3. Wideband Gabor Compression Application. For our third application, we looked for ways to pack more raw RF signal into a smaller digital bandwidth. The general term for this problem is data compression and expansion, or "companding." Application 3 relies on a "lossy" companding algorithm, which is suited to the transmission of RF signals as long as certain minimum fidelity requirements are met. LANL patented such a method several years ago, and now is actively exploring the technology [20].

Our scheme is built on the realization that most RF signals are modulated as the mathematical product of the information and an underlying carrier, where the strength of the received signal can vary dramatically. Thus, the great

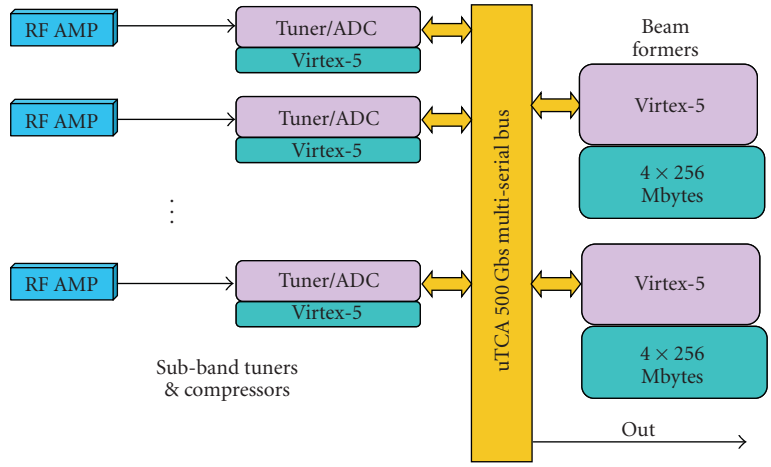


Figure 15: Multiple-Input Multiple-Output (MIMO) architecture for Virtex-5 FPGA-based beamformer.

majority of practical sensor applications are multiplicative modulation processes, which only require constant relative precision. This implies that logarithmic magnitude representations may be used without loss of resolution, as long as they are properly scaled. Few assumptions can be made about phase, requiring more information to be kept. The basic block diagram of Application 3 is illustrated in Figure 13.

Given the above assumptions, our process consists of applying a logarithmic transform to the output of a Fourier transform. The purpose of the frequency transform is to convert convolution processes to multiplication in frequency space, and to separate various signals that overlap in the time series, but which are separate in the frequency space. Use of the log transform then allows the multiplicative signals still contained in each frequency sample to be treated as additive superpositions.

The 120 Mega-Sample/second input data is subjected to a 1/3 overlap delay (1/3 of the FFT size of 16 k samples) to produce two data streams. This allows the output to completely cover the input signal after the windowing function is applied. Without the overlap delay, signals occurring at the edge of the transform might be masked and inadvertently destroyed. The two signals are then fed through a window multiplier to reduce spectral leakage and to allow near-perfect reconstruction. The outputs of the windowing operator are suitably delayed such that both signals can be presented to the FFT simultaneously. This is necessary because the system uses a single FFT to compute the transform of both streams, reducing the area and power requirements for the large 16 k point FFT. Because the inputs to the FFT are real, the output of the FFT is separable into two unique frequency transforms.

The output of the FFT is then fed into a rectangular/polar CORDIC transform. This is a pipelined core that converts the complex FFT output into corresponding polar representation consisting of magnitude and phase, which is the natural domain to either compress or to interpret man-made signals. At this point the magnitude enters a lookup table, providing Mu Law scaling, which is linear at low amplitudes and logarithmic at larger amplitudes, providing a balance between range and precision. At present we do not process the phase, but methods of removing phase trend information and compressing it are very promising, and under study.

Finally the data enters a run length coder, with variable threshold, so that only magnitude values over a chosen level are accepted. For the discarded values below threshold, the coder discards both magnitude and phase, and substitutes a sequence length code instead. In this way, larger or smaller data compaction ratios are achieved, depending on the degree of fidelity required. Figure 14 illustrates the effectiveness of the scheme with data collected on FORTE, another LANL satellite. A sample size of 16 k real points is reduced to 140 points in log-polar representation (with magnitude and phase). In this scheme, low energy frequency bins are discarded and the length of runs of discarded bins are recorded.

7. Summary and Work in Progress

Even as the receiver was developed, new COTS hardware appeared that dramatically changed architectural possibilities again. The two most significant developments for LANL were the following.

(i) The advent of Multi-Gigabit Serial cores in FPGAs, DSPs, and microprocessors. These hard IP blocks enable all-serial modular systems.

(ii) The development of silicon-on-insulator ADC such as the TI5424, which offer very strong Rad Tolerance while providing input bandwidths exceeding a GHz and sample rates > 400 Msps. This enables continuous band coverage from Direct Conversion architectures.

Together, these key evolutions led us to consider multichannel SDR architectures for MIMO and other spatial diversity applications. One such architecture now under active pursuit at LANL is shown in Figure 15, which we call NextRF. A TI ADS5474 ADC was chosen, and operates at 400 Msps, followed by a Xilinx Virtex-5 SX95T processor, creating an SDR with almost seamless coverage from DC to 500 MHz.

Test results are expected from NextRF by 2010. Our simulations indicate that the V5SX95 with the 16 k FFT and CORDIC cores described above can operate at 400 MHz, and occupancy is about 18% of the device. In this partitioning scheme, the compressor runs within the receiver head, and delivers a serial stream of digital data to postprocessors which combine and further reduce the signal size into a final output datastream.

Acknowledgments

The LANL team gratefully acknowledges long-term support from DOE NA-22, DARPA, and other DOD agencies in development of all our SDR work. Only through such long-term support can a new capability be fully developed. The authors also recognize the contributions of our commercial partners, Xilinx, Lockheed-Martin, BAE Systems, Linear Technology, and Texas Instruments. This document is released under LA-UR 10-06455.

References

[1] P. S. Graham, M. Caffrey, M. J. Wirthlin, and D. E. J. N. H. Rollins, "Reconfigurable computing in space: from current technology to reconfigurable systems-on-a-chip," in *Proceedings of the IEEE Aerospace Conference*, pp. 2399–2410, 2003.

[2] Published specifications of the BAE Rad750 PPC (50 Mips/Watt) device, or the Atmel AT697E (100Mips/Watt, versus measured Virtex-4 performance (50 GOps/Watt) at several facilities).

[3] H. Quinn and P. Graham, "Terrestrial-based radiation upsets: a cautionary tale," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 193–202, April 2005.

[4] H. Quinn, P. Graham, and B. Pratt, "An automated approach to estimating hardness assurance issues in triple-modular redundancy circuits in xilinx FPGAs," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3070–3076, 2008.

[5] H. Quinn, P. Graham, J. Krone, M. Caffrey, and S. Rezgui, "Radiation-induced multi-bit upsets in SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2455–2461, 2005.

[6] R. Crochiere and C. Rabiner, *Multi-Rate Digital Signal Processing*, Prentice-Hall, Upper Saddle River, NJ, USA, 1983.

[7] CCIR Reports 341-6, 258-5, & 670-1, International Telecommunication Union, Geneva, Switzerland.

[8] S. Dhawana, O. Bakera, R. Khannad, et al., "Radiation resistant dc-dc power conversion with voltage ratios > 10 capable of operating in high magnetic field for lhc upgrade detectors," in *Proceedings of the Topical Workshop on Electronics for Particle Physics*, 2008.

[9] E. Johnson, M. Caffrey, P. Graham, N. Rollins, and M. Wirthlin, "Accelerator validation of an FPGA SEU simulator," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2147–2157, 2003.

[10] P. S. Ostler, M. P. Caffrey, D. S. Gibelyou,, et al., "SRAM FPGA reliability analysis for harsh radiation environments," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3519–3526, 2009.

[11] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin, "SEU-induced persistent error propagation in FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2438–2445, 2005.

[12] "Open On-Chip Debugger: Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing," 2008, http://openocd.berlios.de/web/.

[13] "URJTAG: Universal JTAG library, server and tools," 2008, http://www.urjtag.org/book/index.html.

[14] "Xilinx UG071 Virtex-4 Configuration Guide," 2006, http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[15] "PicoBlaze User Resources," 2008, http://www.xilinx.com/ipcenter/processor central/picoblaze/picoblaze user resources .htm.

[16] "PIC18F2455/2550/4455/4550 Product Family," 2007, http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=dDocName=en010280.

[17] "Xilinx In-System Programming Using an Embedded Microcontroller," 2007, http://www.xilinx.com/support/documentation/application_notes/xapp058.pdf.

[18] "Using the JTAG Interface as a General-Purpose Communication Port," 2005, http://www.xilinx.com/support/support/documentation/user_guides/ug070.pdf.

[19] "Xilinx Chipscope Pro Tool," 2009, http://www.xilinx.com/ise/optionalprod/cspro.htm.

[20] M. Dunham, "Logarithmic compression methods for spectral data," US Patent no. 6,529,927, 2003.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 357839, 13 pages
doi:10.1155/2010/357839

Research Article

A Decimal Floating-Point Accurate Scalar Product Unit with a Parallel Fixed-Point Multiplier on a Virtex-5 FPGA

Malte Baesler, Sven-Ole Voigt, and Thomas Teufel

Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, 21073 Hamburg, Germany

Correspondence should be addressed to Malte Baesler, malte.baesler@tu-harburg.de

Received 26 February 2010; Revised 1 October 2010; Accepted 20 November 2010

Academic Editor: Viktor K. Prasanna

Decimal Floating Point operations are important for applications that cannot tolerate errors from conversions between binary and decimal formats, for instance, commercial, financial, and insurance applications. In this paper, we present a parallel decimal fixed-point multiplier designed to exploit the features of Virtex-5 FPGAs. Our multiplier is based on BCD recoding schemes, fast partial product generation, and a BCD-4221 carry save adder reduction tree. Pipeline stages can be added to target low latency. Furthermore, we extend the multiplier with an accurate scalar product unit for IEEE 754-2008 *decimal64* data format in order to provide an important operation with least possible rounding error. Compared to a previously published work, in this paper, we improve the architecture of the accurate scalar product unit and migrate to Virtex-5 FPGAs. This decreases the fixed-point multiplier's latency by a factor of two and the accurate scalar product unit's latency even by a factor of five.

1. Introduction

Financial calculations are usually carried out using decimal arithmetic, because the conversion between decimal and binary numbers introduces unacceptable errors that may even violate legal accuracy requirements [1]. Therefore, commercial application often use nonstandardized software to perform decimal floating-point arithmetic. These software implementations are usually 100 to 1000 times slower than equivalent binary floating-point operations in hardware [1]. Because of the increasing importance, specifications for decimal floating-point arithmetic have been added to the recently approved IEEE 754-2008 Standard for Floating-Point Arithmetic [2] that offers a more profound specification than the former Radix-Independent Floating Point Arithmetic IEEE 754-1987 [3]. Therefore, new efficient algorithms have to be investigated, and providing hardware support for decimal arithmetic is becoming more and more a topic of interest. However, most modern microprocessors still lack of support for decimal floating-point arithmetic, because additional hardware is costly. The POWER6 is the first microprocessor with implementing the IEEE 754-2008 decimal floating-point format fully in hardware [4, 5], while the earlier released Z9 architecture already supports decimal floating-point operations but implements them mainly in millicode [6]. Nevertheless, the POWER6 decimal floating-point unit is as small as possible and optimized to low cost. Thus, its performance is low. It reuses registers from the binary floating-point unit, and the computing unit mainly consists of a wide decimal adder. Other floating-point operations such as multiplication and division are based on this adder, that is, they are performed sequentially.

Due to the increasing integration density of CMOS devices, Field-programmable Gate Arrays (FPGAs) have recently become attractive for complex computing tasks, rapid prototyping, and testing algorithms. Furthermore, today's FPGA vendors integrate additional dedicated hard-wired logic, such as embedded multipliers, DSP slices, large amount of on-chip RAM, and fast serial transceiver modules. Thus, using FPGA platforms as coprocessors is an interesting alternative to traditional and expensive VLSI designs.

Besides the four basic arithmetic floating-point operations, that is, addition +, subtraction −, multiplication ×, and division /, a fifth arithmetical operation was introduced in the IEEE 754-2008 standard, that is called fused multiply-accumulate (MAC). This operation can assist to improve the accuracy of scalar products. Unfortunately, this approach does not go far enough as consecutively applied

MAC operations, for example, a scalar product, can still lead to totally wrong results because of cancellation. The reason is rounding of intermediate results. For example, the summation of $a_1 = 10^{30}, a_2 = -10^{30}, a_3 = 10$, and $a_4 = -20$, each with 16 digits precision, can lead to four different results, depending on the order of execution

$$\begin{aligned} ((a_1 + a_2) + a_3) + a_4 &\longrightarrow -10, \\ ((a_1 + a_3) + a_2) + a_4 &\longrightarrow -20, \\ ((a_1 + a_4) + a_2) + a_3 &\longrightarrow 10, \\ ((a_1 + a_3) + a_4) + a_2 &\longrightarrow 0. \end{aligned} \tag{1}$$

Scalar products are calculated in many applications, in which cancellation may cause serious problems or numerical overhead slows down algorithms. This includes linear system solving, least squares problems, and eigenvalue problems [7]. In order to overcome these problems, we consider another operation, the so-called accurate scalar product or accurate MAC [8] which is calculated in two steps. First, the products are computed exactly and are added to a long fixed-point register without loss of accuracy. Then, to obtain a floating-point number, the result is rounded only once. This approach guarantees an optimal scalar product with least significant bit accuracy. It can be shown that by providing the accurate scalar product all operations of computer arithmetics can be performed with maximum accuracy, too [9].

Specifications for decimal arithmetic have been added to IEEE 754-2008 mainly for financial applications. Generally, these applications only use a limited range of floating-point numbers such that cancellation errors are not an issue, and an accurate scalar product unit seems to be no gain for decimal arithmetic. Nevertheless, the accurate scalar product unit proposed in this work might be useful because scalar product calculations and accumulations are common operations in financial mathematics, for instance, in portfolio valuation and optimization. Thus, even if cancellation is not an issue, the accurate scalar product unit speeds up these operations because one multiplication and accumulation are computed in the pipeline in one cycle without interlocks, and the high accuracy is gained at no extra cost.

As specified by IEEE 754-2008 [2], the computation of the elementary floating-point operations $+, -, \times$, and / is performed by the computation of the exact (infinitely precise) result followed by a rounding to the destination format. We extend this accuracy requirement to the accurate scalar product operation. Let us denote $R = R(b, p, q_{\min}, q_{\max})$ a floating-point system, where b is the radix, p is the significand's precision, and $q_{\min}$ and $q_{\max}$ are the exponent's range. Moreover, $\mathrm{fl}(x) : \mathbb{R} \rightarrow R$ is a rounding operation that induces floating-point addition $\oplus$ and multiplication $\otimes$ such that

$$a \oplus b := \mathrm{fl}(a + b), \quad a \otimes b := \mathrm{fl}(a \times b), \quad \forall a, b \in R. \tag{2}$$

Then the exact scalar product s can be expressed by

$$\begin{aligned} s &:= \sum_{i=1}^{n} a_i \times b_i = a_1 \times b_1 + \cdots + a_n \times b_n, \\ &\forall a_i, b_i \in R(b, p, q_{\min}, q_{\max}),\ i = 1 \cdots n, \end{aligned} \tag{3a}$$

and the accurate floating-point scalar product $\bar{s}$ by

$$\bar{s} := \mathrm{fl}\left(\sum_{i=1}^{n} a_i \times b_i\right) = \mathrm{fl}(s). \tag{3b}$$

For comparison, the traditional floating-point scalar product $\tilde{s}$ is computed by software, rounding each intermediate result. It can be expressed by

$$\tilde{s} := (a_1 \otimes b_1) \oplus \cdots \oplus (a_n \otimes b_n). \tag{3c}$$

The novelty of the decimal fixed-point multiplier presented here is its parallel and pipelined FPGA nature that is faster than other comparable FPGA implementations and is even time competitive with binary multipliers implemented in FPGAs. The concept of accurate scalar product is not new, but hardware support for binary MAC is seldom and even more rare for the decimal accurate MAC. However, [9] presents a decimal accurate scalar product, but most of the components are serial and have long latencies. Contrary to this, in the new FPGA-based design presented here, we use a fast parallel decimal multiplier and a parallel accurate scalar product unit that can be pipelined to improve latency. This paper summarizes and extends the research published in [10]; in particular, it gives a more detailed introduction and description of the proposed architecture. Furthermore, we improved the speed of the decimal fixed-point multiplier by a factor of two and the accurate scalar product unit by a factor of five, respectively. The outline is given as follows. Section 2 begins with an overview of decimal fixed-point multiplication followed by the description of our proposed parallel decimal multiplier. Section 3 shortly introduces accurate scalar product and presents our proposed architecture. In Section 4, the accurate MAC unit is extended by the concept of working spaces which allow a quasiparallel use of the accurate scalar product unit. Post-place & route results are presented in detail in Section 5, and finally in Section 6, the main contributions of this paper are summarized. Additionally two proofs about complement calculation and simplification of the summation of sign extensions are given in the appendix.

2. Decimal Fixed-Point Multiplier

The Decimal Fixed-Point Multiplier is the basic component of the accurate MAC unit. It computes the product $A \cdot B$ of the unsigned decimal multiplicand A and multiplier B, both natural numbers with the same precision p.

Decimal multiplication is more complex than binary multiplication due to the inefficiency of the digit representation on binary logic. It requires to handle carries

across decimal and binary boundaries and introduces digit correction stages. Furthermore, the number of multiplicand multiples that have to be computed is higher because each digit ranges from 0 to 9. To reduce this complexity, several different approaches have been proposed that are described in the following. All of them have in common that the multiplication is performed in two steps: the generation of partial products and their accumulation. However, they differ in the optimization of these steps.

For the calculation of the partial products, there are two approaches proposed. The first method generates and stores the required multiplicand multiples $\{A\times1,\ldots,A\times9\}$ a priori which are then distributed to the reduction stage through multiplexers controlled by the multipliers digits. Since this approach requires the generation of eight multiples and some of them, for example, $A \times 3$, require a time-consuming carry propagation, Erle et al. [11, 12] proposed a reduced set of multiples $\{A \times 1, A \times 2, A \times 4, A \times 5\}$. All remaining multiplicand multiples can be generated by adding only two from the set. Lang and Nannarelli [13] describe a parallel design that recodes the multiplier's digit set $\{0,\ldots,9\}$ into the digit sets $\{0, 5, 10\}$ and $\{-2,\ldots,+2\}$ exploiting that the multiples $A\times2$ and $A\times5$ can be calculated very fast due to the absence of carry propagation. Vazquez et al. [14, 15] present three different multiplier recoding schemes. The Signed-Digit Radix-10 Recoding transforms the digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,5\}$. The drawback is the need of a carry propagate adder for the calculation of the multiple $A \times 3$. The two others recoding schemes are Signed-Digit Radix-4 Recoding and Signed-Digit Radix-5 Recoding using the transformation sets $\{0, 4, 8\}$, $\{-2,\ldots,+2\}$ and $\{0, 5, 10\}$, $\{-2,\ldots,+2\}$. Both do not need a slow carry propagate adder for partial product generation but require a more complex partial product reduction.

The second method generates the partial products only as needed using digit-by-digit multipliers with overlapped partial products. To reduce the many combinations, in [16] is proposed a digit recoding of both operands from $\{0,\ldots,9\}$ to $\{-5,\ldots,+5\}$. In [17] is described a direct implementation for BCD digit multipliers. It implements a binary digit multiplication followed by a binary product to BCD conversion. Compared to this, in [18] the digit-by-digit multiplier is implemented by means of the FPGA's memory; however, no digit recoding is applied.

The accumulation of the partial products consists of two stages: the fast reduction (addition) to a two-operand and a final carry propagate addition. Similar to binary multiplication, the accumulation of the partial products can be performed sequentially, in parallel, or by a semiparallel approach. A sequential multiplier iteratively adds up each partial product to an accumulated sum. In [19], the accumulation is performed sequentially by decimal (3 : 2) carry save adders and a final carry propagate adder which leads to a short critical path delay and low area usage but longer latency. It performs a multiplication in $p+4$ cycles. In parallel multipliers, the area consumption is much higher, but the latency can be reduced and the architecture can be pipelined to achieve a higher throughput. In [13], a fully parallel multiplier with digit recoding (see above) is presented. The accumulation is performed by a tree of carry save adders and a final carry propagate adder. Vazquez et al. [14] present a new family of parallel decimal multipliers. The carry-save addition in the reduction stage uses new redundant decimal BCD-4221 and BCD-5211 digit encodings. In [20] is introduced a new method of partial product generation and together with the reduction scheme of [13] and the carry propagate addition method of [14] this design is believed to be the fastest design in literature but sacrifices area for high speed. Despite the partial product reduction scheme presented in [20] is the fastest for ASIC designs, the reduction scheme presented in [14] is more appropriate to FPGA designs. The reason is that [20] is based on BCD full adders which introduce a delay of two lookup tables per reduction stage, whereas the reduction scheme presented in [14] can be implemented with a delay of only one lookup table per reduction stage.

Contrary to the several works on implementations in ASICs, decimal multipliers are not often implemented in FPGAs. These few are [10, 18, 21]. The method in [21] exploits the FPGA's internal binary adders and uses decimal to binary conversion and vice versa. This approach is only feasible for small multipliers. The decimal multiplication in [18] is sequentially and is based on digit-by-digit multipliers that are implemented by memory (BRAM or distributed RAM). It also describes a combinational multiplier design which is only applicable for small precisions p. In a recent work [10], we proposed a fully combinational decimal fixed-point multiplier optimized for Xilinx Virtex-II Proarchitectures [22]. It is based on fast partial product generation and a combinational fast carry save adder tree. It can be pipelined to achieve a high throughput which is a crucial feature for the usage in an accurate scalar product unit. In this work, we adapted the design for Xilinx Virtex-5 devices [23], and in doing so we could double speed and throughput.

2.1. Proposed Parallel Decimal Multiplier. The proposed Decimal Fixed-Point Multiplier computes the product $A \cdot B$ of the unsigned decimal multiplicand A and multiplier B. It is fully combinational and can be pipelined. In particular, it is based on BCD recoding schemes, fast partial product generation, and a BCD-4221 carry save adder (CSA) reduction tree, which is based on [15]. It is optimized for use on Xilinx Virtex-5 FPGAs. A decimal natural number Z is called BCD-$\beta_3\beta_2\beta_1\beta_0$ coded when Z can be expressed by

$$Z = \sum_{m=0}^{p-1} Z_m \cdot 10^m,$$
$$Z_m = \sum_{n=0}^{3} Z_{mn} \cdot \beta_n, \quad Z_{mn} \in \{0, 1\}. \tag{3}$$

Time-critical components are BCD-8421 carry propagate adders (CPAs) that are used in partial product generation to calculate the multiplicand's triple fold $A \times 3$ and for final addition. The adders are proposed in [24] and are designed and placed on slice level, considering a minimum carry chain length and least possible propagation delays. Figure 1 shows

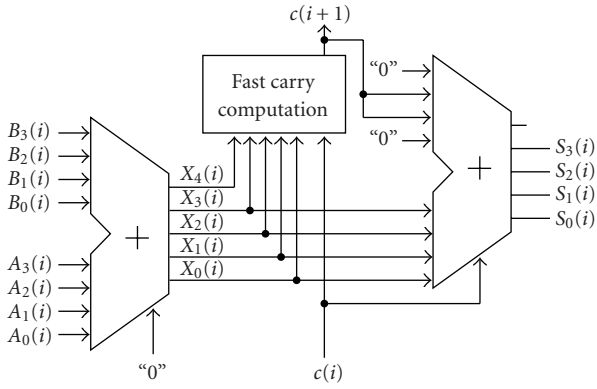


Figure 1: BCD-8421 full adder.

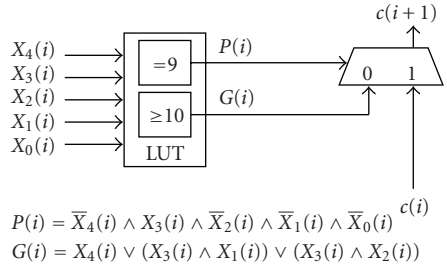


Figure 2: Fast carry computation.

an elementary BCD-8421 full adder. It consists of an adding and a correction stage using two binary 4-bit adder and a fast carry computation unit that is depicted in Figure 2. It exploits the FPGA's internal fast carry chains to minimize latency. The fast carry computation unit implements two functions on the intermediate result of the first stage, *propagate* $P(i) = P(y(i))$ and *generate* $G(i) = G(y(i))$ with $y(i) := 16x_4(i) + 8x_3(i) + 4x_2(i) + 2x_1(i) + x_0(i)$,

$$P(i) = P\big(y(i)\big) = \begin{cases} 1 & \text{if } y(i) = 9, \\ 0 & \text{otherwise,} \end{cases}$$
$$G(i) = G\big(y(i)\big) = \begin{cases} 1 & \text{if } y(i) \geq 10, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

and the carry signal $c(i+1)$ yields to

$$c(i+1) = \begin{cases} c(i) & \text{if } P(i) = 1, \\ G_i & \text{otherwise.} \end{cases} \tag{5}$$

Altogether the adder consumes 9 lookup tables (LUTs) per digit. In particular, the fast carry-bypass logic (carry computation unit) spans only over one LUT.

Generally, the fixed-point multiplier consists of six functional blocks as depicted in Figure 3. The basic idea is to generate $p + 1$ partial products and to sum them up which is performed by the parallel carry save adder tree (CSAT) and the final BCD-8421 carry propagate adder (CPA). The CSAT is based on (3 : 2) CSA blocks for BCD-4221 format.

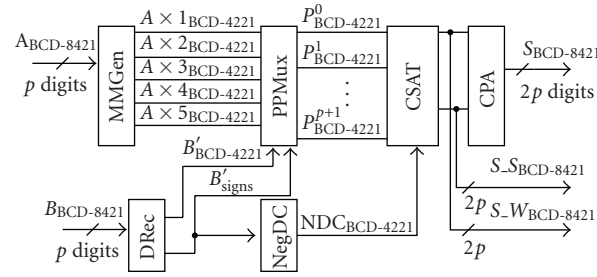


Figure 3: Parallel fixed-point multiplier.

The partial products are the multiplicand's multiples and are selected via the partial product multiplexer (PPMux). Due to the multiplier recoding that transforms the multiplier's digit set $\{0, \ldots, 9\}$ into the signed digit set $\{-5, \ldots, 5\}$ [15], and a simple method to handle negative partial sums (10's complement), only five multiples ($A \times 1$, $A \times 2$, $A \times 3$, $A \times 4$, $A \times 5$) have to be generated by the multiplicand multiples generator (MMGen) a priori. It can be easily proven that the 10's complement can be calculated by inverting each bit of all digits and adding one (see the appendix) . The functionality of the negative digits correction (NegDC) block is explained in the following.

The MMGen is similar to the generator of multiplicand multiples for SD radix10 encoding in [15], but the decimal quaternary tree is replaced by the BCD-8421 CPA. It exploits the correlation between shift operation and constant value multiplication. For example, a BCD-5421 coded decimal number left shifted by one bit is equivalent to a multiplication by 2, and the result is being BCD-8421 coded. Similarly, a BCD-5211 coded number left shifted by one results in a multiplication by two with a BCD-4221 coded result. And finally, a BCD-8421 coded decimal number shifted by three results in a multiplication by 5, and the result is of type BCD-5421.

$$\begin{aligned} (X)_{\text{BCD}-5421} \ll 1 &\equiv (X \cdot 2)_{\text{BCD}-8421}, \\ (X)_{\text{BCD}-5211} \ll 1 &\equiv (X \cdot 2)_{\text{BCD}-4221}, \\ (X)_{\text{BCD}-8421} \ll 3 &\equiv (X \cdot 5)_{\text{BCD}-5421}. \end{aligned} \tag{6}$$

A recoding operation is very fast and consumes two (6 : 2) LUTs per digit, whereas a constant shift operation costs nothing because it is just a renaming of signals. Hence, with exception of $A \times 3$, all multiples can be easily generated by simple shift operations and digit recodings. For the $A \times 3$ multiple, an additional CPA is inevitable which unfortunately limits the maximum working frequency and thus emphasizes the need of pipelining. Alternatively, the multiples could be composed of two operands and be added in the following CSAT, as proposed in [12]. This would speedup the MMGen but would also double the inputs to the CSA and increase significantly its complexity and resource consumption. Figure 4 depicts the functionality of the MMGen. It is similar to the generator of multiplicand multiples presented in [14], but we replaced the decimal quaternary tree by our BCD-8421 adder.

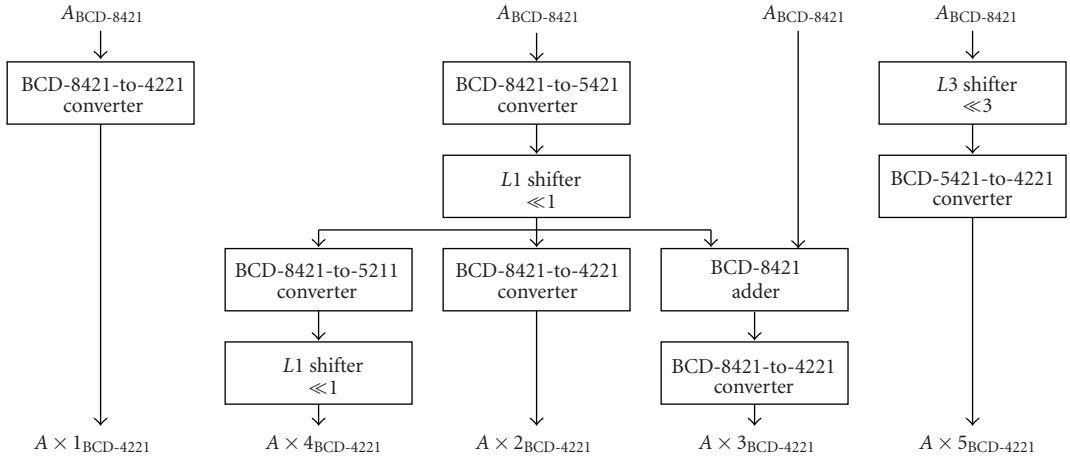


Figure 4: Multiplicand multiples generator.

The decimal recoding unit (DRec) depicted in Figure 3 reduces the number of multiplicand multiples that have to be computed by the MMGen, as proposed by Vazquez et al. [15]. In the first step, it transforms each multiplier's digit B_k from the digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,4\}$ and an output carry bit c_k which coincides with the sign signal sign_k

$$\left(B'_k, c_k\right) = \begin{cases} (B_k, 0) & \text{for } B_k \in 0,\ldots,4, \\ (B_k - 10, 1) & \text{for } B_k \in 5,\ldots,9. \end{cases} \tag{7}$$

In the second step, the carry signal from the previous digit is added to the intermediate result

$$B''_k = B'_k + c_{k-1}. \tag{8}$$

This recoding increases the number of partial products by one $(p+1)$ but gets along without any ripple carry, hence it is a very fast operation.

Since the multiplier's output is of length $2p$ but one single partial product is of length p, for 10's complement generation each partial product has to be extended and if necessary padded with 9. To keep the input length of CSAT short, the negative digits correction unit (NegDC) combines the paddings of all partial products in a single word and passes it to the CSAT. This is feasible because adding several words, composed of leading nines and following zeros, always yields to a decimal word composed of only 0, 8, and 9 (see the appendix) . For example,

$$\begin{array}{ll} & 999999990000 \\ + & 999900000000 \\ + & 990000000000 \\ \hline = & x989899990000. \end{array} \tag{9}$$

Moreover, as shown in Figure 5 the position of the nines and eights can be calculated very fast by means of the FPGA's fast carry chain considering the following equations:

$$\text{NDC}(i) = \begin{cases} 9 & \text{for } c(i) = 0 \text{ and } \text{sign}\, B(i) = 1, \\ 8 & \text{for } c(i) = 1 \text{ and } \text{sign}\, B(i) = 1, \\ 0 & \text{else,} \end{cases} \tag{10}$$

$$c(i+1) = \begin{cases} 1 & \text{for } \text{sign}\, B(i) = 1, \\ c(i) & \text{else.} \end{cases}$$

The reduction of the partial products is based on BCD-4221 $(3:2)$ CSAs [15] that reduce three BCD-4221 digits to a sum and a carry digit, both of BCD-4221 coding scheme. In a first version, CSA1, the carry save adder is implemented as proposed by Vazquez et al. [15]. It consists of a 4-bit binary $(3:2)$ CSA and a BCD-4221 to BCD-5211 digit recoder. By means of an implicit shift operation of the BCD-5211 coded carry digit, we obtain a multiplication by two. The block diagram of CSA1 is shown in Figure 6. It consumes overall six LUTs per digit. The drawback of this architecture is that the computation of the sum digit s_i has a latency of one LUT, whereas the computation of the carry digit w_i has a latency of two LUTs. To reduce the computation latency of w_i, we propose a new type of carry save adder, CSA2. It consists of a 2-bit binary $(3:2)$ CSA and a carry digit computation unit. The block diagram of CSA2 is shown in Figure 7. The 2-bit binary $(3:2)$ CSA sums up the two least significant bits of the three input digits and generates the sum digit. The carry digit is computed from the remaining six most significant bits of the three input digits which requires four $(6:2)$ LUTs. The CSA2 method also consumes six LUT per digit but has a lower latency than CSA1.

The $(n:2)$ CSA tree is composed of parallel and consecutively wired $(3:2)$ CSAs. It reduces n decimal words to two BCD-4221 coded decimal words. The $n = p+2$ decimal words are composed of $p+1$ partial products and one summand that regards the sign paddings, as described previously. The CSAT is organized in stages, each reduces p_i

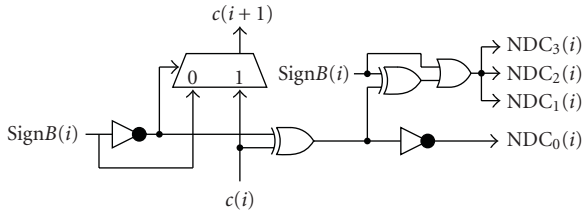


Figure 5: NegDC unit.

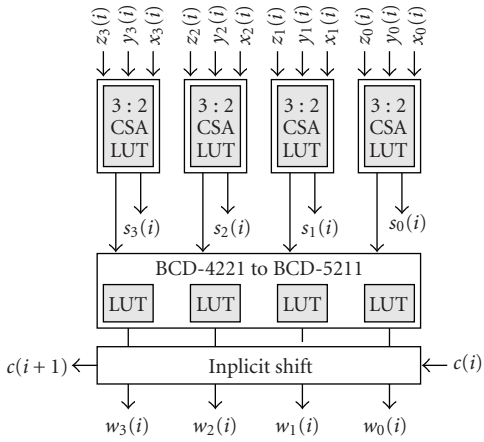


Figure 6: (3 : 2) CSA type1 implementation.

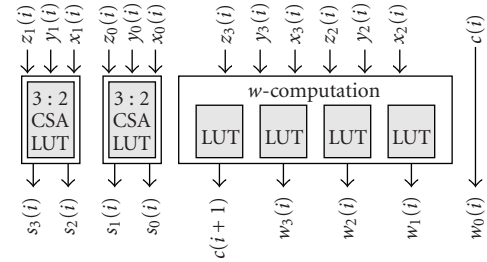


Figure 7: (3 : 2) CSA type2 implementation.

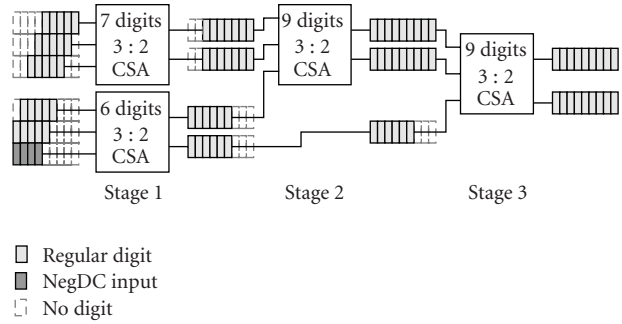


Figure 8: Example: CSA Tree for 6 input words.

words to $p_{i+1} = p_i \cdot 2/3$ words. As in general the ranges of the input words differ, the word length increases with each stage as depicted exemplary in Figure 8.

The redundant carry-save format of the CSAT can be further reduced by a carry propagate adder of length $2p$ to obtain a unique result. However, this CPA can be omitted because the accurate scalar product unit processes on the carry-save format directly.

The maximum frequency of the fixed-point multiplier is limited on the one hand by time-critical components like the CPA and on the other hand by the FPGA's routing overhead. While the maximum propagation delay of the time-critical components can be determined in advance, the routing delay depends highly on the overall project's size. Hence, several pipeline registers can be optionally implemented by means of VHDL generic switches. For a 16×16 digits multiplier, this is one possible pipelining stage to buffer the input words, three for the MMGen, one for PPMux, six for the CSAT (one for each reduction stage), and two for the final BCD-8421 conversion and CPA. Altogether, these are 11 possible pipeline registers for the BCD-4221 carry-save format output and 13 stages for the final BCD-8421 carry-propagation format output. It should be noted that the last CSA stage can be combined with the final BCD-8421 converter, as it is proposed by [15]. However, since the following accurate scalar product unit accumulates redundant BCD-4221 numbers, this improvement could not be applied.

3. Accurate Scalar Product

The accurate scalar product is important for applications in which cancellation may cause problems or numerical overhead slows down algorithms. It is calculated in two steps. First, the products are computed exactly and are summed up to a long fixed-point register without loss of accuracy. Then the result is rounded only once to obtain a floating-point number. Hardware support for the accurate binary scalar product is rare; the accurate decimal scalar product is even less supported by hardware. In [25] is presented a coprocessor with an accurate binary scalar product using the concept of the long accumulator. Reference [9] presents a decimal floating-point arithmetic with hardware as well as software support. It implements the concept of accurate scalar product, but due to the given hardware restrictions most of the components are serial and have long latencies. Contrary to this, in the new FPGA-based design presented here, we use a fast parallel multiplier and parallel shift registers. We accelerate the scalar product's accumulation by use of carry save adders and get rid of overflow and carry signals by the concept of carry caches. Our design is pipelined and requires generally five cycles to multiply and accumulate with an operating frequency of more than 100 MHz.

3.1. Proposed Accurate Scalar Product Unit. The fundamental concept of the long accumulator (LA) is to provide a fixed-point register that covers the entire floating-point range of products, as well as adder, that accumulates these products without loss of accuracy, see Figure 9. When computing the scalar product (3a) n, individual results coming from the decimal fixed-point multiplier are shifted and added

to a section of the LA. The respective section depends on the operands' exponents and is calculated by the address generator (AGen). In order to avoid time-consuming carry propagation, the central adder (CAdd) is implemented as carry-save adders which implies a doubling of the LA's memory to store both operands. Contrary to [9], positive as well as negative operands are accumulated in the same LA by using 10's complement data format. To prevent time-consuming ripple-carry propagations due to sign swapping and overflow, we use a so-called carry cache (CC) that buffers any overflow signals. Contrary to a previously published paper [10], in this work, we have simplified the carry handling by removing the principle of fast carry resolution in case of a carry cache overflow. Instead, we have increased the block size of the long accumulator for carry cache (LACC) to 16 digits, assuming that the CC will never overflow. Actually, in the worst case scenario, it would take the CC over three years to overflow at a reasonable working frequency of 100 MHz. Applying this simplification, we could increase the operating frequency significantly. Before the final accurate scalar product can be output and stored on a temporary result stack (ResSt), the two carry-save operands of the long accumulator for operands (LAOPs) and the entries of the LACC must be summed up and reduced by a final carry propagate adder (FCPA). Therefore, the entire long accumulator would have to be traversed which is a highly inefficient step, since due to locality most applications normally use a minor percentage of the LA and the remaining entries equal zero. To solve this problem, we introduced a so-called touched blocks register (TBR). During MAC operation, the TBR marks the corresponding blocks of the LA as *touched*, which means they are most likely unequal to zero. During final result calculation, only these blocks, that have previously been marked as touched, are actually addressed and read out.

The required length l in digits of the long accumulator can be calculated from the significand's length p and the minimum exponent's value $q_{\min}$ and maximum exponent's value $q_{\max}$, respectively. In order to consider possible overflows, k more guarding digits are provided on the left. For our design, the number of guarding digits $k = 18$ is chosen. Considering a maximum working frequency of 100 MHz, it would take the LA over 300 years to overflow. Hence, 18 guarding digits are a reasonable choice. Since a multiplication doubles the significand's length and the exponent's range, the LA must hold a total number of digits as follows:

$$l = k + 2 \cdot (q_{\max} - q_{\min}) + 2 \cdot p. \tag{11}$$

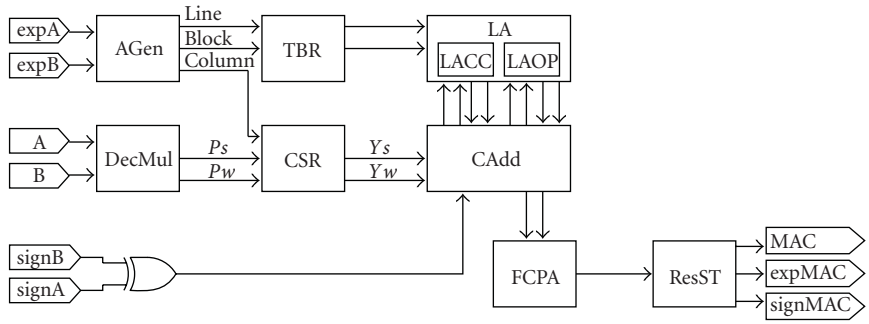
We implemented the MAC unit for IEEE 754-2008 *decimal64* interchange format with $p = 16$ digits precision. With $k = 18$, $q_{\max} = 369$, and $q_{\min} = -398$, the accumulator length results in $l = 1584$. The interchange format *decimal32* with 7 digits precision is downward compatible and thus can be applied to the decimal MAC unit, too.

The LA is implemented by use of local Block SelectRAM (BRAM). It is organized in $\lceil l/p \rceil$ segments, each covers $p = 16$ digits. Since the shifted multiplier's result always fits into $3p$ digits, three arbitrary consecutive segments can be addressed, yielding a word of $3p$ digits. Therefore, the LA is organized in three blocks with $l/(3 \cdot 16) = 33$ lines. It provides memory for both the long accumulator for operands (LAOP) as well as for the long accumulator for carry cache (LACC). To each LAOP block, an LACC block is assigned that handles any overflow signals during accumulation. This prevents pipeline interrupts and allows the storage of negative numbers in 10's complement data format. One LA line comprises of LACC and LAOP each with three blocks composed of 16 digits with 4 bits. As the central adder is of $(4:2)$ carry-save type with length $3p$, two carry-save operands and two carry cache entries must be stored separately. The advantage of this approach is its high speed because of the absence of a ripple carry signal. The drawback is twice as much memory consumption. Since BRAM is a dual-ported memory, the two carry-save operands can be accessed simultaneously through different ports. Thus, $n = ((4 \cdot 16 \cdot 3) \cdot 2) \cdot 2 = 768$ bits must be addressed in parallel which requires 12 parallel dual-ported BRAMs with 32 bit data ports. Each BRAM has a memory depth of 1024, but both operands only need a depth of $2 \cdot (l/(3 \cdot p)) = 66$. The remaining memory can be used for the implementation of the so-called working spaces (WSs) which are introduced below. The LA runs at double data rate, because within one cycle the operands and carry cache entries from the LA have to be fetched and added to the multiplier's output and then in the same cycle the result has to be written back to the LA.

When a block address is not a multiple of three, then the operand spans over two memory lines, that is, the least significant digits (LSDs) are not located in the first block but in the second or third. The block alignment is performed by the shift register which is therefore implemented as a cyclic shift register, see Figure 10. Alternatively, the block alignment could have been implemented between LA and CAdd, but this approach would have increased the longest path and would have reduced the overall operating frequency.

The drawback of the memory organization in lines comprising three segments is a complicated address generation, that is, the need of a division by three. An alternative solution with four blocks per line leads to an easier address calculation but also requires larger multiplexers for operand shift operations. Fortunately, the complicated division by three can be accomplished by applying an embedded binary multiplier, as described in the following.

The address generator (AGen) shown in Figure 9 transforms the input exponents into three addresses (column, block, and line address) to access the LA and to control the shift register. The line and block addresses define a segment $s = \text{line} \cdot 3 + \text{block}$, and the column address locates the position inside this segment. Thus, each digit in the LA can be characterized by its exponent E that relates to the three addresses as follows: $E = \text{line} \cdot 48 + \text{block} \cdot 16 + \text{column}$. The central adder can only sum up block-aligned operands. For that reason, the multiplier's result has to be shifted cyclically. The shift left amount (SLA) arises from the column and



AGen: Address generator
TBR: Touched blocks register
DecMul: Decimal multiplier
CSR: Cyclic shift register
LA: Long accumulator
LACC: LA for carry cache
LAO: LA for operands
CAdd: Central adder
FCPA: Final carry propagation adder
ResSt: Reset stack

Figure 9: Accurate scalar product unit.

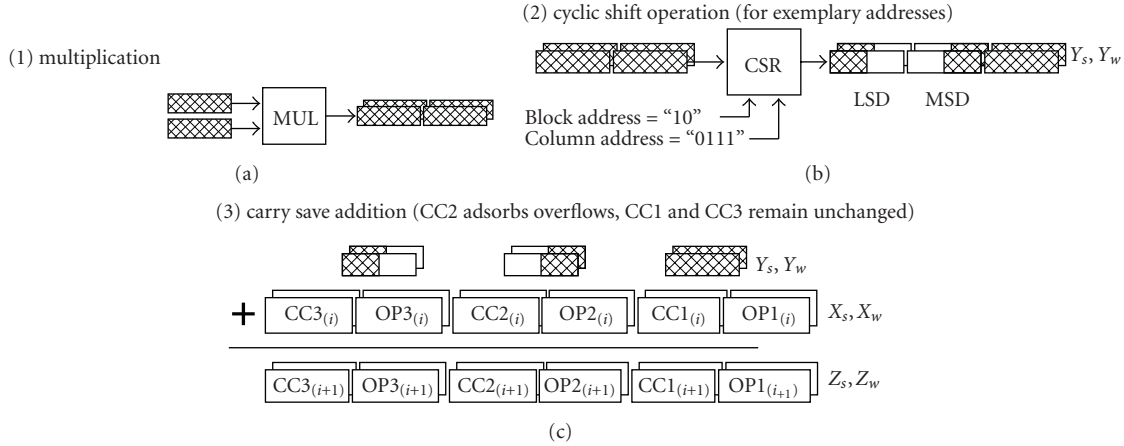


Figure 10: Block alignment of LA and CSR.

block addresses, whereas the block and line addresses are used to address the LA,

$$\text{SLA} = \text{block\_addr} \cdot 16 + \text{column\_addr}. \tag{12}$$

Unfortunately, the memory partitioning applies a division by $3 \cdot 16 = 48$ to determine the line address. That division is accomplished by inverse multiplication considering the maximum digit's exponent of $E_{\max} = 1550$. This approach requires besides logical, shift, and add operations one additional binary fixed-value multiplication which can be performed by the dedicated multiplier of the FPGA's DSP48E slices, see Algorithm 1 .

Once the result has been computed from the decimal multiplier it enters the shift register before it is accumulated by the central adder and is stored in the LA, as already described above. The shift register extends the decimal multiplier's outputs from $2p$ to $3p$ length and shifts the operands according to the column address. Because the decimal multiplier internally uses digit recoding combined with 10's complement representation, there might arise a carry signal (whenever at least one multiplier's digit is greater than or equal to five) which is discarded by the subsequent CPA but is still present as a *hidden carry* in the output of carry-save format. In such cases, the most significant digits

$E = \exp(\text{A}) + \exp(\text{B})$
$\text{line} = \lfloor E/(3*16) \rfloor = (E*1366) \gg_2 16$
$\text{res} = E - \text{line}*48 = E - (\text{line} \ll_2 5 + \text{line} \ll_2 4)$
$\text{block} = \text{res} \gg_2 4$
$\text{column} = \text{res} \,\&\, 0\text{b}1111$

Algorithm 1: Address generation.

Table 1: Shift register post-place & poute results.

	T [ns]	freq_{max} [MHz]	# LUTs
mul-based, 48 bit	5.072	197	193
mul-based, 2×48 digits	5.177	193	1201
mux-based, 48 bit	2.565	389	960
mux-based, 2×48 digits	2.993	334	7512

(MSDs) of the extended $3p$ word must be padded with 9's, and the overflow has to be cleared by a subtraction of 1 in the carry cache adder, see Algorithm 2 . However, the main challenge is the vast shift depth up to 47 digits along with a large number of operands to be shifted, that is two operands each with four bits per digit. These are $2 \cdot 4 = 8$ 48-bit cyclic shift register. Since serial shift register with low resource consumption cannot be pipelined, only parallel solutions are applicable. Two solutions for parallel cyclic shift registers are analyzed, the first one is a shift register using multiplexers and the second one applies the hard-wired multiplier of the DSP48E slices. The latter one is possible because an Lk-shift operation complies with a multiplication by 2^k. Virtex-5 devices support the design of large multiplexers by using the dedicated F7AMUX, F7BMUX, and F8MUX multiplexers. Hence, four LUTs can be combined into a $16:1$ multiplexer.

A 48-bit shift register can be implemented by three 16-bit shift register stages wired consecutively. These shift registers are composed of 16-bit multiplexers or multipliers. Each stage can be pipelined to obtain a low latency as shown in Figure 11. Table 1 summarizes the maximum delay and the number of LUTs used for both cyclic shift register solutions. The multiplexer-based solution is faster but requires much more LUTs, up to 6.25 times more. Since the longest path in the accurate scalar product unit is bounded by the central adder (approximately 10 ns), the multiplier-based cyclic shift register is preferred because of its far less resource usage.

The central adder is a $(4:2)$ CSA to keep latency low. The four inputs are two cyclically shifted words from the decimal multiplier and two operands from the long accumulator. The central adder is composed of two sequentially arranged $(3:2)$ CSA stages. Furthermore, negative numbers are applied by their 10's complement that requires an additional correction of +1. Since the multiplier's output is of redundant carry-save type, two correction factors of +1 are needed. For this purpose, the carry inputs of the $(3:2)$ CSA stages are used. Each CSA stage also produces a carry signal that has to be absorbed by the Carry Cache described below. One $(3:2)$ CSA stage comprises of three 16-digit $(3:2)$ CSAs that are interconnected depending on the block address, see Figure 12.

To handle overflow during accumulation without interfering the pipeline and to allow the storage of negative numbers in 10's complement format without carry propagation, we introduced the CC. It temporarily adds and stores carry and sign signals. The CC uses the carry-save format, too. To each LAs operand block is assigned a CC block which consists of 16 digits and adsorbs the two carry signals of the LA (cout1, cout2) and the two negative sign signals due to 10s complement (sign). Because of its size, the CC blocks are not supposed to overflow. Finally, the CCAs neutralize the hidden carry signal, too, that is weighted negative in case of positive numbers but positive in case of negative numbers. Summarizing all factors yields to the pseudocode depicted in Algorithm 2.

The final result is computed by successively reading out the LA, starting with the least significant digit (LSD) and reducing the CC's entries as well as the LA's operands by means of the CAdd. Finally, this redundant result is summed up by the final carry propagate adder and stored on the result stack (ResSt). Hence, the FCPA produces a series of positive and negative floating-point numbers with the precision of $3 \cdot 16 = 48$ digits and ascending block aligned exponents. The carry out signal of the FCPA is fed back to the FCPA's carry input. The ResSt is composed of a dual ported memory. On the one port, the result of the FCPA is written into the memory, whereby zero entries are omitted. On the other port, the result is accessible for external components with either greatest or smallest number first, depending on requirements of the further data processing. For example, when a final rounding is required to fit the result into IEEE 754-2008 data format, then it is advantageous to read out the greatest number first.

As application is usually subject to locality only a small percentage of the LA is filled with nonzero entries. Thus, it would be very inefficient to traverse the complete LA during final readout. Due to performance issues, we introduced the so-called touched blocks register (TBR). Each time the MAC unit accesses a block in the LA, an according flag in the TBR is set to indicate highly probable nonzero data. Only these previously *touched* blocks in the LA are regarded to compute the final result. In order to reduce the complexity for final result computation, four consecutive blocks are marked as *touched* instead of three as might be expected. This method simplifies the final result computation because possible overflows are already considered and no further exceptions must be regarded.

The parallel fixed-point multiplier as well as the accurate scalar product unit are designed to support pipelining. As already described, the fixed-point multiplier with redundant carry-save output has 11 configurable pipeline registers that can be switched on and off by VHDL generic switches. The accurate scalar product unit further adds three stages for the cyclic shift register and also three stages for the final carry propagate adder. Especially the latter ones are important to reduce the longest asynchronous path and to achieve high operating frequencies.

4. Working Spaces

The introduction of so-called working spaces (WS) allows the quasiparallel use of the MAC unit, that is, there can be several users concurrently accessing the MAC unit without interfering each other. The users can be different processors or different processes on one processor. There can even be a single process that handles more than one accurate scalar product unit, for example, to compute complex scalar products, interval scalar products, and so forth. Working

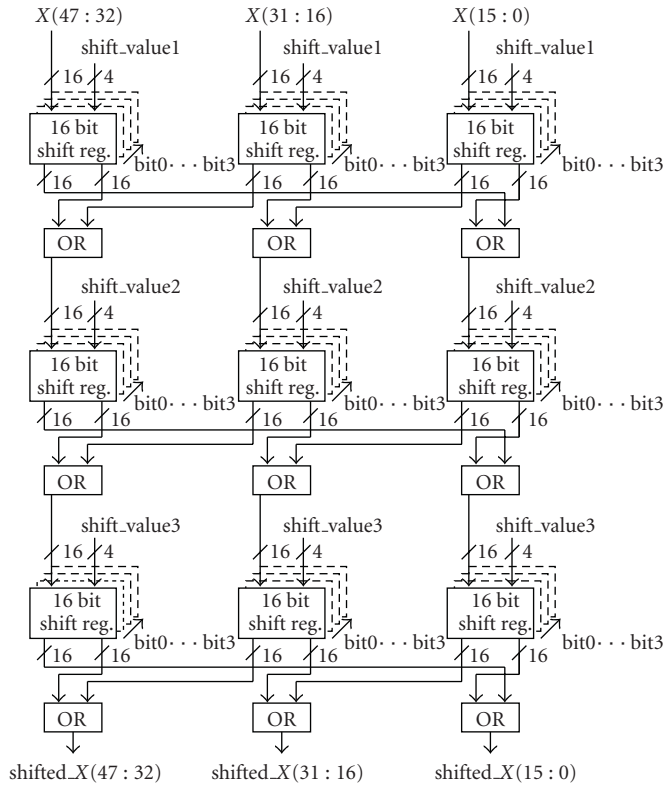


Figure 11: 48-digit cyclic shift register.

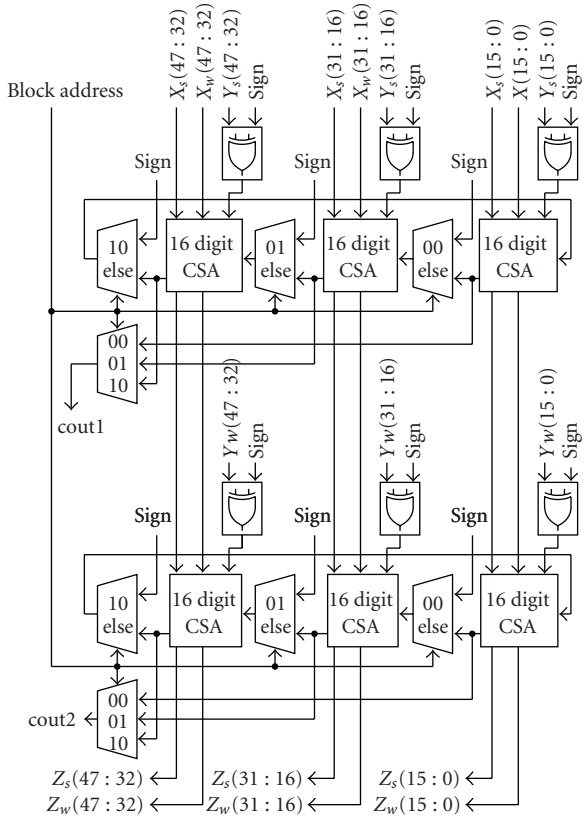


Figure 12: Central (4 : 2) CSA for operands.

```
CC[line + block] += cout1 + cout2
                    −2 · sign
                    −hidden_carry · (−1)^sign
```

Algorithm 2: Pseudocode for CCA calculation.

spaces are realized by duplications of all memory elements together with some additional multiplexers. These are the long accumulator with operand storage and carry cache, the touched blocks register, and the reset stack. The assignment and access to the working spaces has to be managed by a central control unit, for example, an operating system. The number of working spaces can be set by VHDL generics, too. Actually, it is only limited by available resources.

5. Synthesis Results

All circuits are modeled using VHDL. For synthesis and implementation Xilinx ISE 10.1 [26] has been used. The fixed-point multiplier and the accurate scalar product unit have been implemented for Xilinx Virtex-5 speed grade -2 devices. Firstly, only the fixed-point multiplier with unique carry propagate output has been implemented for several pipeline configurations, see Table 2 and Figure 13.

The results show that the minimum overall latency of about 18 ns can be achieved with no pipeline registers, and

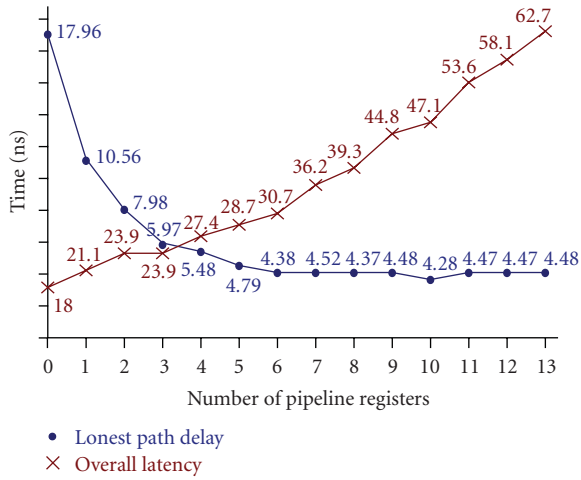


Figure 13: Latency and delay (Post-Place & Route Results) for decimal multiplier, $p = 16$ digits,

Table 2: Post-place & route results for decimal fixed-point multiplier with CPA output.

No. of pipeline stages	longest path delay [ns]	max. freq. [MHz]	No. of LUT [ns]	latency
0	17.96	57	6557	17.96
1	10.56	95	5916	21.12
2	7.98	125	5385	23.94
3	5.97	167	6074	23.88
4	5.48	183	6237	27.40
5	4.79	209	6402	28.74
6	4.38	229	5576	30.66
7	4.52	221	5574	36.16
8	4.37	229	5576	39.33
9	4.48	224	5610	44.80
10	4.28	234	6139	47.08
11	4.47	224	6283	53.64
12	4.47	224	6318	58.11
13	4.48	224	6520	62.72

the best operating frequency of 234 MHz can be obtained with 10 pipeline registers. However, using 6 or more pipeline registers does not reduce the longest path delay significantly and increases the overall latency instead. The LUT usage varies only slightly for different pipeline configurations. In [18], combinational and sequential memory-based digit-by-digit multipliers are analyzed for Xilinx Virtex-4 platforms. A combinational 16×16 multiplier uses 22,033 LUTs and has an overall latency of 26.9 ns. A sequential 16×16 multiplier uses 1,054 LUTs, 8 BRAMs, and has an overall latency of 110.5 ns. A fair speed comparison with the design proposed in this work is difficult because of different FPGA devices. Nevertheless, the unpipelined design proposed in this work is 50% faster than the combinational multiplier proposed in [18]. The sequential multiplier uses rather few

Table 3: Comparison of decimal fixed-point and binary multiplier results.

Multiplier	$T_{latency}$ [ns]	No. of LUTs	No. of DSP48E
Decimal multiplier 0 pipeline registers	17.964	6557	0
CoreGen LUT-based 0 pipeline registers	11.965	2945	0
CoreGen DSP48E-based 0 pipeline registers	25.712	0	10
Decimal multiplier 6 pipeline registers	4.377	5576	0
CoreGen LUT-based 6 pipeline registers	3.475	2982	0
CoreGen DSP48E-based 6 pipeline registers	4.958	123	10

Table 4: Post-place & route results.

	T[ns]	No. of LUTs	No. of RAMB36	No. of DSP48E
MAC[1], 1–8 WS	9.964	10,032	18	73
MAC[1], 8–16 WS	9.971	11,891	36	73
MAC[1], 16–24 WS	9.976	13,302	54	73

[1] Decimal fixed-point multipliers in accurate MAC use two pipeline registers.

LUTs. But contrary to the combinational multiplier, it has a poor latency and cannot be pipelined. Thus, only the combinational multiplier might be suitable for an accurate scalar product unit. However, it uses a considerable amount of LUTs more than the multiplier proposed in this work.

To compare our design with multiplier designs implemented for the same FPGA chip, we have analyzed a binary 53×53 multiplier on a Virtex-5 provided by the Xilinx Core Generator, see Table 3. Our architecture is faster than the DSP48E-based binary multiplier. On the other hand, our fixed-point multiplier consumes approximately twice as much LUTs as the binary LUT-based multiplier and is slower, but it has to be considered that decimal multiplication is much more complex than binary multiplication.

The accurate MAC unit has been implemented with two pipeline registers for the decimal fixed-point multiplier. Together with the three pipeline registers of the cyclic shift register, this amounts to a 5-cycle latency to calculate and store the product of two operands on the long accumulator. The accurate MAC unit can be clocked with up to 100 MHz. Compared to a previously published paper [10], this is an improvement by a factor of five. In comparison, a software implementation of a single 16 digits floating-point multiplication without any long accumulator on a high-performance processor already uses 233 cycles, on lower performance architectures even more [27].

The resource consumption of the accurate MAC unit depends on the number of implemented working spaces. Table 4 summarizes the resource consumption for different configurations.

6. Conclusion

In this paper, we presented a decimal fixed-point multiplier that maps onto FPGA architectures and can help to implement a fully IEEE 754-2008 compliant coprocessor. We analyzed the performance with respect to the number of pipeline registers. Moreover, we integrated the decimal multiplier into an MAC unit which can compute scalar products without loss of accuracy and thus can prevent numerical cancellation. Using the MAC unit on multitasking machines is supported by the concept of working spaces. Compared to a previously published paper [10], we ported our former architecture that was designed to map on (4 : 1) LUT-based Xilinx Virtex-II Pro devices to up to date (6 : 2) LUT-based Xilinx Virtex-5 devices. Furthermore, we improved the algorithm of the accurate scalar product unit. For the fixed-point multiplier, we could achieve a speedup of two, and for the entire accurate scalar product unit we could even achieve a speedup of five. Even though the migration from Virtex-II to Virtex-5 devices has improved the speed of the accurate scalar product unit, the greater part of the speedup is attributable to the improved algorithm.

Appendix

Proofs

Theorem 1 (B's complement). *Let $B \in \mathbb{N}$ denote a radix, p the precision, $X = \sum_{i=0}^{p-1} x_i \cdot B^i$ a positive integer with digits $x_i = \sum_{k=0}^{n-1} x_{ik} \cdot r_k$, $x_{ik} \in \{0,1\}$, and $r_k \in \mathbb{Z}$.*

If $\sum_{k=0}^{n-1} r_k = B - 1$, then the B's complement of X can be easily computed by

$$-X = -B^p + 1 + \sum_{i=0}^{p-1} \overline{x_i} \cdot B^i, \quad \text{with } \overline{x_i} = \sum_{k=0}^{n-1} \overline{x_{ik}} \cdot r_k, \quad \overline{x_{ik}} = \begin{cases} 0 & \text{when } x_{ik} = 1, \\ 1 & \text{when } x_{ik} = 0. \end{cases} \tag{A.1}$$

Proof. Firstly, we prove the B's complement for one digit

$$\begin{aligned} x_i &= \sum_{k=0}^{n-1} x_{ik} \cdot r_k \in \{0, 1, \ldots, B-1\}, \\ &\Longrightarrow \overline{x_i} = \sum_{k=0}^{n-1} \overline{x_{ik}} \cdot r_k = \sum_{k=0}^{n-1} -(x_{ik} - 1) \cdot r_k, \\ &= \sum_{k=0}^{n-1} r_k - \sum_{k=0}^{n-1} x_{ik} \cdot r_k = B - 1 - x_i \\ &\Longrightarrow -x_i = -B + \overline{x_i} + 1. \end{aligned} \tag{A.2}$$

Then, we calculate the complement $-X$ of a number X

$$\begin{aligned} -X &= \sum_{i=0}^{p-1} (-x_i) \cdot B^i \\ &= \sum_{i=0}^{p-1} (-B + \overline{x_i} + 1) \cdot B^i \\ &= \sum_{i=0}^{p-1} \left(-B^{i+1} + \overline{x_i} \cdot B^i + B^i\right) \\ &= -B^p + 1 + \sum_{i=0}^{p-1} \overline{x_i} \cdot B^i. \end{aligned} \tag{A.3}$$

□

Theorem 2 (Sum of leading nines). *The sum of words composed of leading 9's and following 0's*

$$s = \sum_{i=1}^{p} X_i \quad \text{with } X_i = \sigma_i \cdot \sum_{k=i}^{p} 9 \cdot 10^k, \quad \sigma_i \in \{0,1\} \tag{A.4}$$

is a decimal word for which the $p+1$ least significant digits consist of zero or more leading 9's followed only by 0's, 8's, and 9's, that is,

$$s = c_p \cdot 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i \cdot 10^i, \tag{A.5}$$

with $p \geq j \geq 1$, $y_i \in \{0, 8, 9\}$ and $c_p \in \mathbb{N}$.

Proof. We prove the assumption by complete induction.

(1) If all $\sigma_i = 0$, for all $i = 1, \ldots, p$, then the assumption is true because $s = 0$.

(2) If $\sigma_k = 1 \wedge \sigma_i = 0, i \neq k$, then the assumption is also true because the sum s consists of leading 9's followed by 0's.

(3) Let us assume that the hypothesis is true for all $j-1 \geq k \geq 1$ with $\sigma_k \in \{0,1\}$, that is,

$$s_{j-1} = \sum_{i=1}^{j-1} X_i = c_{j-1} \cdot 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i \cdot 10^i, \tag{A.6}$$

with $y_i \in \{0, 8, 9\}$ and c_{j-1} considers the most significant digits of the sum. Then, for the next inductive step j, we have to consider $\sigma_j = 0$ and $\sigma_j = 1$. The condition $\sigma_j = 0$ leads to $s_j = X_j + s_{j-1} = s_{j-1}$, that is, the assumption is true. For $\sigma_j = 1$ follows

$$\begin{aligned} s_j &= X_j + s_{j-1} \\ &= \sum_{i=j}^{p} 9 \cdot 10^i + c_{j-1} 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i 10^i \\ &= \left(c_{j-1} + 1\right) \cdot 10^{p+1} + \sum_{i=j+1}^{p} 9 \cdot 10^i + 8 \cdot 10^j + \sum_{i=1}^{j-1} y_i 10^i \\ &= c_j 10^{p+1} + \sum_{i=j+1}^{p} 9 \cdot 10^i + \sum_{i=1}^{j} y_i 10^i, \end{aligned} \tag{A.7}$$

which finally proves the assertion. □

References

[1] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proceedings of the 16th IEEE Symposiumon Computer Arithmetic (ARITH-16 '03)*, pp. 104–111, IEEE Computer Society, Washington, DC, USA, 2003.

[2] IEEE, "IEEE 754-2008 Standard for Floating-Point Arithmetic," September 2008.

[3] ANSI/IEEE, "ANSI/IEEE 754-1987 Standard for Radix-Independent Floating-Point Arithmetic," October 1987.

[4] L. Eisen, J. W. Ward, H. W. Tast et al., "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–683, 2007.

[5] E. Schwarz and S. Carlough, "Power6 decimal divide," in *Proceedings of the Application-Specific Systems, Architectures, and Processors (ASAP '07)*, IEEE Computer Society, 2007.

[6] A. Y. Duale, M. H. Decker, H. G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: an implementation and testing perspective," *IBM Journal of Research and Development*, vol. 51, no. 1-2, pp. 217–227, 2007.

[7] X. Li, J. W. Demmel, D. H. Bailey et al., "Design, implementation and testing of extended and mixed precision BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 152–205, 2002.

[8] U. Kulisch, *Advanced Arithmetic for the Digital Computer, Design of Arithmetic Units*, Springer, Secaucus, NJ, USA, 2002.

[9] G. Bohlender and T. Teufel, "BAPSC: a decimal floating point processor for optimal arithmetic," in *Computer Arithmetic: Scientific Computation and Programming Languages*, pp. 31–58, B. G. Teubner, Stuttgart, Germany, 1987.

[10] M. Baesler and T. Teufel, "FPGA implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier," in *Procedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 6–11, IEEE, December 2009.

[11] M. Erle and M. Schulte, "Decimal multiplication via carry-save addition," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 0–348, 2003.

[12] M. A. Erle, B. J. Hickmann, and M. J. Schulte, "Decimal floating-point multiplication," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 902–916, 2009.

[13] T. Lang and A. Nannarelli, "A radix-10 combinational multiplier," in *Proceedings of the 40th Asilomar Conference on Signals, Systems, and Computers (ACSSC '06)*, pp. 313–317, October 2006.

[14] A. Vázquez, E. Antelo, and P. Montuschi, "A new family of high - Performance parallel decimal multipliers," in *Proceedings of the18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, pp. 195–204, June 2007.

[15] A. Vazquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *IEEE Transactions on Computers*, vol. 59, no. 5, Article ID 5313798, pp. 679–693, 2010.

[16] E. M. Schwarz, M. A. Erle, and M. J. Schulte, "Decimal multiplication with efficient partial product generation," in *Proceedings of the17th IEEE Symposium on Computer Arithmetic (ARITH '05)*, pp. 21–28, IEEE Computer Society, Washington, DC, USA, 2005.

[17] G. Jaberipur and A. Kaivani, "Binary-coded decimal digit multipliers," *IET Computers and Digital Techniques*, vol. 1, no. 4, pp. 377–381, 2007.

[18] G. Sutter, E. Todorovich, G. Bioul, M. Vazquez, and J.-P. Deschamps, "FPGA implementations of BCD multipliers," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 36–41, IEEE, December 2009.

[19] M. A. Erle, M. J. Schulte, and B. J. Hickmann, "Decimal floating-point multiplication via carry-save addition," in *Proceedings of the 18th IEEE Symposiumon Computer Arithmetic (ARITH '07)*, pp. 46–55, June 2007.

[20] G. Jaberipur and A. Kaivani, "Improving the speed of parallel decimal multiplication," *IEEE Transactions on Computers*, vol. 58, no. 11, Article ID 5184812, pp. 1539–1552, 2009.

[21] H. C. Neto and M. P. Vestias, "Decimal multiplier on FPGAusing embedded binary multipliers," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 197–202, September 2008.

[22] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs-Complete Data Sheet, November 2007.

[23] Xilinx, "Virtex-5 Family Overview," February 2009.

[24] M. Vazquez, G. Sutter, G. Bioul, and J. P. Deschamps, "Decimal adders/subtractors in FPGA: efficient 6-input LUT implementations," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 42–47, December 2009.

[25] C. Baumhof, *Ein Vektorarithmetik-Koprozessor in VLSATechnikzur Unterstuetzung des Wissenschaftlichen Rechnens*, 1996.

[26] Xilinx Inc, "Xilinx ISE 10.1 Design Suite Software Manuals and Help," 2008, http://www.xilinx.com.

[27] M. Schulte, N. Lindberg, and A. Laxminarain, "Performance evaluation of decimal floating-point arithmetic," in *Proceedings of the 6th IBM Austin Center for Advanced Studies Conference*, 2005.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 357978, 14 pages
doi:10.1155/2010/357978

Research Article

Partial Reconfigurable FIR Filtering System Using Distributed Arithmetic

Daniel Llamocca,[1] Marios Pattichis,[1] and G. Alonzo Vera[2]

[1] *Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM 87131, USA*
[2] *Microelectronics Research and Development Corporation, Albuquerque, NM 87110, USA*

Correspondence should be addressed to Daniel Llamocca, dllamocca@ieee.org

Received 2 March 2010; Revised 8 July 2010; Accepted 20 November 2010

Academic Editor: Viktor K. Prasanna

Dynamic partial reconfiguration (DPR) allows us to adapt hardware resources to meet time-varying requirements in power, resources, or performance. In this paper, we present two new DPR systems that allow for efficient implementations of 1D FIR filters on modern FPGA devices. To minimize the required partial reconfiguration region (PRR), both implementations are based on distributed arithmetic. For a smaller required PRR, the first system only allows changes to the filter coefficient values while keeping the rest of the architecture fixed. The second DPR system allows full FIR-filter reconfiguration while requiring a larger PR region. We investigate the proposed system performance in terms of the dynamic reconfiguration rates. At low reconfiguration rates, the DPR systems can maintain much higher throughputs. We also present an example that demonstrates that the system can maintain a throughput of 10 Mega-samples per second while fully reconfiguring about seventy times per second.

1. Introduction

Dynamically reconfigurable systems offer unique advantages over nondynamic systems. Dynamic adaptation provides us with the ability to adapt hardware resources to match real-time varying requirements. The majority of the 1D FIR filtering literature is dominated by static implementations. Here, we use the term static to refer to both CMOS implementations (e.g., [1–5]) and reconfigurable hardware (nondynamic) (e.g., [6, 7]). Some implementations use the label *reconfigurable* in the sense of having the capability to load different filter coefficients on demand (e.g., [2–5]). In the context of this paper, such implementations are considered static since the underlying hardware is not changed or reconfigured.

For reconfigurable hardware, the most efficient implementations are based on Distributed Arithmetic (DA) [8]. These filters have coefficients fixed or hardwired within the filter's logic. This approach allows fast and efficient implementations while sacrificing some flexibility, since coefficients cannot be changed at run time. Dynamic partial reconfiguration (DPR) can be used in this scenario to provide the flexibility of coefficients' values changes without having to turn off the device and only rewriting a section of the configuration memory. The efficiency of DPR over the full reconfiguration alternative and the savings in terms of power and resources is a function of the relative size of the portion being reconfigured [9].

We consider a DPR approach that allows us to change the filter's structural configuration and/or the number of taps. The proposed approach provides a level of flexibility that cannot be efficiently accomplished with traditional static implementations. In particular, we develop a dynamically reconfigurable DA-based FIR system that uses DPR to adapt the number and value of the coefficients, the filter's symmetry, and output truncation scheme. Two systems are presented that allow the flexibility to change all these filter's characteristics: (i) a system that only allows changes to the coefficients values and (ii) a system that allows changes to the number and value of the coefficients, the symmetry, and the output truncation scheme.

Previous research on dynamically reconfigurable FIR filters has focused on multiply-accumulate-based implementations and coarse reconfiguration. The first system

Table 1: Fir filter implementation savings due to the use of filter blocks.

Implementation	Resource requirements
1 Filter block of size M (LUTs have M inputs)	size $I \times 2^M$ words
M/L filter blocks of size L (LUTs have L inputs)	size $I \times 2^L \times M/L$ words

Table 2: Hardware utilization of Virtex-4 XC4VFX20-11FF672 for coefficient-only reconfiguration.

Module	FF	(%)	Slice	(%)	LUT	%
PRR	0	0%	180	2%	360	2%
Static Region	5303	31%	6130	72%	8698	51%
PRR interface	1313	8%	786	9%	885	5%
Overall	5203	31%	6310	74%	9058	53%

described in this paper is based on dynamically reconfiguring at a coarse level, that is, the entire FIR filter. The second system is based on dynamically reconfiguring at the finest possible level, the LUTs that store the coefficients, with a small dynamic reconfiguration area. We have demonstrated a related, LUT-based approach in a dynamically reconfigurable pixel processor [10]. The paper also explores different ways to execute dynamic partial reconfiguration and elaborates on the impact over reconfiguration time overhead of the different approaches.

This paper provides an extended version of the conference paper presented in [11]. The paper has been extended to provide: (i) extended background information, (ii) more implementation details, (iii) extended methodology, (iv) architectural extensions to allow changes on the filter's internal structure, and (v) new results.

The rest of the paper is organized as follows: Section 2 presents background and related work. Section 3 describes the FIR filter core implementation. Section 4 introduces the dynamically reconfigurable system. Results and conclusions are presented in Sections 5 and 6, respectively.

2. Background and Related Work

Reconfigurable logic has established itself as a popular alternative to implement digital signal processing algorithms [12]. Furthermore, a number of articles have been published on using DPR to implement different signal processing algorithms [9, 11, 13, 14]. In particular, [15–17] report different approaches for taking advantage of DPR in FIR filter implementations. The capability of reconfiguring a filter at run time is of special interest for applications such as wireless communications and software radio.

Hardware realizations of FIR filters can be divided into constant coefficients and multiplier-based implementations [15]. In the latter case, DPR is mainly used to change a filter's overall structure [16, 17], or other filter-wide characteristic. At a higher level, DPR is also used to simply change the level of parallelism of an implementation by changing the number of filter cores in an application's critical path. In all these cases, changes are usually initiated from a desire to implement a new filter, based on power or resources considerations, or simply to obtain new functionality. A change in coefficients does not require reconfiguration for this type of filter implementation. Thus, for these cases, DPR has milder constraints in terms of reconfiguration speed and reconfigurable logic partition.

The case of constant coefficients implementation is considerably more complex, since DPR is used to change inner characteristics of the filters (coefficients are not easily isolated within the filter structure). This requires more complex schemas to segment logic into reconfigurable tiles and more efficient reconfiguration mechanism in order to reduce the amount of time it takes to reconfigurable a filter.

DA filters in Xilinx FPGAs are introduced in [18, 19], where the authors exploit common characteristics between the Xilinx's FPGA architecture and the filter architecture. In [7], the authors present other approaches for flexible FPGA implementations of FIR filters by combining pipelined multipliers and parallel, distributed arithmetic.

In [15], the authors consider different DPR architectures for extending constant-coefficients approaches to implement adaptive filters. This relatively early study already provides insights on the advantages of using run-time partial reconfiguration to modify a filter's behavior at run time. The study used an earlier device (currently unavailable) and explored architectures different than DA, which were a natural fit for such device. Their results in terms of performance cannot be compared to our results due the inherent difference between the reconfigurable devices used.

In [17], the authors describe a self-reconfigurable adaptive FIR filter system composed of up to three multiplier-based filter modules. These modules can be reconfigured at run time by a control manager that uses System ACE to store and fetch the corresponding partial bitstream. This system only allows a full-filter reconfiguration instead of finer reconfiguration schemas such as coefficient-only reconfiguration. In this paper, speed results are not clearly presented. The authors report different reconfiguration overhead times for different filters that apparently occupy the same reconfigurable region in the device. These results are surprising, since reconfiguration time overhead depends mainly on the bitstream size, which depends on the size of the partial reconfigurable area, not on the number of resources used within that area. It is also worth mentioning that reconfiguration speeds reported are slower than speeds reported on other DPR papers [20, 21].

In [16], a similar system is described although in this case, it is not self-reconfigurable and uses an external PC to perform reconfiguration. Reconfiguration times reported are also considerably slower than other reported methods.

In [22], the authors describe a tool-flow to map applications to a self-reconfiguring application. The authors use a 32-tap MAC-based FIR filter as an example. The paper compares the performance of simply reloading coefficients by writing over specific registers and using DPR to reconfigure the whole filter. In this paper, the reconfiguration time

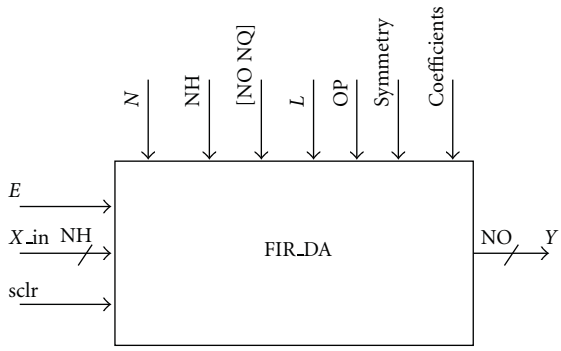


Figure 1: Generalized FIR DA module.

Table 3: Hardware utilization of Virtex-4 XC4VFX20-11FF672 for full-filter reconfiguration.

Module	FF	(%)	Slice	(%)	LUT	%
PRR	1324	8%	818	10%	1306	8%
Static Region	4017	24%	5515	65%	8072	47%
PRR interface	6	0%	5	0%	107	1%
Overall	5341	31%	6333	74%	9378	55%

overhead is large but dismissed as an acceptable handicap for the paper's goals.

In general, the reconfiguration time overhead is an important factor in the evaluation of systems using DPR. Several approaches exist to deal with the overhead. One approach is to hide it by using efficient hardware scheduling strategies (e.g., [23]). A more simplified approach is to select carefully the elements of an architecture that requires reconfiguration for a desired change in functionality (e.g., [11, 21]). By doing so, one can reduce drastically the size of the partial bitstream used to execute the DPR, thus reducing the reconfiguration time overhead. Finally, there is also the approach of maximizing the access speed to the configuration memory (e.g., [20]). Unfortunately, this approach has a limit determined by the device. In the case of Virtex-4 FPGAs, the maximum speed is 3.2 Gbps (32 bit wide bus at 100 MHz). A combination of the last two approaches is used in this paper to deal with reconfiguration time overhead.

Our paper seeks to extend prior research in this area by primarily focusing on developing, analyzing, and improving DPR systems in terms of the dynamic reconfiguration rate on modern devices. This leads us to consider a DA implementation that allows efficient implementations with small hardware footprints on modern FPGA devices. Then, we consider a scalable approach where we have two systems: (i) a DPR system that allows for faster dynamic reconfigurations of coefficient values while fixing the number of taps and (ii) a second DPR system that allows flexibility in the number of taps, the filtering structure, and truncation characteristics while allowing for a slower dynamic reconfiguration rate.

Table 4: PRR measures for both dynamic partial reconfiguration system realizations.

Dynamic realization	PRR size (Slices)	Bitstream size (bytes)
(1) Coefficient-only reconfiguration	$90 \times 6 = 540$	43000
(2) Full-filter reconfiguration	$44 \times 20 = 880$	83000

3. Stand-Alone FIR Filter Core Implementation

A high-performance FIR implementation based on Distributed Arithmetic is described in this section (see also [11]). The approach was coded in VHDL, so as to achieve a level of portability. Specific LUT primitives are employed when the system is compiled in Xilinx devices. We will consider two dynamic realizations based on this core in Section 4.

3.1. Description. The FIR filter module is shown in Figure 1. It shows the FIR filter module with its inputs, outputs, and parameters. Signal "E" controls the input validity. Clearing the register chain ("sclr" signal) at will is an important requirement when performing filtering on finite size signals.

We present two filter implementations in Figure 2. A simplified approach is possible for symmetric filters (see Figure 2) [24]. The more general, nonsymmetric case is also presented in Figure 2.

Here, N denotes the number of taps, NH represents the input/coefficients bitwidth, L is the LUT input size (explained in next subsection). We also use OP for controlling the output truncation scheme: (i) LSB truncation then saturation, (ii) LSB and MSB truncation, and (iii) no truncation. We use the parameter format [NO NQ] to denote the fixed-point output format for NO bits with NQ fractional bits. The filter coefficients are specified in an input text file.

We define $M = \lceil N/2 \rceil$, size$I$ = NH + 1 for symmetric filters, and $M = N$, size$I$ = NH for nonsymmetric filters. The inputs/coefficients format is set at [NH NH-1], which restricts values to $[-1, 1)$. As a result, the maximum number of output integer and fractional bits results

$$\left[2(\mathrm{NH}-1) + \left\lceil \log_2(N+1) \right\rceil + 1 \quad 2(\mathrm{NH}-1)\right]. \tag{1}$$

3.2. FIR DA Implementation. The Distributed Arithmetic technique rearranges the input sequence samples (be it $x[n]$ or $s[n]$) into vectors of length M, which require an array of size IM-input LUTs. This becomes prohibitively expensive when M is large. For efficient implementation, we divide the filter into M/L filter blocks [24], as illustrated in Figure 2. Each filter block works on L coefficients requiring sizeIL-input LUTs (each vector of size$L$ goes to one L-input LUT, see Figure 3). Table 1 summarizes the resources savings associated with the filter blocks approach. An advantage of using FIR filter blocks is that it allows for efficient routing while mapping the implementation to the specific LUT primitives found in an FPGA. As shown in [10], the approach is scalable in that can be easily ported to different LUT sizes.

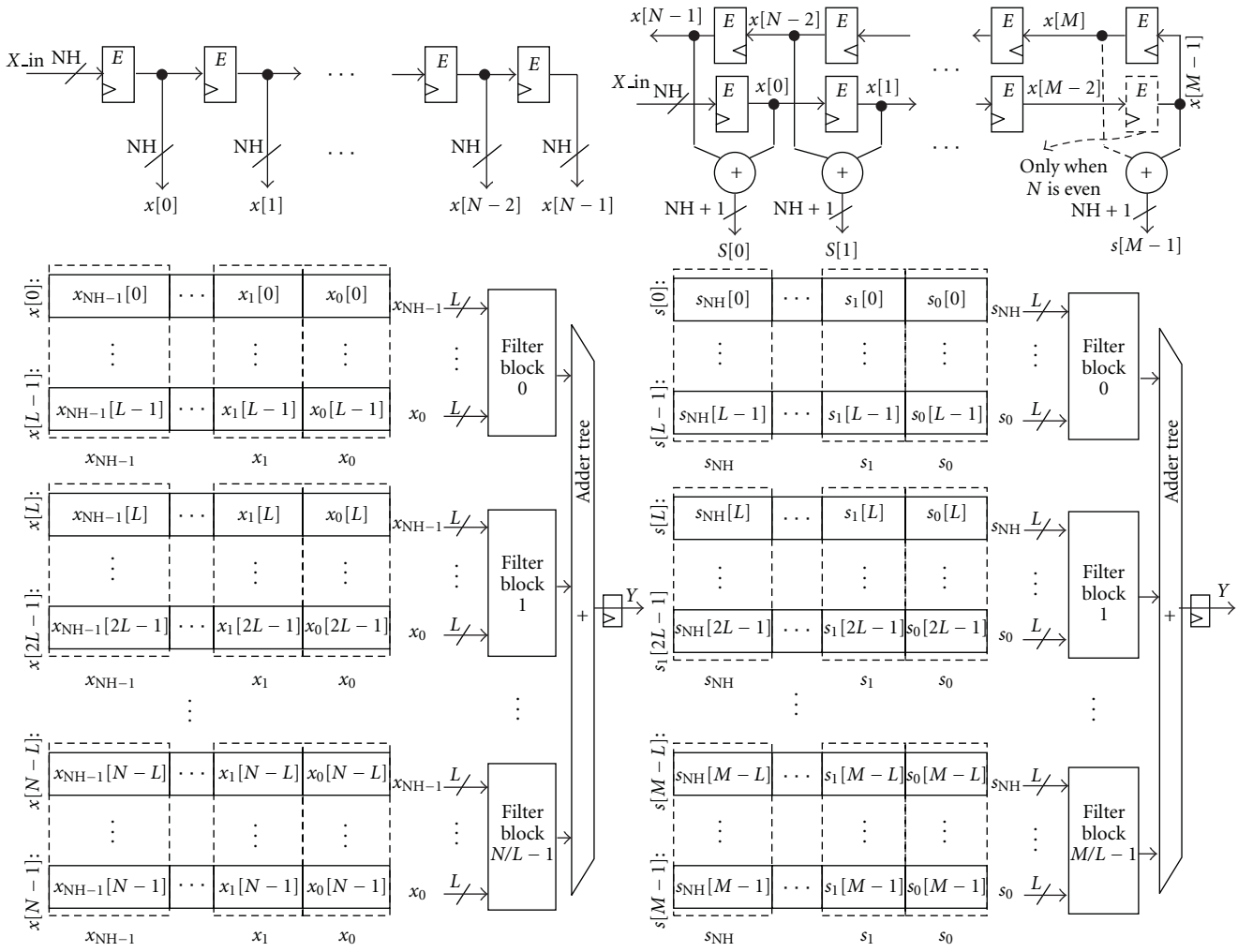


Figure 2: High-performance DA implementation based on the underlying LUT input size (L). Nonsymmetric filter (left) symmetric filter (right).

To demonstrate the savings, we consider a particular example. Using the formulae of Table 1, for $M = 16$, $L = 4$, we have significant savings since $2^{16} \gg 2^4 \times 16/4$. It does require an additional adder tree structure (see Figure 2). However, compared to the savings, the overhead is not significant.

A pipelined implementation of a symmetric filter block example is shown in Figure 3. Here, we have the parameters SYMMETRY = YES and NH = 8. It consists of an array of L-input LUTs, an adder tree, shifters, and registers. The number of register levels is given by the following formula:

$$\#\text{ of register levels in Filter Block} = \left\lceil \log_2(\text{size } I) \right\rceil. \quad (2)$$

The L-input LUT subblocks are shown in Figure 4. Here, the output word size of each L-input LUT is given by LO = NH + $\lceil \log_2(L) \rceil$. It also shows its decomposition into LO L-to-1 LUTs, useful for efficient FPGA implementation. Xilinx FPGA devices contain L-to-1 LUT primitives with $L = 4$ (Spartan-3, Virtex-II Pro, Virtex-4) and $L = 6$ (Virtex-5). Thus, $L = 4$ or $L = 6$ are optimum values of choice. Moreover, as explained in [10] for Virtex-4, optimal LUT implementations can also be obtained for $L = 5, 6, 7, 8$.

Figure 5 depicts the internal pipelined architecture of the adder tree that is used for adding the filter blocks outputs. The result is stored in an output register. The number of register levels of the adder structure is given by

$$\#\text{ of register levels in Filter Adder Structure} = \left\lceil \log_2\left(\frac{M}{L}\right) \right\rceil. \quad (3)$$

Since we can quantize the LUT table values (i.e., the summations), rather than the coefficients, this FIR DA Implementation is slightly less sensitive to quantization noise than a normal implementation, with quantized coefficients.

The latency of the pipelined system is shown in Figure 6. The latency (input-output delay) is given by REG_LEVELS = $\lceil \log_2(\text{size } I) \rceil + \lceil \log_2(M/L) \rceil + 2$ cycles, where REG_LEVELS is the number of register levels between the input and the output.

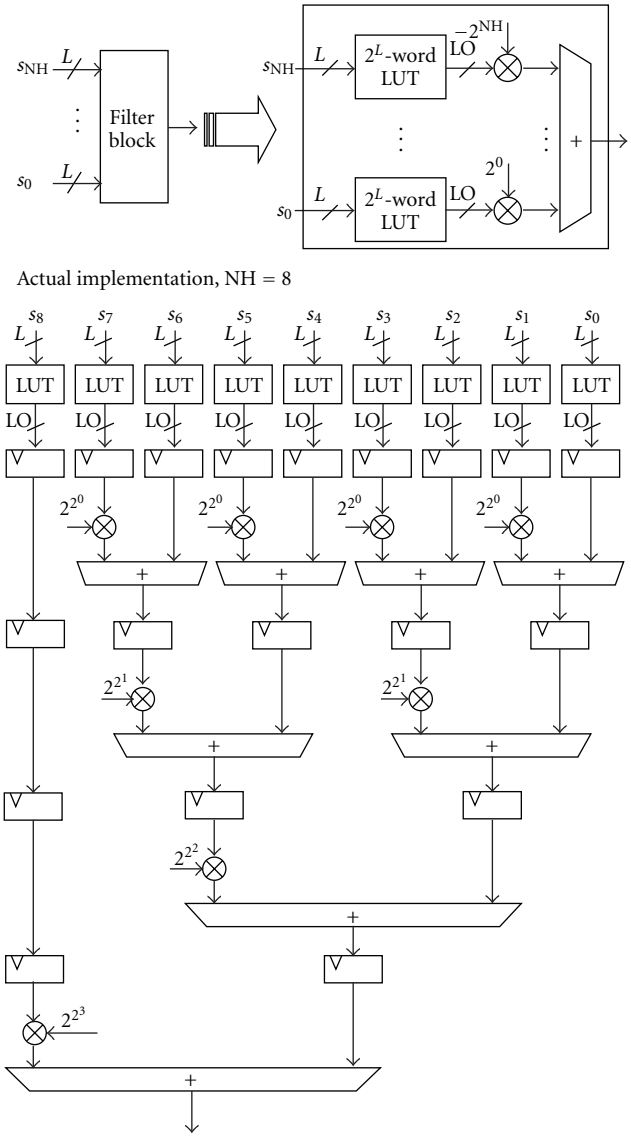


Figure 3: Filter block architecture. SYMMETRY = YES.

Table 5: Reconfiguration time for both DPR system realizations. the 43 kb bitstreamcorresponds to coefficient-only reconfiguration case. The 83 kb bitstream corresponds to full-filter reconfiguration.

Scenario	Reconfiguration speed	Reconfiguration time	
		43 KB bitstream	83 KB bitstream
(1) Current	3.28 MB/s	13.10 ms	25.30 ms
(2) Custom [20]	295.4 MB/s	0.145 ms	0.280 ms
(3) Ideal	400 MB/s	0.107 ms	0.207 ms

4. Dynamically Reconfigurable FIR Filtering System

We now extend the basic FIR filter core to be dynamically reconfigurable. We allow for the dynamic reconfiguration of both the number and the filter coefficients themselves in an embedded system. The basic system is shown in Figure 7. By means of dynamic partial reconfiguration, we turn a constant coefficient FIR filter into an adaptive FIR filter.

The basic approach requires that we prespecify the Partial Reconfiguration Region (PRR). We consider two dynamically reconfigurable realizations.

(1) *Coefficient-only reconfiguration*: The PRR allows modifications to the filter coefficient values, while keeping the rest of the architecture intact.

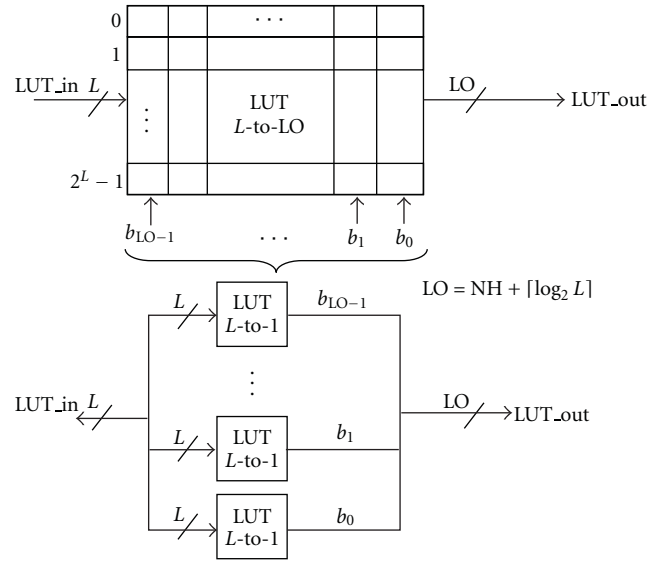


Figure 4: Realization of an L-to-LO LUT using LO L-to-1 LUTs.

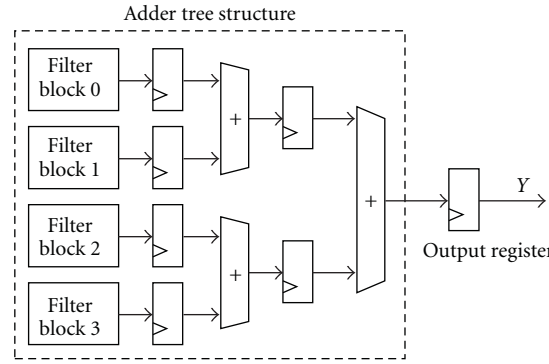


Figure 5: Adder tree structure for filter blocks' outputs. $M/L = 4$.

Table 6: DPR system throughput (MSPS) as function of delay between reconfigurations for 1D fir filtering with full filter reconfiguration.

Scenario	Number of samples between reconfigurations				
	2048 K	1024 K	409.6 K	204.8 K	102.4 K
	Amount of time between reconfigurations				
	68 ms	34 ms	13.6 ms	6.8 ms	3.4 ms
(1) Current	21.9	17.2	10.5	6.3	3.5
(2) Custom [20]	29.9	29.8	29.5	28.9	27.8
(3) Ideal	30.0	29.9	29.6	29.2	28.3

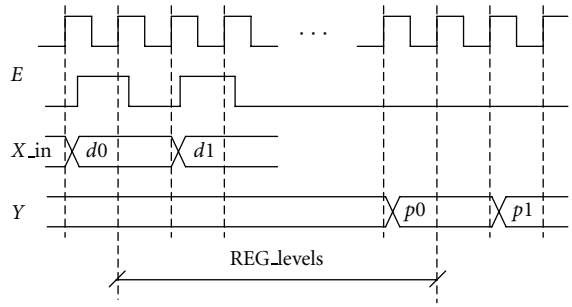


Figure 6: Latency measured from the moment "$d0$" is input until its correspondent output "$p0$" is available.

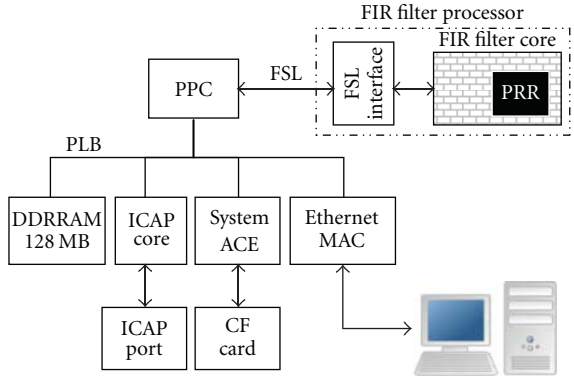


Figure 7: System block diagram.

(2) *Full-filter reconfiguration:* The PRR allows modification to the number of coefficients, the coefficient values, and the filter symmetry.

We start by describing the system architecture and FIR filter dataflow, which are not affected by the PRR definition. Then, we explain each of the dynamic realizations by providing a detailing representation of the PRR in the context of the FIR filter architecture.

4.1. System Architecture. From Figure 7, we can see that the dynamic FIR core and the PowerPC (PPC) communicate using the high speed FSL bus. The Partial Reconfiguration Region (PRR) is dynamically reconfigured via the internal configuration access port (ICAP), driven by the ICAP controller core.

The DDRRAM stores volatile data needed at run time, for example, input streams, processed streams, and partial bitstreams. At power-up, SystemACE reads a Compact Flash (CF) Card that stores the partial bitstreams and input streams. The processed streams are written back to the DDRRAM. The Ethernet core provides reliable communication with a PC and allows us to get new partial bitstreams or new input streams and to send processed streams to the PC for its verification or storage. Also, it serves as an interface for throughput measurements and system status.

Figure 8 depicts the interfacing of the FIR filter processor and the PPC for both dynamic realizations. The FIR filter processor, as shown in Figure 7, consists of the FIR filter core and a control unit that provides interfacing with the 32-bitwide FSL bus. Figure 8 shows a special case when the filter input size is NH = 8 bits. Here, the input is processed sample by sample (one byte at a time). After 32 output samples are computed, they are transmitted through the FSL bus. Other input/output bit-width configurations require different logic and control.

We next provide a description of the different possible modes of operation. First, we note that an FIR filter with N coefficients and NX input values can output a maximum of NX + N − 1 values. The three modes of operation are implemented through a finite state machine as follows.

(i) *Basic output mode*: The system computes the first NX output values. This mode is useful for finite 1D signals.

(ii) *Symmetric output mode*: The system computes the central NX output samples (i.e., in the range $\lfloor N/2 \rfloor + 1 : \mathrm{NX} + \lfloor N/2 \rfloor$). This mode is useful when performing 2D separable convolution on images.

(iii) *Streaming mode*: with infinite number of input samples, that is, NX = ∞.

4.2. FIR Filter Processor Data Flow. The FIR Filter processor receives and sends 32 bits at a time via the FSL bus. Due to the FIFO-like nature of the FSL bus [25], the PPC processor sends a data stream to FIFOw to be grabbed by the FIR filter processor that in turn writes an output data stream on FIFOr to be retrieved by the PPC processor (see Figure 8).

We optimize FSL bus usage by letting the PPC write a large block of data on FIFOw. The FIR filter processor then processes the data and writes the results on FIFOr in a pipelined fashion. After reading all data in FIFOr, the PPC writes another large block of data on FIFOw, that is, the PowerPC is busy only when reading/writing each large block of data. In addition, the FIR filter processor starts reading the next available block of data on FIFOw right after writing a processed chunk of data on FIFOr. Each FIFO depth has been set to 64 words (32-bit words).

4.3. Dynamic Partial Reconfiguration Setup. Figure 8 presents two dynamically reconfigurable systems and the associated PRRs. In the full-filter reconfiguration case, we do not allow any changes to the I/O bit-width. Here, we note that a change to the I/O bit-width would also require a generalized FSL interface to be included in the PRR, further complicating the design. Despite the complexity of doing so, this will be of interest for allowing us to build a dynamic precision system.

The static region is defined by everything else outside the PRR, including FSL interface, FSL circuitry, peripheral controllers, and the FIR filter core static portion (coefficient-only reconfiguration).

All signals between the dynamic region (PRR) and the static part are connected by prerouted bus macros in order to lock the wiring. Also, the PRR I/Os are registered as the reconfiguration guidelines advise [26]. To perform DPR, the partial bitstreams are read from a CF card and stored in

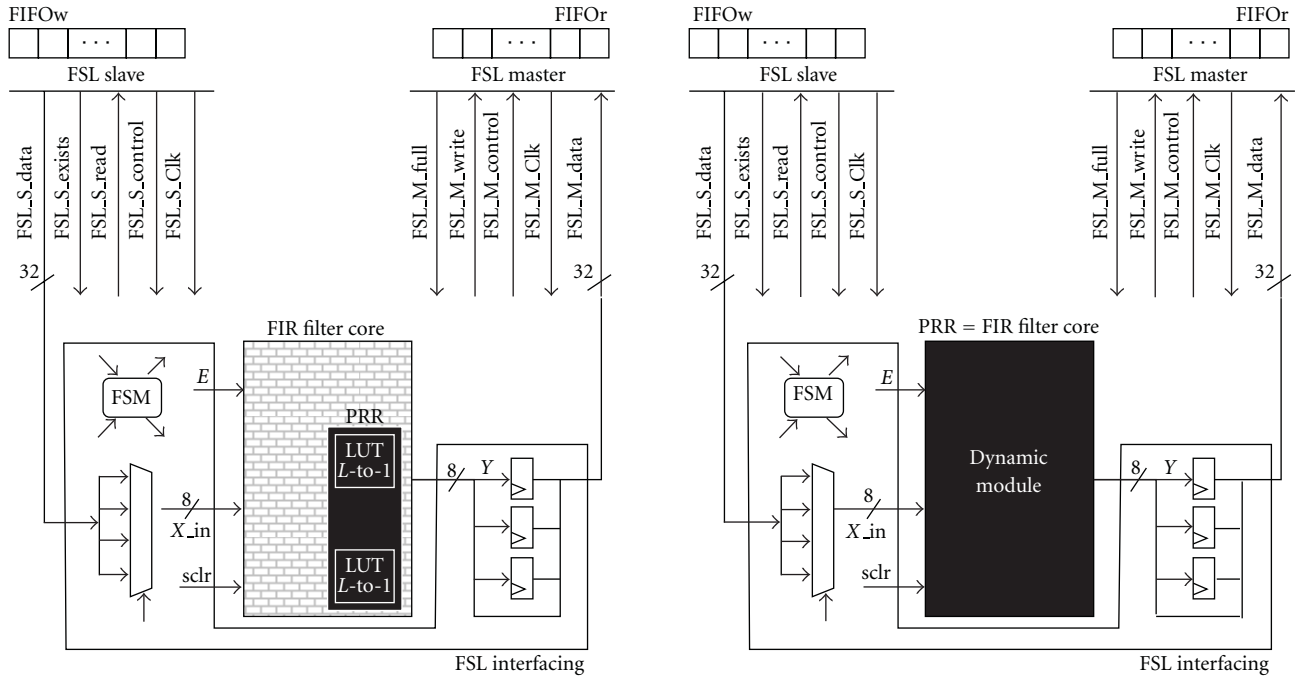


Figure 8: Dynamic FIR filter processor interfacing with FSL. PRR for dynamic reconfiguration of the coefficients (left) and PRR for dynamic reconfiguration of the number of coefficients, their values, and symmetry (right).

DDRRAM. When needed, they are written to the ICAP port. This fairly simple technique is explained in [21].

For throughput measurement purposes, the partial bitstreams and the input set of streams reside on DDRRAM. The streams are sent to the FIR Filter processor, and the output streams are written back to the DDRRAM. This process is repeated with different partial reconfiguration bitstreams loaded at specific rates, so as to get different filter responses and measure performance as the reconfiguration rate varies.

4.3.1. Coefficient-Only Reconfiguration. In this dynamic realization, the dynamic region is made of $(M/L) \times \text{size}\, IL$-to-1 LUTs, resulting in a PRR with $(M/L) \times \text{size}\, I \times L$ inputs and $(M/L) \times \text{size}\, I \times \text{LO}$ outputs. Figure 9 depicts the PRR along with the bus macros when SYMMETRY = NO, NH = 8, N = 8, L = 4. The PRR is depicted in the context of the FIR filter core.

This realization is very useful for applications that only require filter coefficients modification, and it exhibits a smaller reconfiguration time overhead than the full reconfiguration case. Also, since only the LUT values are modified, the routing inside the PRR does not change. This has potential advantages in the area of run-time bitstream generation, as there is no need for run-time place-and-route operation. Fast routing is a very demanding task, and in most cases, it cannot be performed at run time [27].

4.3.2. Full-Filter Reconfiguration. In this case, the PRR involves the entire FIR filter core. It enables us dynamically modify the coefficients, number of coefficients, symmetry, and LUT input size. Figure 10 depicts the PRR along with the bus macros in the context of the FIR filter processor (with the FSL interface). We can see that the PRR has NH+2 inputs and NH outputs.

5. Results

5.1. Stand-Alone Fir Filter Core. Figure 11 shows hardware resource utilization as a function of the number of coefficients (N), input bitwidth (NH), and symmetry (dotted lines: nonsymmetric filters, solid lines: symmetric ones). Also, we set OP = 0, L = 4. Here, we use the XC4VFX20-11FF672 Virtex-4 device, with 8544 slices.

In addition, for each input bitwidth, we are considering the largest output format attainable (in the range $[-1, 1)$). The output format ([NO NQ]) plays a negligible role in resource consumption (a difference of at most 12 slices).

Regarding frequency of operation, the goal of 200 MHz minimum frequency of operation was attained in all cases.

In addition, an error analysis is performed for the same parameters. Figure 12 shows the relative error curves for three cases (input stream = 1024 sinusoid samples). The error metric is

$$\text{Relative error} = \left| \frac{\text{ideal value} - \text{FPGA output}}{\text{ideal value}} \right|. \quad (4)$$

Figure 12 shows that in most cases the relative error is below 5%. The peaks correspond to FPGA values of zero and ideal values close to zero, resulting in a deceptive 100% error.

5.2. Embedded System. Results are shown using the following FIR Filter core parameters: N = 32, NH = 8, [NO NQ] = [8 7], L = 4, OP = 0, SYMMETRY = YES.

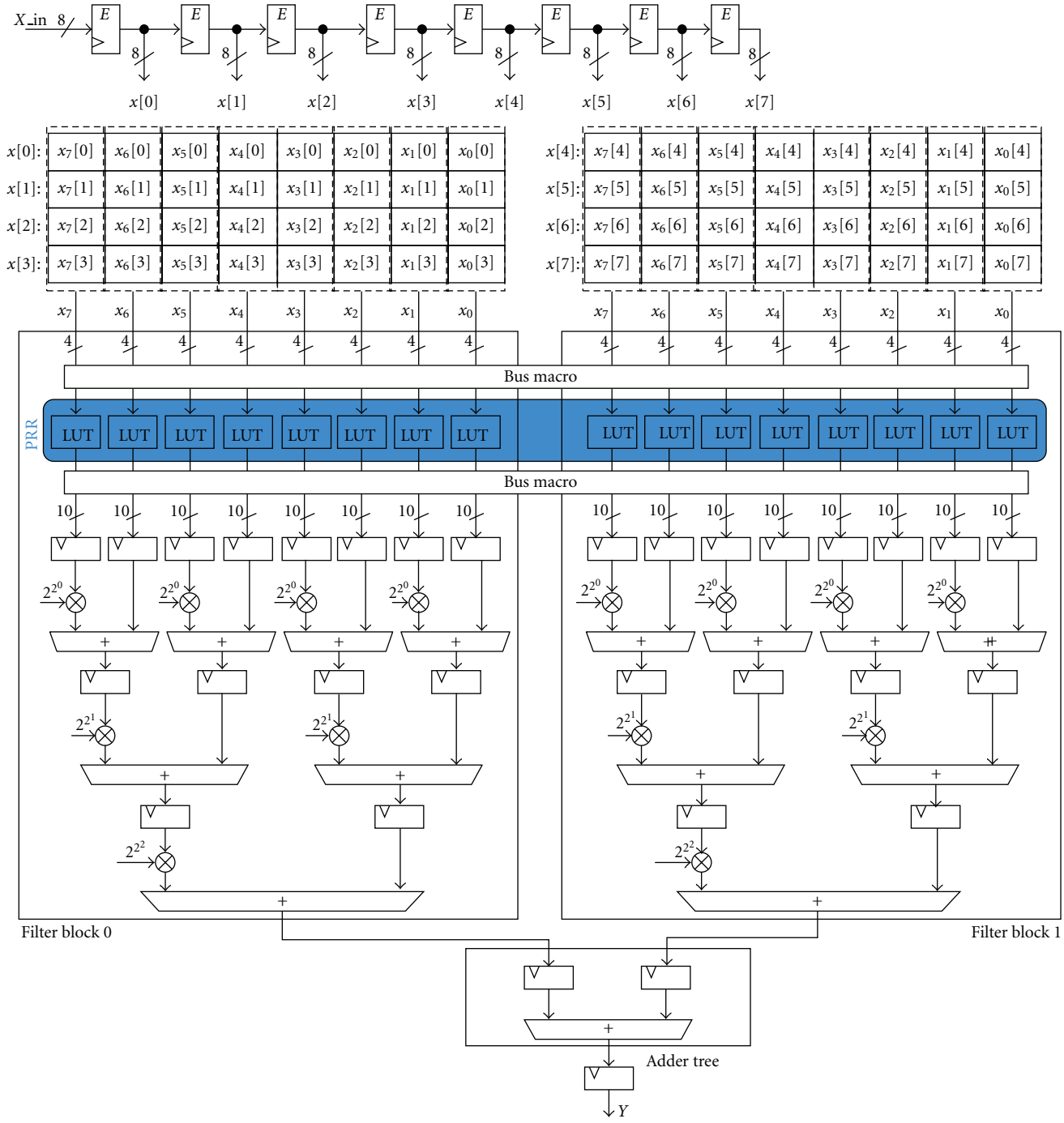


Figure 9: FIR filter core where the PRR and bus macros can be appreciated. Here, we can modify only the coefficients via the LUTs.

The system is implemented on the ML405 Xilinx Development Board that houses a XC4VFX20-11FF672 Virtex-4 FPGA. The PPC is clocked at 300 MHZ and the peripherals run at 100 MHz.

In order to improve performance, the DDRRAM memory space is cached. Also, the dynamic systems are tested in the basic output mode; that is, only the first NX outputs are considered.

5.2.1. Hardware Resource Utilization. Results for this section depend on the specific dynamic realization. Tables 2 and 3 show hardware resource utilization for two DPR systems: (i) coefficient-only reconfiguration and (ii) full-filter reconfiguration. It shows the static region, dynamic region and the entire system resource usage. The module "PRR interface" is the gluing static logic needed to join the static and dynamic regions.

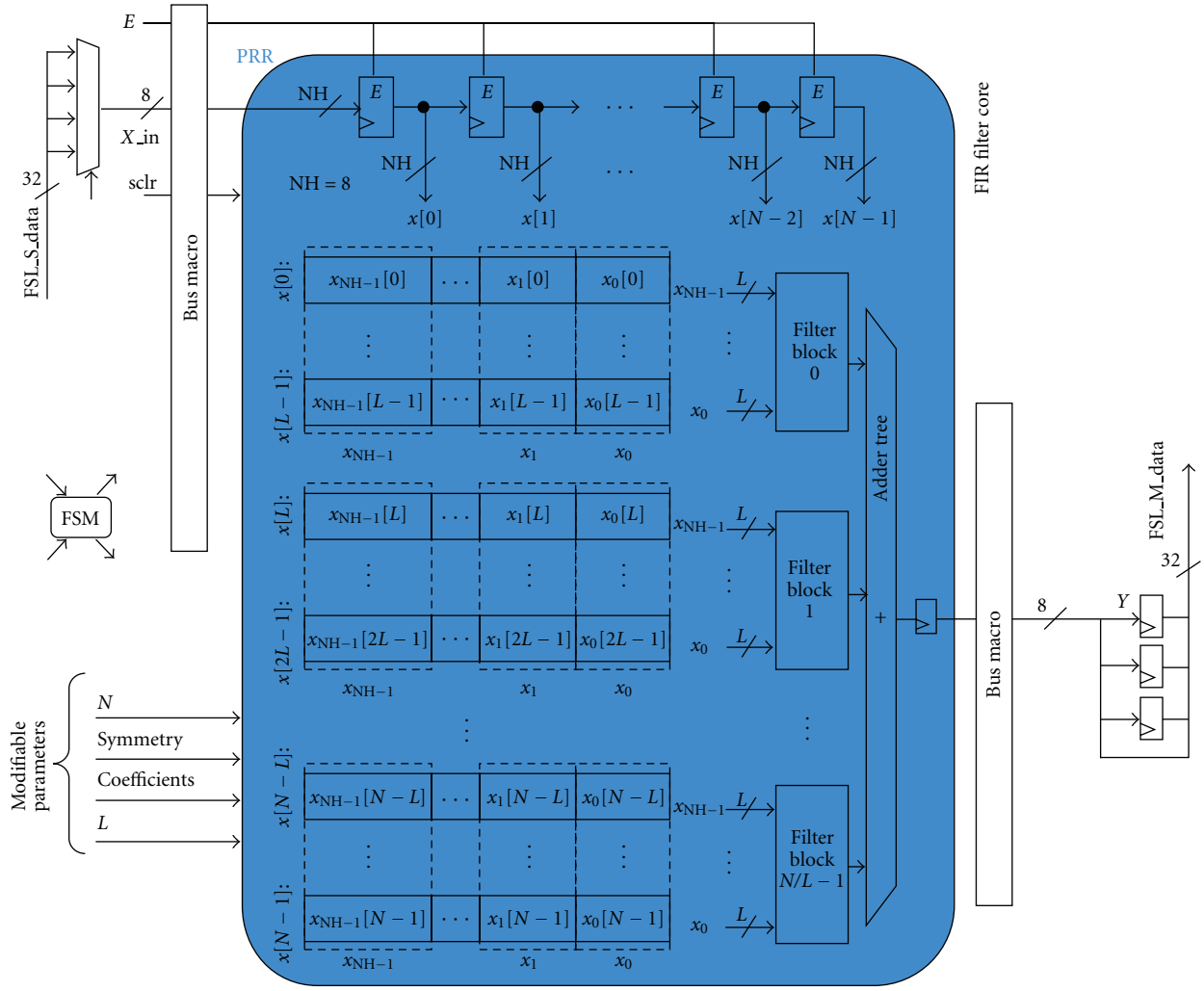


Figure 10: FIR filter processor where the PRR is the FIR Filter core. Note the parameters we can modify.

As expected, the overall resource utilization is about the same. What varies is the static region size, which is larger in the coefficient-only reconfiguration case.

Table 4 shows the reconfiguration size and its partial bitstream size. Note that the PRR in the first case is somewhat larger than expected (about 62% of the second case). This can also be appreciated in Figure 13 that shows the dynamic region (PRR) for both realizations, which are functionally the same.

The reason for the large PRR in the first case is the large number of required bus macros I/Os. In the coefficient-only reconfiguration case, the system needs access to the LUTs (see Section 4.3.1). As a result, for the special case shown, we require $(M/L) \times \text{size}\, I \times L = 4 \times 9 \times 4 = 144$ inputs and $(M/L) \times \text{size}\, I \times \text{LO} = 4 \times 9 \times 10 = 360$ outputs.

As explained in Section 4.3.2, in the second case (full-filter reconfiguration), we only need NH + 2 = 10 inputs and NH = 8 outputs. So, the PRR in the first case is larger than what it is actually needed for the L-to-1 LUT array, thereby wasting hardware resources in order to accommodate the large number of bus macros I/Os.

5.2.2. FIR Filter Processor Performance Bounds. The maximum throughput of this particular FIR filter processor (NH = 8) is given by

$$\text{Max.Throughput} = \frac{1\,\text{byte}}{1\,\text{cycle}} = \frac{8\,\text{bits}}{10\,\text{ns}} = 0.8\,\text{Gbps}. \quad (5)$$

Note that since the system is pipelined, there is an initial setup delay that becomes negligible over time. Actual throughput depends on many factors, such as cache size, PPC instruction execution, and FSL usage. Note that the maximum throughput of (5) cannot be attained since the PPC cannot read and write into the FIFOs at the same time.

5.2.3. Reconfiguration Time. Table 5 shows the reconfiguration time for 3 scenarios. Both dynamic realizations are included. In our setup, called Scenario 1, we used the Xilinx ICAP core and obtained a reconfiguration average speed of 3.28 MB/s. The reconfiguration time of Scenario 2 is computed based on the speed results reported in [22]. The dramatic improvement in reconfiguration lies on the use of a custom ICAP controller, DMA access, and burst transfers.

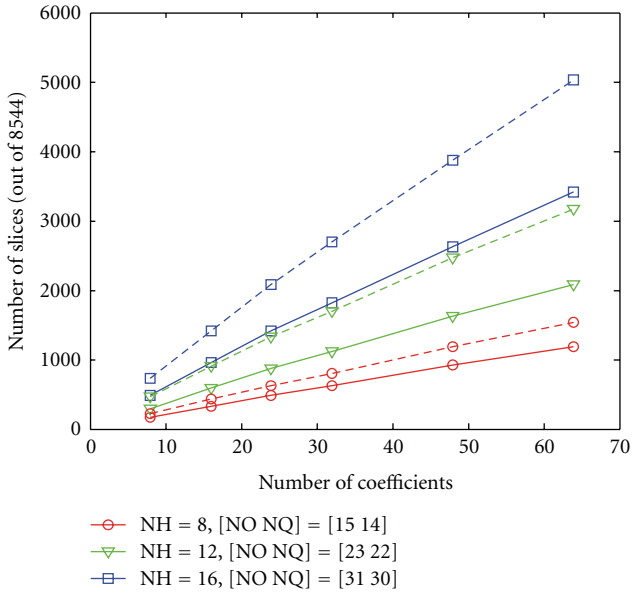


Figure 11: Resources versus number of coefficients and input bitwidth. Solid lines represent the symmetric case. Dotted lines represent the nonsymmetric case.

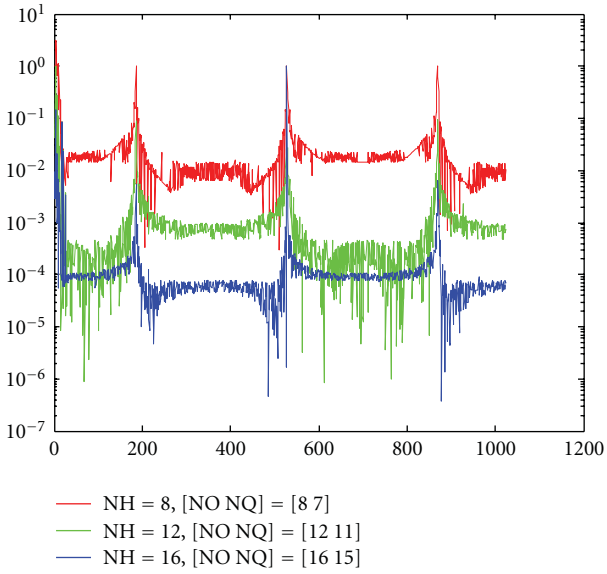


Figure 12: Relative error, $N = 32$. Three bitwidth cases.

Scenario 3 is the maximum theoretical throughput, which for the Virtex-4 is 400 MB/s [21].

5.2.4. Dynamic Performance. We use software timers to measure the elapsed time from the moment we start reading the input stream from DDRRAM until the processed stream is written back on DDRRAM. We are considering sinusoids as our inputs. Here, we refer to Section 4.3 for some of the details that will be discussed in this section.

In order to evaluate the dynamic performance of the system, we use a stream of 102400 samples (1 sample = 8 bits). The stream is processed a number of times (100 runs). Within the 100 runs, partial bitstreams are loaded at a specific rate. Each partial bitstream amounts to a different filter response.

Note that for the coefficient-only reconfiguration case, we only load a different set of coefficient values.

For the full-reconfiguration case, we switch between a filter with $N = 32$ coefficients and one with $N = 16$ coefficients. The PRR size is defined to be sufficiently large so as to allow implementation of the larger filter; that is, the $N = 32$ filter case. The filter with $N = 16$ requires only one fewer latency cycle (3). As a result, the static performance improvement of the smaller filter is not significant.

We report the average throughput over the 100 runs. Here, we define the dynamic reconfiguration rate in terms of the inverse of the number of samples that are being processed prior to a hardware reconfiguration. For better visualization, we report throughput in terms of the number of processed Mega samples per second (MSPS). This corresponds to the inverse of the reconfiguration rate.

Figures 14 and 15 show the dynamic performance over 100 runs for both dynamic realizations. There are 3 curves that correspond to the 3 scenarios shown in Table 5. In the limit, at zero reconfiguration rate, we have static performance. The performance results converge for the static case.

From Figure 14 (coefficient-only reconfiguration), we see that for Scenario 1 (our actual measurements), the static performance resulted in 29.25 MSPS. At the maximum reconfiguration rate (1 reconfiguration every stream), the dynamic performance resulted in 6.1651 MSPS. The other curves (Scenarios 2 and 3) provide performance bounds based on the static performance and reconfiguration speeds of Table 5.

We can appreciate that the dynamic performance of the full-filter reconfiguration case is slightly lower than the coefficient-only reconfiguration. This is due to differences in the PRR size. But as we increase the number of samples before a reconfiguration, or use a scenario other than the first one, this effect is less noticeable.

As expected, the dynamic performance heavily depends on reconfiguration speed and input stream size. Better reconfiguration speeds offset the reconfiguration time overhead (Scenarios 2 and 3). We have the same effect for smaller dynamic regions. The slower reconfiguration rates due to longer data streams help to offset the reconfiguration overhead as well.

In Table 6, we present the full-filter reconfiguration system throughput as a function of the time between reconfigurations. It is quite clear from the results that even for the slowest scenario, we can maintain throughputs over ten MSPS while dynamically reconfiguring seventy times per second.

5.3. Experimental Results with ECG Processing. We present an example application for electrocardiogram (ECG) characterization (R-wave detection). Here, we consider coefficient-only reconfiguration for implementing a 3-channel, 1D

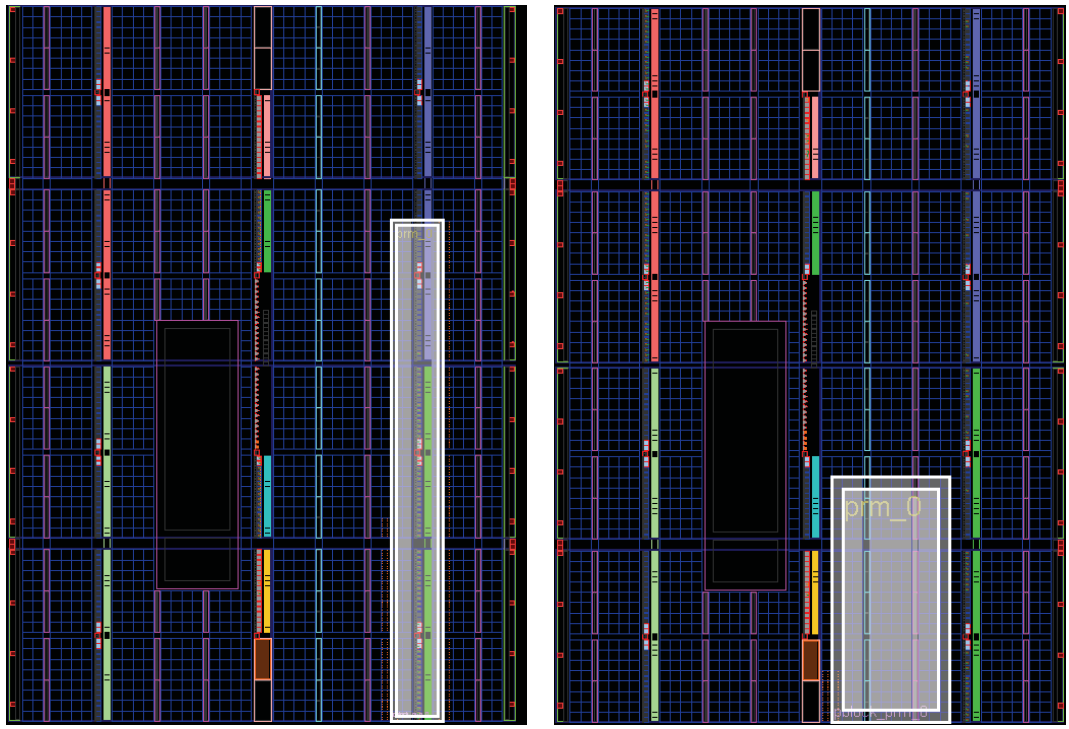


Figure 13: Dynamic reconfiguration region for (i) coefficient-only reconfiguration system (left), and (ii) full-filter reconfiguration system (right).

filterbank. We make use of the embedded system detailed in Section 5.2. Each channel filter is symmetric, with 32 8-bit coefficients for 8-bit I/O, using truncation (saturation) arithmetic for the outputs. Our approach here is to implement a variation of the ECG processing algorithms presented in [28].

ECG signal processing is of great interest for emergency applications, including the detection of cardiac arrhythmias [29] and stenosis assessment for atherosclerotic plaque video analysis [30]. A popular approach based on [28] is to use the outputs of a Wavelet filterbank for ECG analysis.

As in Wavelet analysis, we design a dyadic filterbank to cover the entire, discrete frequency space. We have a high-pass filter with a positive frequency pass-band from $\pi/2$ to π, a band-pass filter from $\pi/4$ to $\pi/2$, and a low-pass filter for frequencies up to $\pi/4$. For each channel filter, we consider efficient implementations using 32 8-bit coefficients. The magnitude response of the designed filterbank is shown in Figure 16.

For testing the implementation, we use the first recording (record 100) from the MIT arrhythmia database [31]. In this record, we have 2 channels with 650 K samples sampled at 360 Hz and quantized at 11-bits over a 10 mV range. We further quantized the input down to 8 bits, downloaded them to the DDRRAM using the Ethernet core and tested using the procedure outlined in Figure 17.

Based on [28], we implemented a simple R-wave detection algorithm. For detection, we look for thresholds in the outputs. In the example of Figure 18, we threshold as follows: low-pass filter ($[-1/32, 1/32]$), band-pass ($[-1/32, 1/32]$), and high-pass ($>1/64$). This results in perfect R-wave detection for the first 5 cycles of the second channel (1500 samples). We refer to [28] for more details on how to adjust thresholds in such algorithms for near-perfect results verified over the entire database. Our goal is to simply demonstrate the DPR FIR system on real signals.

The detection algorithm is included in the embedded PowerPC software routines, and the resulting signal is stored into the DDRRAM. We note that performance improves with larger input signals (see Figure 14). The detection algorithm is performed at the end of the operations and takes about 80 ms. Dynamic reconfiguration of a channel filter requires 13.1 ms. At a sampling rate of 360 Hz, the system allows significant time for implementing real-time detection algorithms and DPR. As a result, the number of samples that are processed prior to reconfiguration can be significantly reduced. By processing every 2000 samples, the processing rate stands at 4.62 MSPS (2000 samples takes 140 ms to process). Thus, after 5.5 seconds spent in acquiring 2000 samples, we get a detection response in 140 ms.

5.4. Comparison with Other PR Systems For FIR Filtering. The majority of previously reported work on FIR filtering is based on multiply-and-add approaches [14, 16, 17, 22]. In [14], the authors reported a reconfiguration time of 1.5 ms for changing the coefficients and their number on a Virtex-II (74.7 KB bitstream). In [16, 17], the authors presented a DPR FIR system that only allowed for changes in the number of coefficients. Reconfiguration time for 8794 slices for a 20-tap filter required 700 ms. The filter presented in [22] most

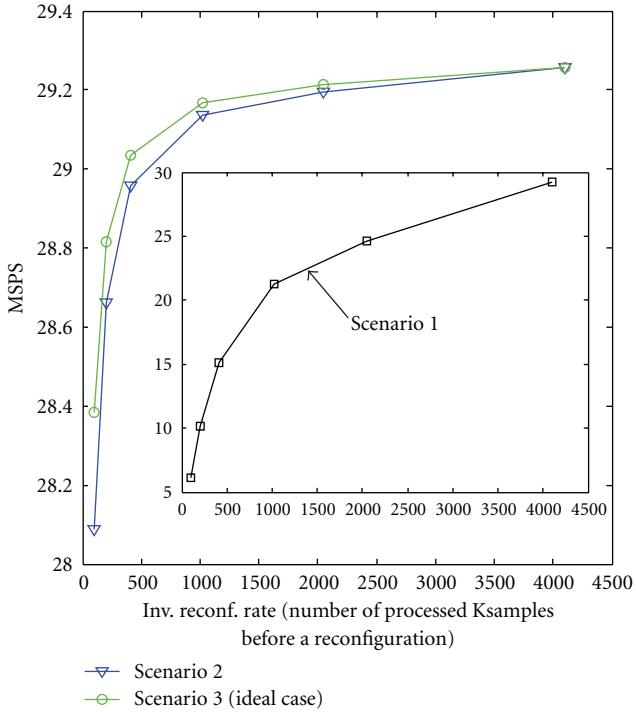


Figure 14: DPR system performance for coefficient-only reconfiguration.

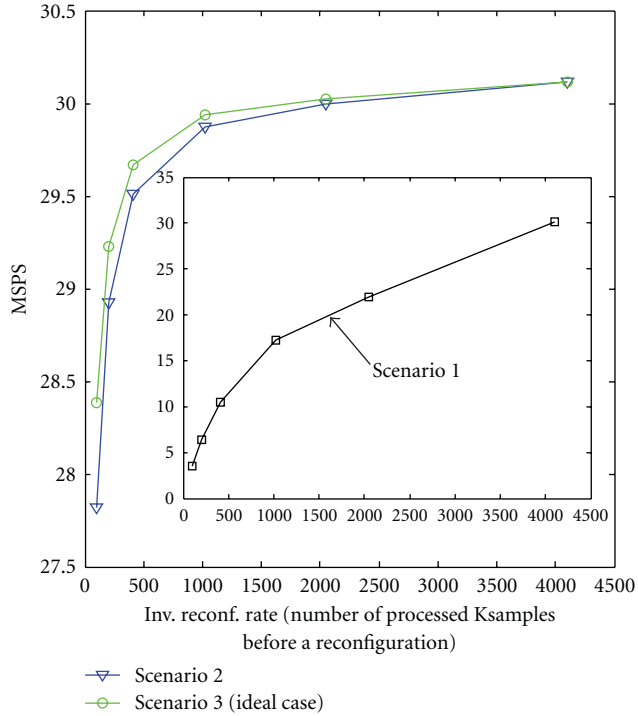


Figure 15: DPR system performance for full-filter reconfiguration.

closely resembles our FIR filter: 32-taps, 8-bit coefficients, 8-bit input, but with multiply-and-add approach. It required 1985 LUTs for a 13.1 ms reconfiguration time. We can change the entire filter using a 83 KB bitstream for a reconfiguration time of 25.3 ms.

As mentioned earlier, for FPGA implementations, the distributed arithmetic presented here is far better suited than these multiply-and-add approaches. DA approaches allow for efficient use of hardware resources. Beyond this, multiply-and-add approaches tend to have fixed input/output characteristics as opposed to the flexible, dynamically reconfigurable arithmetic representations presented here.

The constant-coefficient filter with DPR is mentioned in [15], but the work is more theoretical and the results are noncomparable with ours, as stated in Section 2.

6. Conclusions

We presented two efficient dynamic partial reconfiguration systems that allow us to implement a wide range of 1D FIR filters. Requiring a significant smaller partial reconfiguration region, the first system allows changes to the FIR filter coefficients while keeping the rest of the architecture intact. Using a larger partial reconfiguration region, the second system allows full-filter reconfiguration. This system can be used to switch between FIR filters based on power, performance, and resources considerations.

For both systems, the required partial reconfiguration region is kept small by using Distributed Arithmetic implementations. System performance is evaluated in terms of the

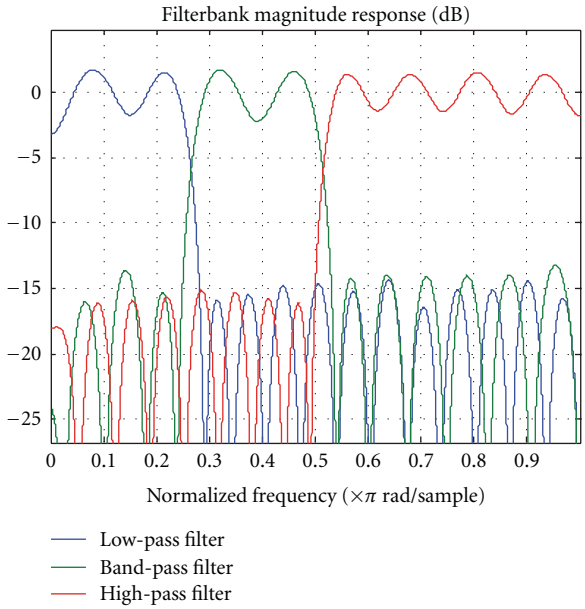


Figure 16: Filterbank used for R-wave detection.

dynamic reconfiguration rate. For a representative example, it is shown that we can process over ten Mega samples per second while dynamically reconfiguring about seventy times per second. The introduction of faster dynamic reconfiguration controllers can lead to much higher throughputs for the same number of reconfigurations per second. Alternatively, we can maintain much higher throughputs at much lower reconfiguration rates.

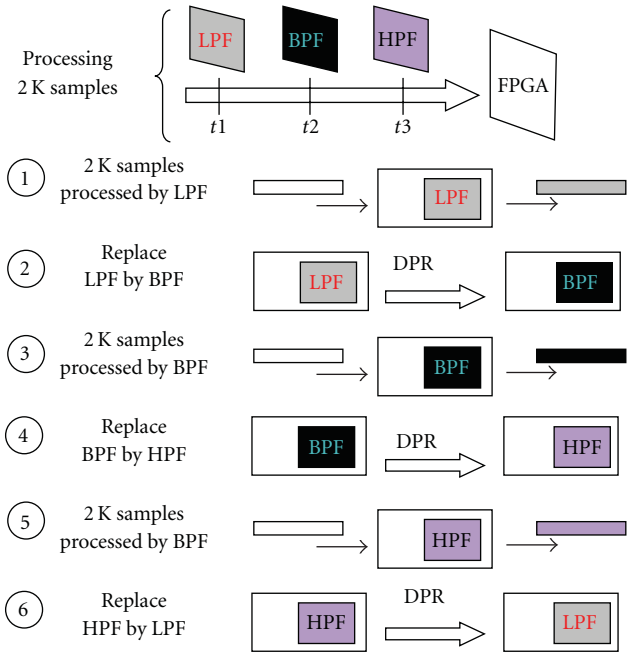


Figure 17: Filterbank data processing. LPF: low-pass filter, BPF: band-pass filter, HPF: high-pass filter.

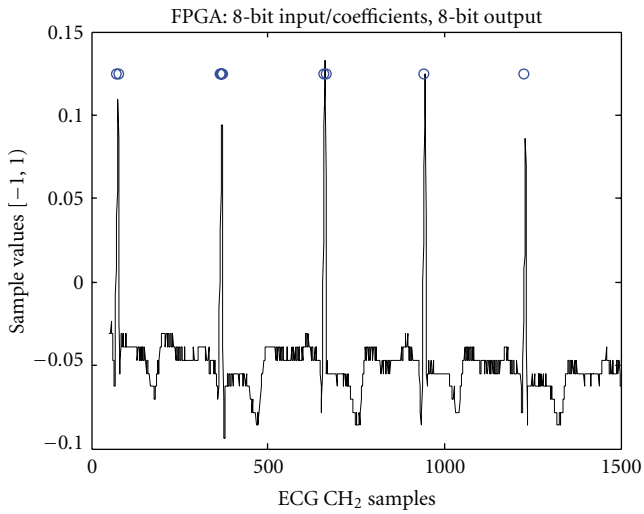


Figure 18: Perfect detection of R-waves for the first 5 ECG cycles.

The results have encouraged us to explore the use of dynamically reconfigurable filtering for digital image and video processing applications. As seen from the results of this paper, it is possible to dynamically reconfigure at real-time frame rates. For such applications, the DPR systems can be extended to separate implementations of 2D dynamically reconfigurable filterbanks.

Acknowledgment

The research presented in this paper has been funded by the Air Force Research Laboratory under Grant no. QA9453-060C-0211.

References

[1] M. Hatamian and G. L. Cash, "A 70 MHz 8 bit x 8 bit parallel pipelined multiplier in 2.5 μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, 1986.

[2] K. G. Smitha and A. P. Vinod, "A reconfigurable high-speed RNS-FIR channel filter for multi-standard software radio receivers," in *Proceedings of the 11th IEEE Singapore International Conference on Communication Systems (ICCS '08)*, pp. 1354–1358, Guangzhou, China, 2008.

[3] F. Gallazzi, G. Torelli, P. Malcovati, and V. Ferragina, "A digital multistandard reconfigurable FIR filter for wireless applications," in *Proceedings of the 14th IEEE International Conference on Electronics, Circuits and Systems (ICECS '07)*, pp. 808–811, December 2007.

[4] H. Bruce, R. Veljanovski, V. Öwall, and J. Singh, "Power optimization of a reconfigurable FIR-filter," in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation*, pp. 321–324, October 2004.

[5] R. Mahesh and A. P. Vinod, "New reconfigurable architectures for implementing FIR filters with low complexity," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 2, pp. 275–288, 2010.

[6] C. Chou, S. Mohanakrishman, and J. B. Evans, "FPGA implementation of digital filters," in *Proceedings of Signal Processing Applications Technology*, Santa Clara, Calif, USA, 1993.

[7] T. Do, H. Kropp, C. Reuter, and P. Pirsch, "A flexible implementation of high-performance FIR filter on xilinx FPGAs," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications. From FPGAs to Computing Paradigm (FPL '98)*, pp. 441–445, 1998.

[8] S. A. White, "Applications of distributed arithmetic to digital signal processing: a tutorial review," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine*, vol. 6, no. 3, pp. 4–19, 1989.

[9] G. A. Vera, D. Llamocca, M. S. Pattichis, and J. Lyke, "A dynamically reconfigurable computing model for video processing applications," in *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, pp. 327–331, Pacific Grove, Calif, USA, November 2009.

[10] D. Llamocca, M. Pattichis, and A. Vera, "A dynamically reconfigurable parallel pixel processing system," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 462–466, Prague, Czech Republic, August-September 2009.

[11] D. Llamocca, M. Pattichis, and A. Vera, "A dynamically reconfigurable platform for fixed-point FIR filters," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 332–337, Cancun, Mexico, December 2009.

[12] E. C. Tan, A. Wahab, and K. K. Wong, "Programmable DSP systems using FPGA," in *Proceedings of the IEEE Conference on Digital Signal Processing Applications*, vol. 2, pp. 654–658, Perth map, Australia, November 1996.

[13] R. Sidhu and V. K. Prasanna, "Efficient metacomputation using self-reconfiguration," in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream (FPL '02)*, pp. 85–92, Montpellier, France, 2002.

[14] J.-P. Delahaye, J. Palicot, C. Moy, and P. Leray, "Partial reconfiguration of FPGAs for dynamical reconfiguration of a software radio platform," in *Proceedings of the 16th IST Mobile and Wireless Communications Summit*, pp. 1–5, Budapest, Hungary, July 2007.

[15] T. Rissa, R. Uusikartano, and J. Niittylahti, "Adaptive FIR filter architectures for run-time reconfigurable FPGAs," in *Proceedings of IEEE International Conference on Field-Programmable Technology*, pp. 52–59, 2002.

[16] C. S. Choi and H. Lee, "An reconfigurable FIR filter design on a partial reconfiguration platform," in *Proceedings of the 1st International Conference on Communications and Electronics (ICCE '06)*, pp. 352–355, Hanoi, Vietnam, October 2006.

[17] C. S. Choi and H. Lee, "A self-reconfigurable adaptive FIR filter system on partial reconfiguration platform," *IEICE Transactions on Information and Systems*, vol. E90-D, no. 12, pp. 1932–1938, 2007.

[18] K. Chapman, *Constant Coefficient Multipliers for the XC4000E*, Xilinx, San Jose, Calif, USA, 1996, Xilinx's Application Note 054.

[19] *Distributed Arithmetic FIR Filter (DS240)*, Xilinx, San Jose, Calif, USA, v9.0 edition, 2005.

[20] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 535–538, Heidelberg, Germany, September 2008.

[21] G. A. Vera, *A dynamic arithmetic architecture: precision, power, and performance considerations*, Ph.D. dissertation, University of New Mexico, Albuquerque, NM, USA, May 2008.

[22] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 964–969, April 2009.

[23] S. Chevobbe and S. Guyetant, "Reducing reconfiguration overheads in heterogeneous multicore RSoCs with predictive configuration management," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 390167, 7 pages, 2009.

[24] *Implementing FIR Filters in FLEX Devices (AN73)*, Altera Corp., San Jose, Calif, USA, v1.01 edition, 1998.

[25] *Fast Simplex Link (FSL) Bus Product Specification (DS449)*, Xilinx, San Jose, Calif, USA, v2.11a edition, 2007.

[26] *Early Access Partial Reconfiguration User Guide for ISE 9.204i (UG208)*, Xilinx, San Jose, Calif, USA, v1.2 edition, 2008.

[27] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-assisted fast routing," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '02)*, April 2002.

[28] J. S. Sahambi, S. N. Tandon, and R. K. P. Bhatt, "Using wavelet transforms for ECG characterization," *IEEE Engineering in Medicine and Biology Magazine*, vol. 16, no. 1, pp. 77–83, 1997.

[29] E. Kyriacou, C. Pattichis, M. Pattichis et al., "An m-Health monitoring system for children with suspected arrhythmias," in *Proceedings of the 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS '07)*, pp. 1794–1797, 2007.

[30] A. Panayides, M. S. Pattichis, C. S. Pattichis, C. P. Loizou, M. Pantziaris, and A. Pitsillides, "Robust and efficient ultrasound video coding in noisy channels using H.264," in *Proceedings of the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS '09)*, pp. 5143–5146, Minneapolis, MN, USA, September 2009.

[31] G. B. Moody and R. G. Mark, "The MIT-BIH Arrhythmia Database on CD-ROM and software for use with it," in *Proceedings of Computers in Cardiology*, pp. 185–188, September 1990.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 313479, 14 pages
doi:10.1155/2010/313479

Research Article

Reconfigurable Hardware Implementation of a Multivariate Polynomial Interpolation Algorithm

Rafael A. Arce-Nazario,[1] Edusmildo Orozco,[1] and Dorothy Bollman[2]

[1] *Department of Computer Science, University of Puerto Rico, Río Piedras, PR 00924, Puerto Rico*
[2] *Department of Mathematical Sciences, University of Puerto Rico, Mayagüez, PR 00681, Puerto Rico*

Correspondence should be addressed to Rafael A. Arce-Nazario, rafael.arce@upr.edu

Received 2 March 2010; Revised 26 July 2010; Accepted 26 October 2010

Academic Editor: Viktor K. Prasanna

Multivariate polynomial interpolation is a key computation in many areas of science and engineering and, in our case, is crucial for the solution of the reverse engineering of genetic networks modeled by finite fields. Faster implementations of such algorithms are needed to cope with the increasing quantity and complexity of genetic data. We present a new algorithm based on Lagrange interpolation for multivariate polynomials that not only identifies redundant variables in the data and generates polynomials containing only nonredundant variables, but also computes exclusively on a reduced data set. Implementation of this algorithm to FPGA led us to identify a systolic array-based architecture useful for performing three interpolation subtasks: Boolean cover, distinctness, and polynomial addition. We present a generalization of these tasks that simplifies their mapping to the systolic array, and control and storage considerations to guarantee correct results for input sequences longer than the array. The subtasks were modeled and implemented to FPGA using the proposed architecture, then used as building blocks to implement the rest of the algorithm. Speedups up to 172× and 67× were obtained for the subtasks and complete application, respectively, when compared to a software implementation, while achieving moderate resource utilization.

1. Introduction

Recent years have seen a significant increase in methods and tools to collect genetic data from which important information can be extracted using a number of techniques [1]. For instance, microarray data collected at various steps in organism development can help geneticists understand its developmental process and response to environmental stimuli [2]. Several models such as ordinary differential equation models [3], continuous models [4], stochastic models [5], and discrete models, of which Boolean models [6] are special cases, have been proposed. Essentially, each node represents the expression level of a gene (or genes) of interest at a time point. A directed edge from node a to node b symbolizes the influence of gene(s) in a to the gene(s) in b.

Our research group focuses on multivariate finite field gene network (MFFGN) models, in which multiple genes are monitored at each time step and their expression levels are discretized to a predefined set of values $\{0, 1, 2, \ldots, p-1\}$, where p is a prime number, that is, the expression level of each gene is an element of the prime field $\mathbb{Z}_p$ [7, 8]. One way to deduce gene interaction from these models is to solve the reverse engineering problem, that is, find functions that describe the network's state transitions. An important step in the solution of this problem is determining an interpolating polynomial for the points by using such methods as a multivariate version of Lagrange's interpolation formula [8]. Polynomial interpolation over finite fields also has applications in error correcting codes and is a major building block of many numerical methods [9, 10].

Fast algorithms and implementations are needed to sustain rapidly increasing bioinformatics computational demands [11]. Reconfigurable logic represents an attractive platform for the high performance implementation of finite field algorithms, with many designs achieving significant speedups over their software equivalents. However, despite the availability of tools to ease the hardware design flow, there is still a notable gap between the bioinformatics and

hardware experts [12]. Our group developed a reconfigurable logic implementation of a multivariate polynomial interpolation algorithm suitable for reverse engineering in genetic networks. Rather than striving for a specialized punctual design, we used this opportunity to identify and develop common structures that might be of use to other researchers and applications. Well-documented and parameterizable components that perform functionalities that are common in many algorithms will alleviate the effort spent on new implementations.

In this paper, we discuss the interpolation algorithm and our proposed architecture. We emphasize several computational substructures that appear repeatedly throughout the design and which can be implemented effectively using a hardware systolic array-based architecture. These tasks are of the type where we perform a certain reduction or rearrangement of a sequence of elements from multivariate polynomials or boolean expressions. The systolic array concurrently manages data receipt and parallel processing, making it an amenable structure when dealing with streamed data. The simplicity of the array cells, storage, and control unit, allows the instantiation of multiple cells while maintaining competitive clock frequencies, thus achieving high performance. Several tasks critical to interpolation were modeled and implemented to FPGA using the proposed architecture, obtaining speedups up to 172× when compared to a software implementation, while achieving low resource utilization. These implementations were used as components to develop a complete, high-performance multivariate polynomial interpolation methodology on hardware.

Paper Outline. Section 2 discusses the interpolation methodology, while Section 3 details data representation for the implementation. Section 4 describes the hardware implementation in general. Sections 5 and 6 describe an algorithmic generalization for the reduction tasks and present the proposed hardware architecture. Section 7 explains the rest of the implementation blocks. Section 8 reports the results of FPGA implementations, both of reduction tasks and the complete implementation. Section 9 provides our conclusions.

2. Interpolation Methodology

Discrete models, in particular finite field models, have been proposed for regulatory processes such as genetic networks [7] and biochemical networks [13]. In the setting of a finite field model a genetic network can be seen as a finite system whose states are governed by the iterations of a tuple of polynomials $(P_1, P_2, \ldots, P_n)$, where n is the number of genes.

Given a sequence of k n-tuples of values from a set F, representing the states of n genes at times t_0 through t_{k-1}, the reverse engineering problem seeks to find a function $f : F^n \to F^n$ for which $f(s_0) = s_1$, $f(s_1) = s_2, \ldots, f(s_{k-2}) = s_{k-1}$.

When F is a finite field, function f can be represented by a tuple of polynomials $(P_1, P_2, \ldots, P_n)$ where $P_i : F^n \to F$ for each $i = 1, 2, \ldots, n$. As pointed out in [8] such polynomials can be found by the following variation of Lagrange's interpolation formula:

$$P_i(x) = \sum_{m=0}^{k-1} f_i(a_m) \prod_{j=0, j\neq m}^{k-1} \frac{x_{l_{jm}} - a_{jl_{jm}}}{a_{ml_{jm}} - a_{jl_{jm}}}, \quad (1)$$

where each a_m denotes an n-tuple and a_{ml} denotes the lth component of a_m. Given a_j and a_m, where $a_j \neq a_m$, l_{jm} is the first component in which a_j and a_m differ.

In the context of genetic networks modeled by polynomials over a finite field, the resulting polynomials give a sense to the biologist about the interactions between genes. Nevertheless, results from (1) may contain *redundant* variables even after algebraic simplifications.

Table 1: Table for $f : \mathbb{Z}_3^3 \to \mathbb{Z}_3$.

x_1	x_2	x_3	f
0	1	2	0
1	1	2	0
1	2	0	1
0	2	1	2
1	1	0	2

Example 1. Let us consider the function $f : \mathbb{Z}_3^3 \to \mathbb{Z}_3$ which is incompletely specified by Table 1.

Using (1), an interpolating polynomial for f is given by $P = x_1^2x_2x_3 + x_1^2x_2^2 + x_1^2x_3 + 2x_1^2x_2 + x_1x_2 + 2x_2 + 2x_1^2 + 2x_1 + 1$. This polynomial depends on x_1,x_2, and x_3 and cannot be further simplified by any algebraic means. However, the polynomial $P' = 2x_2 + 2x_2x_3$ also interpolates f at the given points but depends only on variables x_2 and x_3. In this context, x_1 is a redundant variable, since we found another interpolating polynomial for f that does not include x_1 in its expression.

Redundant variables are undesirable as they introduce complexity into the polynomials without adding information valuable to the biologist. Furthermore, empirical data suggest that genetic networks are sparsely connected [14]. Redundant variables can be identified using Sasao's algorithm for the detection of dependent variables in incompletely specified multiple valued functions [15].

2.1. Essential Variables and Bases. For any set S and any positive integer n, we denote each $s \in S^n$ by $(s_1, s_2, \ldots, s_n)$. Let $U = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables. For any set of variables $X = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\} \subset U$ and any $s \in S^n$, we define $\text{proj}_X s = \{s_{i_1}, \ldots, s_{i_m}\}$. For instance, if $U = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $X = \{x_2, x_3, x_6\}$, and $s = (1, 0, 3, 2, 0, 1)$, the projection of s on X, $\text{proj}_X s$, is $\{0, 3, 1\}$.

Let $A = \{0, 1, \ldots, a-1\}$ and $B = \{0, 1, \ldots, b-1\}$ where a and b are integers, $a, b \geq 2$ and let $f : D \to B$, where $D \subset A^n$.

Table 2

x_1	x_2	x_3	x_4	x_5	f
0	2	1	2	3	1
0	2	1	3	1	1
2	1	1	0	1	1
0	1	1	2	0	2
2	1	2	1	0	2
2	1	2	0	1	2
2	1	2	2	1	2
0	1	2	1	1	3
2	2	2	2	2	3

We say that a variable x_i is *essential* in f or *depends on* f if there exist $c, d \in D$ such that $c_i \neq d_i$, $c_j = d_j$ for all $j \neq i$, and $f(c) \neq f(d)$. A set of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$ is a *basis* for f if (1) for every $c, d \in D$, $\text{proj}_X c = \text{proj}_X d$ if and only if $f(c) = f(d)$ and (2) there is no other set of variables that properly contains X with this property.

Lemma 1. *If x is an essential variable and X is a basis, then $x \in X$.*

Proof. Let x be an essential variable. Then there exists $c, d \in D$ and i such that $c_i \neq d_i$, $\text{proj}_{U-\{x_i\}} c = \text{proj}_{U-\{x_i\}} d$ and $f(c) \neq f(d)$. Let X be any basis. Then $X \not\subset U - \{x_i\}$, but $x \subset U$. Hence $x_i \in X$. □

For any function $f : D \to B$ and any basis X, f can be expressed solely in terms of the variables of X. For any basis X, any variable $x \notin X$ is *redundant* (with respect to X.)

Our goal is to determine interpolating polynomials over finite fields in terms of the variables of any of its bases. To this end, let us first review an algorithm of Sasao [15] which determines all bases for any multiple valued function, not just those over finite fields.

Let $f : D \to B$, where $D \subset A^n$. For each $i = 0, 1, \ldots, b-1$, let $S_{f,i} = \{s \in D \mid f(s) = i\}$.

The set of variables appearing in each disjunct of R constitutes a basis.

Thus, the algorithm consists of two stages. First, determine R, which requires $O(b^2)$ steps and second, convert R to disjunctive normal form, which requires $O(2^n)$ steps.

The following lemma, whose proof is immediate, is useful in simplifying expressions such as $x_1 \wedge x_1 x_2 = x_1$ and $x_1 \vee x_1 x_2 = x_1$ in the computation of R.

Lemma 2. *Let $S = \{x_{i_1}, x_{i_2}, \ldots, x_{i_d}\}$ be a subset of the variables $\{x_1, x_2, \ldots, x_n\}$, let E be the disjunction of the variables in S and let F be a disjunction of some subset of S. Also let G be the conjunction of some subset of S. Then*

(a) $E \wedge F = F \wedge E = F$,

(b) $E \wedge G = G \wedge E = G$.

In what follows we abbreviate $E \wedge F$ by EF.

```
for i := 0 to b − 1 do
    R = 1;
    for j := 0 to b − 1, j ≠ i do
        r(i, j) = ⋁_{(u,v)∈S_{f,i}×S_{f,j}} {x_k | u_k ≠ v_k}
    end
    R = R ∧ r(i, j);
    Express R in disjunctive normal form (DNF)
end
```

Algorithm 1: Algorithm to determine the collection of bases.

Example 2. Let $f : \{0, 1, 2, 3\}^5 \to \{1, 2, 3\}$ be a function whose values are included in Table 2.

Applying Algorithm 1 and the reduction rules of Lemma 2, we obtain

$$\begin{aligned} R &= (x_2 \vee x_5)(x_3)(x_1 \vee x_5)(x_1 \vee x_4) \\ &= x_1 x_2 x_3 \vee x_1 x_3 x_5 \vee x_3 x_4 x_5. \end{aligned} \tag{2}$$

Thus, the bases are $X_1 = \{x_1, x_2, x_3\}$, $X_2 = \{x_1, x_3, x_5\}$, and $X_3 = \{x_3, x_4, x_5\}$.

2.2. Interpolating Polynomials with No Redundant Variables. Given any partially defined function f over a finite field and any basis X for f, we would like to determine interpolating polynomials involving only those variables of X, that is, polynomials containing no redundant variables.

Definition 1. Let $f : D \to B$ where $D \subset A^n$ and let X be a basis for f. For all $c, d \in A^n$ define the relation $c \equiv_X d$ if and only if $c_i = d_i$, for each $x_i \in X$, $f(c)$, and $f(d)$ are defined, and $f(c) = f(d)$.

In other words, $c \equiv_X d$ if and only if $(\text{proj}_X c, f(c)) = (\text{proj}_X d, f(d))$. It is straightforward to verify the following.

Lemma 3. *$\equiv_X$ is an equivalence relation.*

Let $D/\equiv_X$ be a set of representatives of each equivalence class under $\equiv_X$.

Theorem 1. *Let F be a field, let (a_0, b_0), (a_1, b_1), …, (a_{k-1}, b_{k-1}) be a set of points in $F^n \times F$, and let X be a basis for the function f defined by $f(a_i) = b_i$, $i = 0, 1, \ldots, k-1$. Then an interpolating polynomial for f whose only variables are those of X is*

$$P(X) = \sum_{m=0,\ a'_m \in D/\equiv_X}^{k-1} b_m \prod_{j \neq m} \left(\frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} \right), \tag{3}$$

where ℓ_{jm} is the smallest index such that $x_{\ell_{jm}} \in X$ and a'_j and a'_m differ.

Proof. For each a_m, $m = 0, 1, \ldots, k-1$, let a'_m be the representative of the equivalence class C_m containing a_m. Then

$$\frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} = \begin{cases} 0 & \text{if } x = a_j \\ 1 & \text{if } x = a_m \end{cases} \tag{4}$$

and so $P(a_m) = b_m$, where b_m is the value of each tuple of C_m. □

Note that for each of the factors

$$c_{jm}(x) = \frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} \tag{5}$$

in (3), $c_{jm}(a_j) = 0$ and $c_{jm}(a_m) = 1$ and so for all $i = 0, 1, \ldots, k-1$ and all $r = 1, 2, \ldots$

$$c^r_{jm}(a_i) = c_{jm}(a_i). \tag{6}$$

Equation (3) gives us a new algorithm for polynomial interpolation that involves only those variables in the chosen basis X, and furthermore, it also avoids redundant tuples by computing only on the representatives of the equivalence classes given by the relation $\equiv_X$.

Using (3), $O(k^2)$ operations are needed to interpolate k points for each function $f : F^n \to F$. Given k points of a function $f : F^n \to F^n$, the bases for the component functions f_i need not be the same. In this case, we can apply (3) n times, thus giving an algorithm with complexity $O(nk^2)$.

Example 3. Let us assume that the 0, 1, 2, 3 of Example 2 represent the corresponding elements of $\mathbb{Z}_5$. We found that the bases are $X_1 = \{x_1, x_2, x_3\}$, $X_2 = \{x_1, x_3, x_5\}$, and $X_3 = \{x_3, x_4, x_5\}$. Now let us use the new algorithm given by (3) to find an interpolating polynomial over $\mathbb{Z}_5$ in terms of the basis X_1. The equivalence classes under X_1 are

$$\begin{aligned} C_1 &= \{(0,2,1,2,3), (0,2,1,3,1)\}, \\ C_2 &= \{(2,1,1,0,1)\}, \\ C_3 &= \{(0,1,1,2,0)\}, \\ C_4 &= \{(2,1,2,1,0), (2,1,2,0,1), (2,1,2,2,1)\}, \\ C_5 &= \{(0,1,2,1,1)\}, \\ C_6 &= \{(2,2,2,2,2)\}. \end{aligned} \tag{7}$$

and the set of class representatives is

$$\begin{aligned} \frac{D}{\equiv_{X_1}} = \{&(0,2,1,2,3), (2,1,1,0,1), (0,1,1,2,0), \\ &(2,1,2,1,0), (0,1,2,1,1), (2,2,2,2,2)\}. \end{aligned} \tag{8}$$

Applying (3) and making use of (6), we have

$$\begin{aligned} P(X_1) = {} & 1 * \frac{x_1 - 2}{0 - 2}\frac{x_2 - 1}{2 - 1} \\ & + 1 * \frac{x_1 - 0}{2 - 0}\frac{x_3 - 2}{1 - 2}\frac{x_2 - 2}{1 - 2} \\ & + 2 * \frac{x_2 - 2}{1 - 2}\frac{x_1 - 2}{0 - 2}\frac{x_3 - 2}{1 - 2} \\ & + 2 * \frac{x_1 - 0}{2 - 0}\frac{x_3 - 1}{2 - 1}\frac{x_2 - 2}{1 - 2} \\ & + 3 * \frac{x_2 - 2}{1 - 2}\frac{x_1 - 2}{0 - 2}\frac{x_3 - 1}{2 - 1} \\ & + 3 * \frac{x_1 - 0}{2 - 0}\frac{x_2 - 1}{2 - 1} \\ = {} & 2(x_1 - 2)(x_2 - 1) + 3x_1(x_2 - 2)(x_3 - 2) \\ & + 4(x_1 - 2)(x_2 - 2)(x_3 - 2) \\ & + 4x_1(x_3 - 1)(x_2 - 2) \\ & + 4(x_1 - 2)(x_2 - 2)(x_3 - 1) + 4x_1(x_2 - 1). \end{aligned} \tag{9}$$

Thus, a polynomial which depends only on the variables x_1, x_2, and x_3 and interpolates the tuples given by Example 2 is

$$P(X_1) = 4x_1x_2 + 4x_2x_3 + 3x_1 + 2x_3 + 1. \tag{10}$$

3. Data Representation

As can be deduced from the previous section, an implementation of the multivariate polynomial interpolation algorithm must be able to represent and compute on disjunctive normal form (DNF), conjunctive normal form (CNF), and multivariate polynomial expressions. The following subsections establish our method of representation, which ultimately determines the techniques used for implementing the algorithms.

3.1. DNF and CNF Expressions. A Boolean expression D in DNF using only uncomplemented variables is a disjunction of conjunctive clauses $D = \bigvee_{i=0}^{q-1} c_i$, where $c_i = x_0^{t_{i_0}} \wedge x_1^{t_{i_1}} \wedge \cdots \wedge x_{m-1}^{t_{i_{m-1}}}$, $t_{i_j} \in \{0, 1\}$. We shall refer to each c_i term as a *cube*. For example, the following is a Boolean expression in DNF: $x_1x_2 \vee x_1x_3 \vee x_0$. In our scheme, a DNF expression is represented as a set of cubes $\{c_0, c_1, \ldots, c_{q-1}\}$ where each c_i is represented as a sequence $\langle t_{i_0}, t_{i_1}, \ldots, t_{i_{m-1}} \rangle$. Observe that complemented variables are not used in the algorithm for finding essential variables and bases, hence their representation is not necessary in this context.

Example 4. The DNF expression $D(x_0, x_1, x_2, x_3) = x_1x_2 \vee x_1x_3 \vee x_0$ is represented as $\{\langle 0, 1, 1, 0\rangle, \langle 0, 1, 0, 1\rangle, \langle 1, 0, 0, 0\rangle\}$

A Boolean expression C in CNF using only uncomplemented variables is represented as a set of disjunctions of literals (DOL) $\{d_0, d_1, \ldots, d_{q-1}\}$ where each DOL d_i

is represented as a sequence $\langle t_{i_0}, t_{i_1}, \ldots, t_{i_{m-1}} \rangle$. Within our implementation there is no need to distinguish the CNF from the DNF representation since no individual component needs to operate on both simultaneously.

Example 5. The CNF expression $C(x_0, x_1, x_2) = (x_0 \vee x_2) \wedge (x_0 \vee x_1) \wedge (x_1)$ is represented as $\{\langle 1, 0, 1\rangle, \langle 1, 1, 0\rangle, \langle 0, 1, 0\rangle\}$

3.2. Polynomial Representation. A multivariate polynomial P over $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime is defined as $P(x_0, x_1, \ldots, x_{n-1}) = \sum_{i=0}^{q-1} m_i$, where $m_i = \alpha_i x_0^{t_{i_0}} x_1^{t_{i_1}} \cdots x_{n-1}^{t_{i_{n-1}}}$ and $\alpha_i, t_{i_j} \in \mathbb{Z}_p$. For example, one multivariate polynomial over $\mathbb{Z}_5$ is $P(x_0, x_1, x_2) = 4 + x_0^3 x_1 + 3x_2 + x_2^4$. In our scheme, a polynomial is represented as a set of monomials $\{m_0, m_1, \ldots, m_{q-1}\}$ where each m_i is represented as a tuple $(\alpha_i, \langle t_{i_0}, t_{i_1}, \ldots, t_{i_{n-1}}\rangle)$. Binary representation requires $\lceil \log_2 p \rceil$ bits for each α_i and t_{i_j}.

Example 6. The multivariate polynomial $P(x_0, x_1, x_2) = 4 + x_0^3 x_1 + 3x_2 + x_2^4$ over $\mathbb{Z}_5$ is represented as $\{(4, \langle 0, 0, 0\rangle), (1, \langle 3, 1, 0\rangle), (3, \langle 0, 0, 1\rangle), (1, \langle 0, 0, 4\rangle)\}$.
For the three variable polynomial over $\mathbb{Z}_5$ each monomial requires $4 \times \lceil \log_2 5 \rceil = 12$ bits for binary representation.

4. General Description of Implementation

Our algorithm for determining interpolating polynomials that do not contain any redundant variables consists of two stages. The first stage identifies the dependent variables using Sasao's algorithm, while the second uses this information along with (3) to determine P_i. Thus, a key part of the reconfigurable logic implementation is choosing efficient hardware structures to perform the critical parts of both stages. For the first stage, the most time-consuming task is the conversion of a Boolean expression in conjunctive normal form (CNF) to disjunctive normal form (DNF). The second step relies heavily on the identification of distinct binomials as they are generated from (3) and multivariate polynomial multiplication/addition.

To take advantage of the FPGA's fine-grained parallelism and considering their limitations in I/O bandwidth, these computations were implemented in a pipelined manner. In other words, computational blocks were designed to sustain stream processing as allowed by the FPGA's resources, rather than accumulate all needed data and then perform block processing.

Figure 1 shows a block diagram of the implementation. The *CNF Generator* receives the n-tuples and performs Algorithm 1 to generate the disjuntions of literals, that is, the $r(i, j)$ terms. These are streamed to the *Cover DOLs* block which eliminates redundant disjuntions of literals by using Lemma 2. The nonredundant DOLs are then fed to block *CNF2DNF* which computes the conjunctions of literals and uses Lemma 2 to eliminate redundant conjunctions. The output of *CNF2DNF* is the set of bases. A priori biological knowledge about the organism under study is required to choose the most adequate base. The n-tuples are also fed to the second stage which uses the chosen base to determine representatives of the equivalence classes from the n-tuples. The representatives are then streamed to the modified *Lagrange* interpolation block which implements (3) by generating the binomials for each pair of class representatives (*Binomial Generator*), eliminates duplicate binomials (*Filter Duplicates*), then multiplies and adds the product results.

The highlighted blocks in Figure 1, that is, *Cover DOLs*, *Cover COLs*, *Determine Class Representatives*, and *Polynomial Addition*, have two characteristics that make them suitable candidates for systolic array processing: (1) each performs an operation on a sequence of elements that are streamed from the preceeding block, (2) the operation is such that it performs reduction of a sequence of elements from multivariate polynomials or Boolean expressions. For example, the *Cover Cubes* receives preliminary results from the CNF to DNF conversion and eliminates redundant cubes using Lemma 2. This is an operation that can be described as given a sequence C of cubes $(C = c_1, c_2, \ldots, c_m)$ determine a subsequence $C' \subset C$ such that it contains only cubes $c_i \in C$ where there exists no other $c_j \neq c_i \in C$ such that $c_i \wedge c_j = c_j$.

The following sections describe a generalized algorithm and systolic array-based architecture for the type of reduction operations common throughout our interpolation methodology. This is followed by a description of how they are utilized as part of the architectural data path.

5. Generalization of Reduction Tasks

The subtasks that deal with reduction in our interpolation algorithm can be generally described as follows. Given the sequence $X = x_1, x_2, \ldots, x_m$ of elements from Σ, compute the sequence $Y = y_1, y_2, \ldots, y_m$ where $y_i \in \Sigma_\varepsilon = \Sigma \cup \varepsilon$ by eliminating or combining elements based on binary comparisons between them. The empty element (ε) is introduced for the representation of operations where groups of elements can be reduced to a single element. Algorithm 2 shows a general form of these problems, where W and S are binary functions of the form $\Sigma_\varepsilon \times \Sigma_\varepsilon \rightarrow \Sigma_\varepsilon$, and the symbol $\|$ denotes parallel computation.

The tasks of distinctness, polynomial addition, and redundant Boolean term elimination (hereon referred to as *cover*) can be implemented by specifying W and S, as explained in the following subsections.

5.1. Distinctness. The task of identifying the distinct elements of a sequence $X = x_1, x_2, \ldots, x_m$ is needed within our interpolation algorithm to determine class representatives and filter out binomial duplicates. It can be implemented by Algorithm 2 with

$$W\left(x_i, x_j\right) := \begin{cases} x_j & \text{if } x_i = \varepsilon \\ x_i, & \text{otherwise,} \end{cases}$$
$$S\left(x_i, x_j\right) := \begin{cases} \varepsilon & \text{if } x_i = x_j \quad \text{or} \quad x_i = \varepsilon \\ x_j, & \text{otherwise.} \end{cases} \tag{11}$$

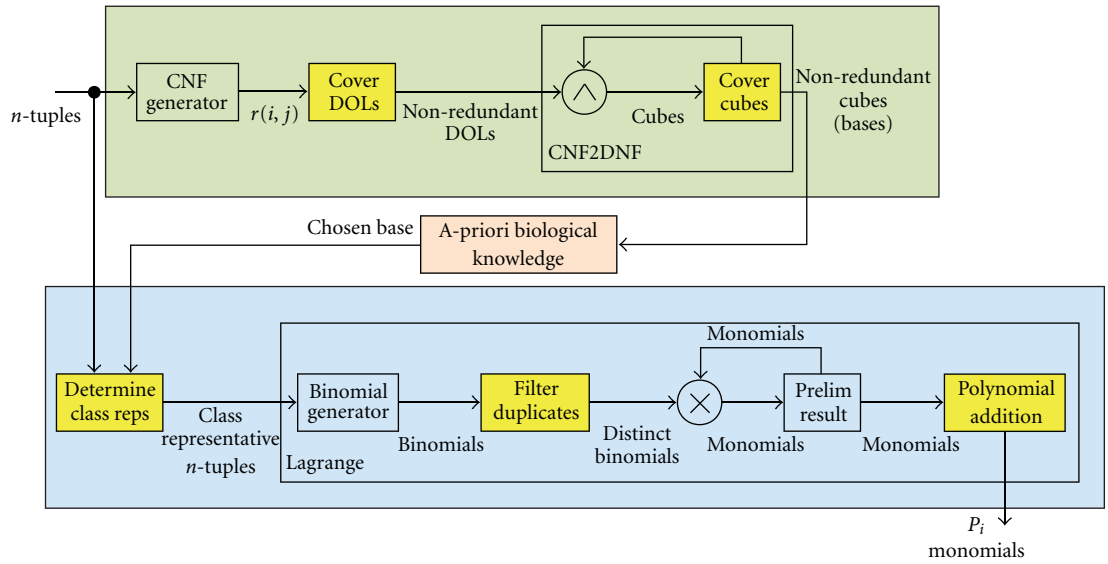


Figure 1: Block diagram of the implementation, highlighting the subtasks that are implemented using the systolic array architecture.

```
for i := 1 to n − 1 do
    for j := i + 1 to n do
        x_i := W(x_i, x_j) ‖ x_j := S(x_i, x_j);
    end
end
```

Algorithm 2: Nested loop algorithm for reduction tasks.

Example 7. Assume $X = 5, 8, 8, 5, 8, 1$. After computation with Algorithm 2 using (11), the result is $X' = 5, 8, 1, \varepsilon, \varepsilon, \varepsilon, \varepsilon$.

We provide a correctness proof for this operation. Proofs of the other reduction operations given in this section are similar.

Lemma 4. *For an input sequence $x = x_1, x_2, \ldots, x_m$, where $x_i \in \Sigma$, the output of Algorithm 2 with functions W and S defined as in (11) outputs $x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \varepsilon, \varepsilon, \ldots, \varepsilon$ where $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is the subsequence of distinct elements of the input sequence.*

Proof. We show by induction that after $j \leq k$ iterations of the outer loop,

$$x_i, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_j}, y_{j+1}, \ldots, y_m, \tag{12}$$

where for each $p \geq j+1, y_p \in \{x_{i_{j+1}}, \ldots, x_{i_k}\} \cup \{\varepsilon\}$.

After $j = 1$ iterations, $x_1 = x_{i_1}$. Next suppose that after j iterations,

$$x_1, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, \ldots, x_{i_j}, y_{j+1}, \ldots, y_m. \tag{13}$$

for some $j < k$ where $y_p \in \{x_{i_{j+1}}, \ldots, x_{i_k}\} \cup \{\varepsilon\}$ for each $p \geq j+1$. Then either (1) $y_{j+1} = x_{j+1}$ or (2) $y_{j+1} = \varepsilon$. In the first case, $y_{j+1} = x_{i_{j+1}}$ since x_{j+1} is not equal to any $x_p, p \leq j$ and is not changed by the $j+1$st iteration. In the second case, $y_{j+1} = \varepsilon$ will be replaced by $x_{i_{j+1}}$ during the $j+1$st iteration. Hence in either case,

$$x_1, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_{j+1}}, y_{j+2}, \ldots, y_m, \tag{14}$$

where for each $p \geq j+2, y_p$ is either ε or an element x_{i_p}, where $p \geq j+2$.

Hence after $i = k$ iterations,

$$x_1, x_2, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \varepsilon, \ldots, \varepsilon \tag{15}$$

and remains the same after $i > k$ iterations since $W(\varepsilon, \varepsilon) = S(\varepsilon, \varepsilon) = \varepsilon$. □

5.2. Polynomial Addition. Assume that each of the elements in $X = x_1, x_2, \ldots, x_m$ is a monomial. A polynomial addition operation can be implemented by Algorithm 2 with

$$W\left(x_i, x_j\right) := \begin{cases} x_i + x_j & \text{if } \operatorname{mon}(x_i) = \operatorname{mon}\left(x_j\right), \\ & \text{or } \quad x_i = \varepsilon, \\ x_i, & \text{otherwise}, \end{cases}$$
$$S\left(x_i, x_j\right) := \begin{cases} \varepsilon & \text{if } \operatorname{mon}(x_i) = \operatorname{mon}\left(x_j\right), \\ & \text{or } \quad x_i = \varepsilon, \\ x_j, & \text{otherwise}, \end{cases} \tag{16}$$

where $\operatorname{mon}(x_i) = \langle t_{i_0}, t_{i_1}, \ldots, t_{i_{n-1}} \rangle$, that is, the monomial without the coefficient.

Example 8. Assume $X = x^2, xy^2, 3x^2, xy, 2xy$. After computation with Algorithm 2 using (16), the result is $X' = 4x^2, xy^2, 3xy, \varepsilon, \varepsilon$.

5.3. Cover. Assume that each of the elements in X is a cube. A cube c_i covers another cube c_j (represented as $c_i \supset c_j$) if all Boolean variables present in c_i are present in c_j (e.g., $x_2 \supset x_1x_2x_3$, but $x_2x_4 \not\supset x_1x_2x_3$). The cover operation can be implemented by Algorithm 2 with

$$W(x_i, x_j) := \begin{cases} x_j & \text{if } x_j \supset x_i, \\ x_i, & \text{otherwise,} \end{cases}$$
$$S(x_i, x_j) := \begin{cases} \varepsilon & \text{if } x_i \supset x_j, \\ x_j, & \text{otherwise,} \end{cases} \tag{17}$$

where $q_i \supset \varepsilon$ for any q_i.

Example 9. Assume $X = x_1x_2, x_2x_4, x_1, x_1x_2x_4$. After computation with Algorithm 2 using (17), the result is $X' = x_1, x_2x_4, x_1, \varepsilon$. Notice that there may be duplicate terms in the result but these can be easily eliminated using a distinctness operation.

Algorithm 2 is also amenable to other problems that can be expressed as the result of binary comparisons between every two elements of a sequence. For example, the problem of sorting a sequence S can be implemented with

$$W(x_i, x_j) := \min(x_i, x_j),$$
$$S(x_i, x_j) := \max(x_i, x_j). \tag{18}$$

6. Hardware Structure for Reduction Tasks

This section explains our proposed hardware structure and outlines the mapping for the reduction tasks. We first discuss the systolic array and cells with the assumption that the array is deep enough to process the input sequence without overflowing. We proceed by discussing the additional components that must be added to guarantee correct processing even if the overflow condition does not hold.

6.1. Systolic Array and Cells. The proposed structure is a linear systolic array of n computational cells, each performing an operation deduced from W and S of the above discussion. Figure 2 illustrates the array and contents of the basic cell. Each cell contains two registers F and M, where each has enough resources to hold any single element from the alphabet Σ_ε. Each cell also contains the logic to perform functions $W(F, M)$ and $S(F, M)$. All cells are initialized to the empty symbol. At each clock step a new element from the sequence X is input to the first cell and every cell performs a comparison between its stored value and the value from the previous cell and assigns (possibly) new values to the registers according to W and S. After all elements have been processed the M registers contain the resulting sequence and can be shifted out of the array using the connections between the M registers.

More formally, our linear systolic array computation can be modeled as a simplified version of the generic systolic array presented in [16], as shown in Algorithm 3. The notation $F_i[r]$ symbolizes the content of register F in cell r at iteration i.

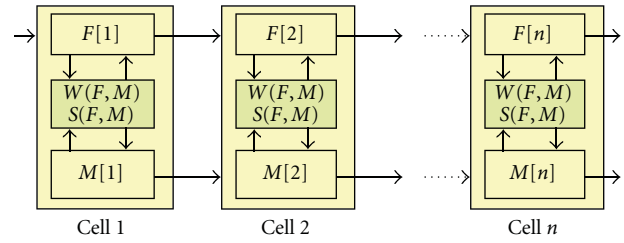


Figure 2: Block diagram of systolic array.

$i := 0$
$\|_{r=1}^{n} \begin{cases} M_0[r] := \varepsilon \\ F_0[r] := \varepsilon \end{cases}$
while stop_condition = false **do**
  $i := i + 1$
  if $i < m$ **then** $F_{i-1}[0] := x_i$
  else $F_{i-1}[0] := \varepsilon$
  end if
  $\|_{r=1}^{n} \begin{cases} F_i[r] := S(F_{i-1}[r-1], M_{i-1}[r]) \\ M_i[r] := W(F_{i-1}[r-1], M_{i-1}[r]) \end{cases}$
end

Algorithm 3: Model of the linear systolic array computation.

Table 3: Illustration of index generation by Algorithm 4 for the example shown in Figure 3.

$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$	$j = 8$	$j = 9$
x_0, x_1	x_0, x_2	x_0, x_3	x_0, x_4	x_0, x_5	x_1, x_5	x_2, x_5	x_3, x_5	x_4, x_5
		x_1, x_2	x_1, x_3	x_1, x_4	x_2, x_4	x_3, x_4		
				x_2, x_3				

Figure 3 illustrates the operation of the array using cells that implement the distinctness task by defining W and S as in (11). Observe that the operations implied by W and S and the connections between neighboring cells accomplish the comparisons established in Algorithm 2, albeit in a different order and in a parallel manner. In fact, the computation implemented by the systolic array can be interpreted as a reindexing of Algorithm 2 as presented in Algorithm 4. Table 3 shows the comparison indexes generated by Algorithm 4 for the example in Figure 3. Notice that even though Algorithm 4 has reordered the indexes, the same data dependence is maintained as in Algorithm 2. In other words, if $f(x_i, x_j)$ is performed before a $f(x_i, x_k)$ (where $j \neq k$) in Algorithm 2 the same order is preserved in Algorithm 4. The same condition holds for the order of $f(x_i, x_j)$ and $f(x_k, x_j)$.

6.2. Processing Sequences Longer than the Array Depth. The functionality of the basic cells mandates their implementation on user logic inside the FPGA (rather than on embedded

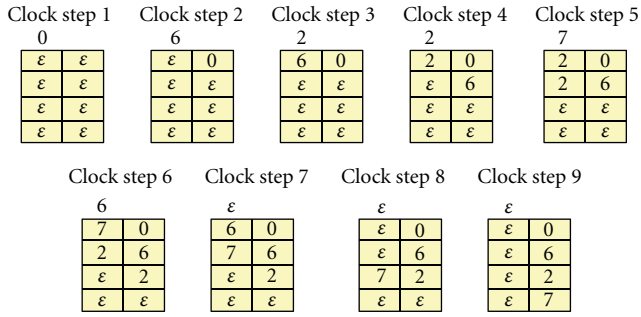


Figure 3: Systolic array implementing the distinctness operation for $X = 0, 6, 2, 2, 7, 6$. Each row represents a cell. Within each cell, the left and right columns represent the S and M registers, respectively. Elements are input to the top row and flow downwards.

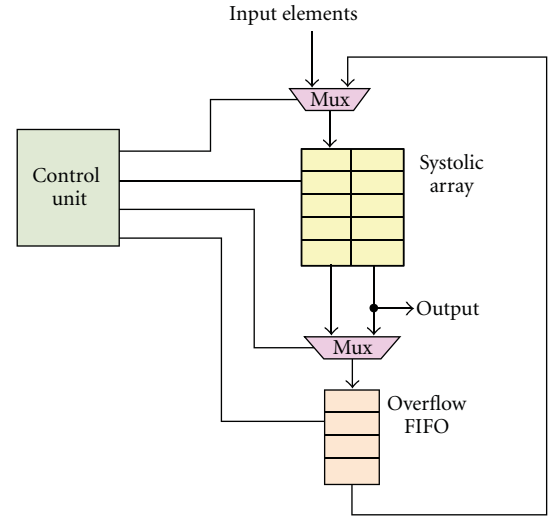


Figure 4: Pipeline with added overflow FIFO and control unit to support sequences longer than n.

for $j := 1$ to $2n - 3$ **do**
  parallel for $i :=$ iif $(j \leq n, 1, j - n)$ to $|(j)/2|$ **do**
    $x_i = W(x_i, x_{j-i+1}) \parallel x_{j-i+1} = S(x_i, x_{j-i+1})$
  end
End

Algorithm 4: Reindexed algorithm for reduction tasks.

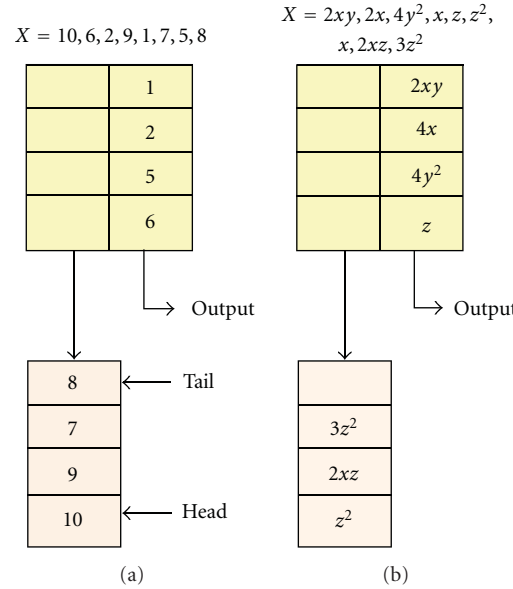


Figure 5: Contents of the systolic array and OF after a first pass of the sequences through (a) sorting and (b) polynomial addition.

units, such as block RAMS). Depending on the application, the characteristics of the data sequence, and FPGA model, the FPGA might not have enough resources to instantiate a systolic array whose depth d_{sa} is greater than or equal to m (the sequence length). In such cases, there is no guarantee that the array by itself will be able to compare each element against each other to obtain the correct solution. To illustrate this anomaly, assume that in the example presented in Figure 3 the input is a sequence of *distinct* elements $X = x_0, x_1, \ldots, x_5$. The array would be filled once x_3 is registered in the last cell, hence none of the cells would have operated on the combination of x_4 and x_5. A control and temporary storage mechanism must be provided to reiterate the data through the pipeline if needed and guarantee that all elements are compared against each other. This can be accomplished by a scheme such as the one shown in Figure 4 through the addition of a FIFO, control unit and several datapath control elements (i.e., multiplexers).

The *Overflow FIFO* (OF) stores elements that make it through the array without having been eliminated or registered. These elements will have to be reiterated through the array to test their relation to other elements in the OF. For example, suppose we have an array of depth $d_{sa} = 4$ that implements the distinctness operation. The sequence $\langle 3, 5, 1, 3, 2, 6, 7, 6 \rangle$ will fill the pipeline with $\langle 3, 5, 1, 2 \rangle$ so that the final $\langle 6, 7, 6 \rangle$ overflows the pipeline. These 3 elements are stored in the OF for a later pass through the array. The FIFO depth $d_f \geq m - d_{sa}$ where m is the number of elements in the sequence and d_{sa} is the array depth. Although both systolic array cells and FIFO spend mostly on register resources, a reason why one may be able to increase FIFO depth and not necessarily array depth is that in FPGAs FIFOs are easily mapped to the embedded block RAM, whereas processing cells map to user logic.

The *control unit* controls the data flow between the array and the OF and determines how to reiterate the overflowed data through the array. For the reduction tasks presented in Section 5 we can deduce three basic modes of operation for the control unit. Figures 5 and 6 illustrate the need for these modes. Figure 5 shows the end of a first pass of a sequence through systolic arrays that implement (a) a sorting algorithm, and (b) sum of polynomials.

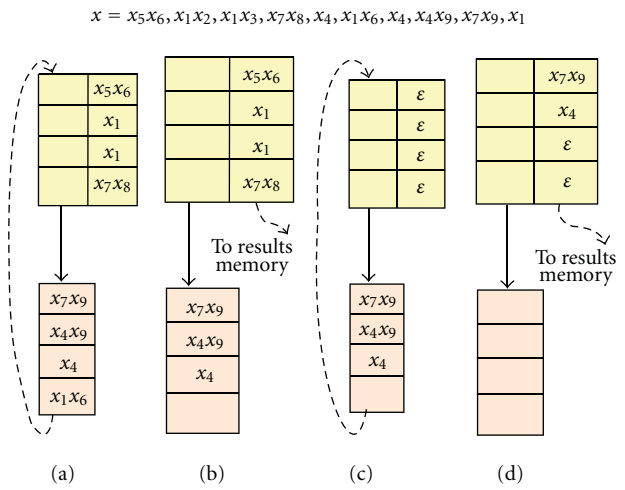


Figure 6: Contents of systolic array and overflow FIFO for the shown sequence X (a) after first pass, (b) after second pass (OF to array refeed), (c) systolic array is emptied for new OF refeed, (d) after final refeed from OF. Dotted lines illustrate the paths of data refeed.

(i) In the case of a nonreducing operation like sorting (Figure 5(a)), the array will contain d_{sa} sorted elements. Thus, elements in the array can be shifted out while elements in the OF are reinputted into the array. This must be repeated until the OF is empty after a pass, that is, at most $\lceil m/d_{sa} \rceil$ times.

(ii) When adding polynomials (Figure 5(b)), the array will contain at most d_{sa} monomials with their final coefficients. Thus, elements in the array can be shifted out while elements in the OF are reinputted into the array. This must be repeated until the OF is empty after a pass, that is, at most $\lceil m/d_{sa} \rceil$ times.

Figure 6 illustrates the various passes needed for the cover operation. At the end of a first pass the array will contain at most d_{sa} cubes. However, in this case a first pass through the array does not guarantee that all cubes in the OF have been compared to all the cubes registered in the array. For example, in Figure 6(a) cube x_1x_6 passed through the array without being registered. Meanwhile, the last cube x_1 covered x_1x_3 and x_1x_2 in two cells of the array. Thus, we have at least one cube (x_1x_6) in the OF which was not compared to a cube in the array (x_1). The elements in the OF must be refed through the (unflushed) array to allow every c_i stored in the FIFO to be compared against every c_j registered in the array. The refeed is repeated until a complete pass of the OF cubes through the array does not modify the cubes registered in the array. After this (Figure 6(b)), if any c_i in the OF is going to be part of the result, for example, x_4, it must only be still compared to other elements in the OF. The contents of the array may be shifted out to a memory that stores the results, the array is reset and the OF cubes are refed (Figure 6(c)). The three-step process continues until the OF is empty after an OF refeed.

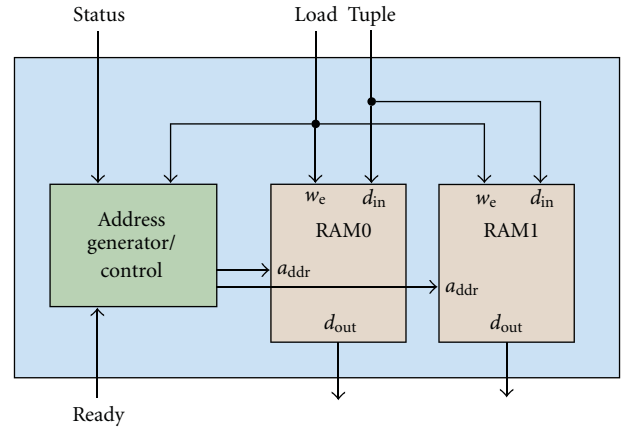


Figure 7: Element pair generator.

7. Further Implementation Considerations

Aside from the reduction tasks, the computational blocks in our implementation can be classified as performing one of two functionalities: element pair generation or multiple term multiplication. Element pair generators, such as the *CNF Generator* and the *Binomial Generator* in Figure 1, input a set of tuples and output a distinct pair of tuples at each computational step. This is accomplished by using the architecture illustrated in Figure 7. As tuples are input (each accompanied by a load signal) both RAMs store the tuples and an Address Control Generation unit counts the number of tuples. When a status signal indicates that all tuples have been received, the ACG begins generating all possible address combinations (a_0, a_1) for $0 \le a_0 < n$, $a_0 + 1 \le a_1 \le n$. The distinct pairs of n-tuples produced by the *CNF Generator* are fed to $n \lceil \log_2 p \rceil$-comparators to compute the $r(i, j)$ terms in Algorithm 1, as shown Figure 8. In the case of the *Binomial Generator*, the corresponding terms within the distinct pairs of n-tuples are subtracted from each other and the results are used to determine the terms of the binomial for (3), as shown in Figure 9.

Multiple-term multiplication/conjunction, such as needed in blocks *CNF2DNF* and *Lagrange*, is performed using the architectures shown in Figures 10 and 11, respectively. The architecture depicted in Figure 10 receives DOLs from the *CNF Generator*, the *DOL2Literals* block generates a literal expression for each found in the DOL. The literals are queued in a circular FIFO and are conjuncted to each of the results in the *Preliminary Results FIFO*. After each DOL is conjuncted, the resulting cubes are passed through the *Cover Cubes* block which eliminates redundant cubes. A control unit monitors and generates signals to coordinate the data path activities.

The architecture depicted in Figure 11 performs the computations of (3). Each binomial that is received from the *Binomial Generator* is registered. Then each of its two monomials is multiplied by each of the monomials that has resulted from previous binomial multiplications. The monomials that result from the multiplication are input to the *PrelimPolyAdd* block which adds similar monomials to

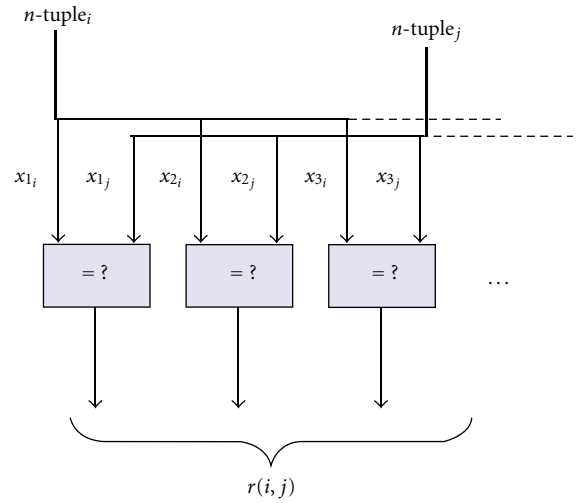


Figure 8: Computation of the disjunction of literals $r(i, j)$ from a pair of n-tuples.

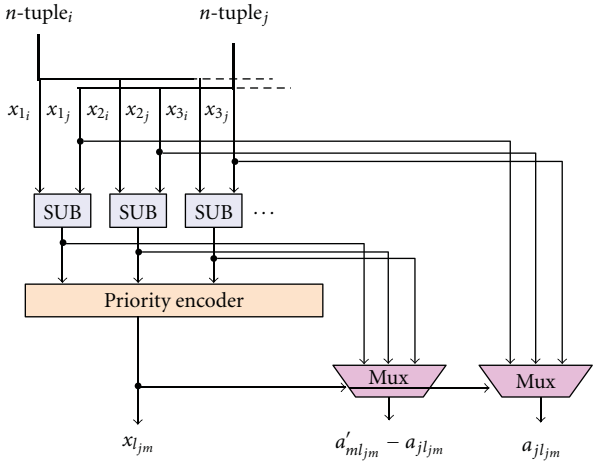


Figure 9: Computation of the terms of a binomial for (3) from a pair of n-tuples.

reduce the size of the preliminary product. *PrelimPolyAdd* outputs its result to the *PrelimRes FIFO*. The iterative process continues until the last binomial of a product is received, in which case each monomial of the preliminary product is output to the final *Polynomial Addition* block (see Figure 1). Once the last binomial of the last product has been multiplied, the (final) *Polynomial Addition* block is signaled and begins to output the monomials of the final result.

8. Results and Discussion

This section presents and discusses results for the individual reduction tasks as well as the complete interpolation algorithm implementation.

8.1. Reduction Tasks. The systolic array cells for the subtasks of distinctness, polynomial addition, and Boolean cover were modeled using Verilog HDL, based on their function

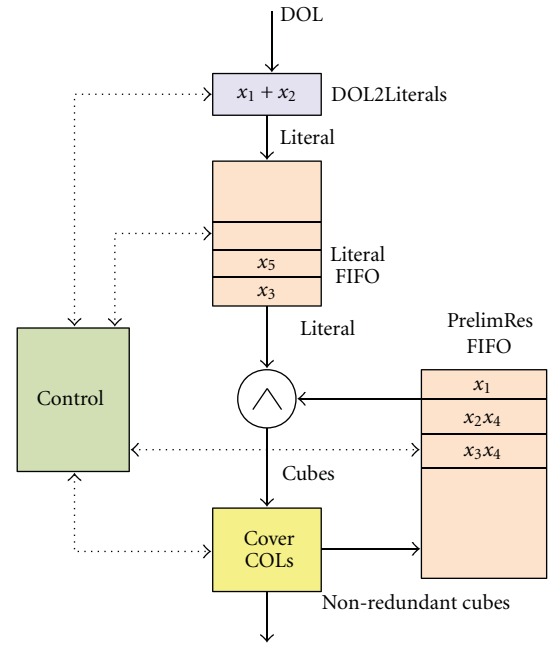


Figure 10: Illustration of a step in the conversion of CNF $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_3 \vee x_5) \wedge (x_1 \vee x_2)$ to DNF. The first two conjunctions have been computed and their redundant terms eliminated and stored in the Preliminary Results FIFO.

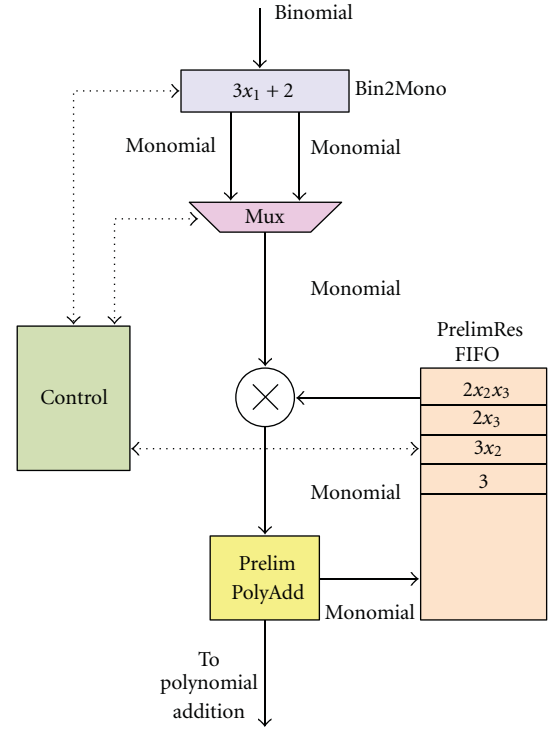


Figure 11: Illustration of a step in the multiplication of binomials $(x_2 + 1)(2x_3 + 3)(3x_1 + 2)$ in $\mathbb{Z}_5$. The multiplication of the first two binomials has completed and the results are stored in the *Preliminary Results FIFO*.

Table 4: Experimental results for subtask blocks.

	d_{sa}	Slices*	Slice F/F*	4 input LUTs*	FIFO16/RAMB16*	freq (Mhz)	t_F (s)	t_C (s)	speedup
3∗Cover	64	4141 (4.65%)	4348 (2.44%)	7708 (4.33%)	9 (2.68%)	176	$1.12E-03$	$1.29E-02$	11.51×
	128	8279 (9.29%)	8700 (4.88%)	15054 (8.45%)	9 (2.68%)	176	$5.90E-04$	$1.29E-02$	21.93×
	256	16554 (18.58%)	17404 (9.77%)	30796 (17.28%)	9 (2.68%)	176	$3.21E-04$	$1.29E-02$	40.33×
3∗P-add	64	2967 (3.33%)	3516 (1.97%)	4119 (2.31%)	7 (2.08%)	200	$6.84E-04$	$3.36E-02$	49.17×
	128	5885 (6.61%)	7455 (4.18%)	7156 (4.02%)	7 (2.08%)	200	$3.58E-04$	$3.36E-02$	93.99×
	256	11741 (13.18%)	14880 (8.35%)	14228 (7.99%)	7 (2.08%)	200	$1.95E-04$	$3.36E-02$	172.28×
3∗Distinct	64	2680 (3.01%)	3741 (2.10%)	3173 (1.78%)	7 (2.08%)	190	$7.20E-04$	$3.37E-02$	46.87×
	128	5342 (6.00%)	7465 (4.19%)	6261 (3.51%)	7 (2.08%)	190	$3.77E-04$	$3.37E-02$	89.60×
	256	10659 (11.96%)	14903 (8.36%)	12437 (6.98%)	7 (2.08%)	190	$2.05E-04$	$3.37E-02$	164.25×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

definitions presented in Section 5. The array and control units were also modeled using Verilog based on the descriptions provided in Section 6. Cell, systolic array and control unit models were designed as parameterizable modules in terms of width of elements, array depth and control mode, respectively. Models were behaviorally simulated using ModelSim SE 6.0 to validate their results against software versions of the algorithms. Xilinx ISE Design Suite 11.1 was used to synthesize the architectures for a Virtex-4 XC4VLX200ff1513-11 in the DRC Dev Systems DS2000. Default synthesis parameters were used, such as mapping FIFOs to block RAMs and speed as the optimization goal. All the reported results for the individual tasks are before Place and Route.

For each of the three subtasks, several systolic array depths (d_{sa}) were implemented, then randomly generated sequences of various bit widths and lengths were input. Table 2 shows timing and resource results for (1) the cover operation on sequences of length 4096 and width of 32 bits, (2) the polynomial addition operation on sequences of length 4096, where each element represents a monomial of 8 variables in $\mathbb{Z}_5$, and (3) the distinctness operation on the same sequences as (3). Results are compared against a software implementation compiled using the GNU project c++ compiler with optimization level 3 and running on a 3.40 GHz Intel Pentium D CPU with 2 MB cache and 3 GB RAM. CPU execution times were measured by accessing the processor's cycle counter. In all cases, we report the smallest measured CPU execution time. The *freq* column indicates the FPGA clock frequency, t_F and t_C are the FPGA and CPU run times, and speedup is computed as t_C/t_F. The numbers shown in parenthesis next to each resource quantity are the resource percentages for the target device.

Several important observations can be highlighted from the results in Table 4. First, the maximum operation frequency is independent of the array depth, allowing performance to scale adequately with depth. In fact, given the simplicity of the required control unit and storage, the longest path was mostly determined by the implementation of functions W and S inside the basic cells. Second, the parallelism achieved through the use of the systolic array greatly compensates for the refeeds that must be performed when the result is longer than the array depth. For instance, the result for the cover operation was of length 3386 on average, yet there is a speedup of more than 40× for $d_{sa} = 256$. Finally, the resource utilization to achieve competitive speedups is minimal and easy to estimate when scaling since it is almost completely attributable to the systolic array cells and overflow FIFO. Hence when using this structure as part of a bigger design, the designer could readily determine which parameters are best suited for the area/performance constraints and objectives.

Part of our purpose here is to argue that the reduction tasks can be implemented efficiently using the systolic array design. With Table 4, we intend to demonstrate the effect of each particular operation for users that might want to use any of them as part of a bigger design. Clearly, the ultimate speedup may be dictated by other components of the final design, as shown by Tables 5 and 6.

8.2. Interpolation Algorithm. We modeled the proposed interpolation algorithm in Verilog HDL using the reduction tasks and the architectures described in previous sections as building blocks. The behavioral simulation and synthesis tools as well as the software compilation parameters and platform were the same as in the reduction task experiments.

The results are shown separately for each of the two interpolation stages since, once the bases have been computed by the first stage, a scientist's intervention would be necessary to choose the most appropriate before proceeding to the second stage. For both stages, the synthesis tool determined maximum operating frequencies above 150 MHz, thus we chose 150 MHz for the experiments. The Xilinx ISE 11.1 place and route tools (with default effort level) were able to meet the timing requirements reported for Stages 1 and 2. We attribute this to the fact that the great majority of connections in our design are local and grow only linearly with n.

8.2.1. Stage 1. Table 5 reports timing and resource results for the first stage. Randomly generated sets of 100 n-tuples ($8 \leq n \leq 13$) were input to implementations with diverse systolic array depths. The table shows results for the d_{SA} case with the shortest execution time for each n. The smaller values of n generated fewer intermediate terms during computation,

Table 5: Experimental results for stage 1.

n	d_{SA}	Slices	Slice FFs*	4-input LUTS*	FIFO16/ RAMB16*	t_F (ms)	t_C (ms)	Speedup
8	32	1651 (1.85%)	1908 (1.07%)	3047 (1.71%)	25 (7.44%)	0.04	1.27	29.59×
9	64	3011 (3.38%)	3596 (2.02%)	5515 (3.1%)	27 (8.04%)	0.19	2.64	13.87×
10	128	5860 (6.58%)	7201 (4.04%)	11186 (6.28%)	33 (9.82%)	0.49	7.22	14.78×
11	128	6191 (6.95%)	7728 (4.34%)	11755 (6.60%)	43 (12.80%)	1.01	22.97	22.74×
12	256	11880 (13.34%)	14907 (8.37%)	22573 (12.67%)	55 (16.37%)	2.54	82.28	32.46×
13	256	14101 (15.83%)	16981 (9.53%)	25676 (14.41%)	95 (28.27%)	8.89	596.9	67.13×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

Table 6: Experimental results for stage 2.

n	Basis Vars	Class Reps	Slices*	Slice FFs*	4-input LUTs*	FIFO16/ RAMB16*	t_F (ms)	t_C (ms)	Speedup
8	2	23	7745 (8.69%)	8335 (4.68%)	13846 (7.77%)	28 (8.33%)	0.10	1.59	16.67×
8	3	63	7950 (8.92%)	8352 (4.69%)	14205 (7.97%)	28 (8.33%)	0.44	7.76	17.68×
8	4	90	7955 (8.93%)	8359 (4.69%)	14213 (7.98%)	30 (8.93%)	0.84	16.60	19.71×
9	2	25	8520 (9.56%)	9220 (5.17%)	15205 (8.53%)	28 (8.33%)	0.10	1.85	17.64×
9	3	77	8539 (9.58%)	9226 (5.18%)	15239 (8.55%)	28 (8.33%)	0.69	11.20	16.13×
9	4	94	8582 (9.63%)	9260 (5.2%)	15340 (8.61%)	34 (10.12%)	1.14	20.26	17.81×
10	2	24	9188 (10.31%)	10098 (5.67%)	16710 (9.38%)	31 (9.23%)	0.10	1.78	17.95×
10	3	65	9192 (10.32%)	10106 (5.67%)	16714 (9.38%)	31 (9.23%)	0.47	8.30	17.69×
10	4	92	9201 (10.33%)	10130 (5.69%)	16742 (9.40%)	33 (9.82%)	0.86	18.43	21.36×
11	2	25	10130 (11.37%)	10978 (6.16%)	18024 (10.12%)	34 (10.12%)	0.11	2.01	18.80×
11	3	75	10156 (11.4%)	11001 (6.17%)	18067 (10.14%)	36 (10.71%)	0.58	12.80	22.22×
11	4	92	10115 (11.35%)	11008 (6.18%)	18025 (10.12%)	37 (11.01%)	1.05	23.00	21.93×
12	2	25	10861 (12.19%)	11861 (6.66%)	19077 (10.71%)	36 (10.71%)	0.10	2.05	19.93×
12	3	70	10806 (12.13%)	11881 (6.67%)	19012 (10.67%)	38 (11.31%)	0.52	10.68	20.41×
12	4	93	10877 (12.21%)	11895 (6.68%)	19121 (10.73%)	38 (11.31%)	1.03	19.90	19.26×
13	2	25	11405 (12.8%)	12747 (7.15%)	20240 (11.36%)	39 (11.61%)	0.10	2.13	20.74×
13	3	71	11469 (12.87%)	12760 (7.16%)	20358 (11.43%)	40 (11.90%)	0.53	12.50	23.68×
13	4	91	11416 (12.81%)	12771 (7.17%)	20274 (11.38%)	41 (12.20%)	0.85	21.38	25.21×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

hence smaller values of d_{SA} produced better results since they incurred in less unnecessary overhead for filling and emptying. As the number of variables increases, the number of terms produced by Algorithm 1 increases exponentially, which is clearly evidenced by the run times, that is, columns t_F and t_C. The FPGA implementation achieves speedups of 13× to 67×, which mostly increase with n. The increase in speedup at the higher values of n can be attributed to a higher percentage of time of full pipeline utilization as well as the increased benefit of parallelism in the comparisons between the n-tuple variables, that is, the computation of the $r(i, j)$ terms in Algorithm 1. As expected, the use of necessarily serial processing tasks such as the multiplication of the DOLs to obtain the DNF (see Example 2) limit the speedup achieved by the FPGA implementation when compared to the results obtained using the cover subtask by itself.

Although the ultimate goal of reverse engineering might be understanding complete complex biological systems, recent reported models limit their exploration over specific subsystems or punctual mechanisms, for example, the study cell-cycle regulatory network of fission yeast and apoptosis (programmed cell death) [17, 18], using a moderate number of genes between 8 and 20. Our results lead us to believe that the advantages of using the proposed architecture in reconfigurable logic will be sustained even at bigger sizes. As the sizes increase our architecture will have to rely increasingly on iterations rather than parallel computations. Nevertheless, due to the custom nature (finite field) of the data, even at these sizes the parallelism over each iteration will be enough to offer significant speedups over the alternative, fully serial implementation in general purpose processors.

Speedup is maintained at a cost of increasing resource utilization. For the considered cases, BRAMs, which are used to implement the various FIFOs and RAMs of stage 1, is the fastest growing resource. This could pose a challenge to maintain performance at even higher n values. However, this issue can potentially be resolved by complementing the BRAM capacity with a dedicated external memory bank, as is commonly included in modern reconfigurable computer platforms [19, 20]. A study of the impact of external memory on the performance of the proposed architecture is future work. An advantage of our proposed architecture is that all the required memories behave as FIFOs. Thus, their accesses

patterns are predictable and (as part of future work) we can use this information to device a buffering scheme that hides the latency of accessing the external memories.

The increase in user logic (Slices, LUTs) in higher n can be circumvented by using shallower systolic arrays (smaller d_{SA}), which still maintain competitive speedups. For example, although not shown in the table, for $n = 13$ speedups of 30× and 45× were obtained by using d_{SA} values of 64 and 128.

8.2.2. Stage 2. As explained in Section 2.2, our algorithm uses a base X_i computed in Stage 1 to determine a set of equivalence classes $D/\equiv_{X_i}$ from the original set of n-tuples. The equivalence classes are then used to compute an interpolating polynomial using (3). A greater number of variables in the chosen X_i and a high variability among the n-tuples translates to larger cardinalities of $D/\equiv_{X_i}$. The greater the cardinality of $D/\equiv_{X_i}$ the higher the number of binomials computed by (3), which constitutes the bulk of computation for Stage 2.

The timing results shown in Table 6 support the previous statements. The sets of 100 n-tuples from Stage 1 along with bases of 2 to 4 variables were input to Stage 2. Observe that since the data sets were randomly generated there was high variability in the sets of n-tuples, which resulted in similar $D/\equiv_{X_i}$ cardinalities (column *Class Reps*) and run times (columns t_F and t_C) for the various sizes of n (for a given number of variables in the chosen basis). The FPGA implementation achieves speedups of 16× to 25×, which mostly increase with n. The moderate increment in speedup as n increases is attributed to the increased benefit of parallelism for the generation of binomials, and the comparison and addition of monomials.

Resource utilization distribution in Stage 2 is roughly uniform between the user logic, that is, Slices, and BRAMs. Furthermore, the increment in resource utilization for increasing values of n and numbers of variables in the basis is quite moderate, for example, the percentage of slices goes from 8.69% for $n = 8$ to 12.87% for $n = 13$.

9. Conclusion

This paper presents a new methodology based on Lagrange interpolation with two important properties: (1) it identifies redundant variables and generates polynomials containing only nonredundant variables, and (2) it computes exclusively on a reduced data set. The analysis of the methodology for its hardware implementation led us to the identification of several reduction tasks which were generalized to a simple algorithm. The generalized algorithm can be efficiently mapped to a systolic array in which each processing cell implements a pair of binary operations between an incoming and a stored value. The tasks of Boolean cover, distinctness, and multivariate polynomial addition were implemented and served as building blocks to the rest of the application. The FPGA implementation of the reduction operations and the complete application achieved speedups of up to 172× and 67×, respectively, as compared to software implementations run on a contemporary CPU, with moderate resource utilization.

Acknowledgments

An earlier version of this paper appeared as "A systolic array based architecture for implementing multivariate polynomial interpolation tasks" in the Proceedings of the 2009 International Conference on ReConFigurable Computing and FPGAs (ReConFig'09) [21]. Dr. E. Orozco was partially supported by Grant no. P20RR016470 from the National Center for Research Resources (NCRR), a component of the National Institutes of Health (NIH). The authors thank Dr. Humberto Ortiz-Zuazaga for his helpful discussions on microarray technology and bioinformatics data trends.

References

[1] U. Muller and D. Nicolau, *Microarray Technology and Its Applications*, Springer, New York, NY, USA, 2004.

[2] P. A. Ortiz-Pineda, F. Ramírez-Gómez, J. Pérez-Ortiz et al., "Gene expression profiling of intestinal regeneration in the sea cucumber," *BMC Genomics*, vol. 10, article 262, 2009.

[3] H. De Jong, "Modeling and simulation of genetic regulatory systems: a literature review," *Journal of Computational Biology*, vol. 9, no. 1, pp. 67–103, 2002.

[4] J. F. Knabe, M. J. Schilstra, and C. L. Nehaniv, "Evolution and morphogenesis of differentiated multicellular organisms: autonomously generated diffusion gradients for positional information," in *Artificial Life XI : Proceedings of the 11th International Conference on the Simulation and Synthesis of Living Systems*, pp. 321–328, MIT Press, 2008.

[5] W. J. Blake, M. Kærn, C. R. Cantor, and J. J. Collins, "Noise in eukaryotic gene expression," *Nature*, vol. 422, no. 6932, pp. 633–637, 2003.

[6] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*, Oxford University Press, Oxford, UK, 1993.

[7] D. Bollman, O. Colón-Reyes, and E. Orozco, "Fixed points in discrete models for regulatory genetic networks," *EURASIP Journal on Bioinformatics and Systems Biology*, vol. 2007, Article ID 97356, 8 pages, 2007.

[8] R. Laubenbacher and B. Pareigis, "Equivalence Relations on Finite Dynamical Systems," *Advances in Applied Mathematics*, vol. 26, no. 3, pp. 237–251, 2001.

[9] Z. Zilic and Z. G. Vranesic, "A deterministic multivariate interpolation algorithm for small finite fields," *IEEE Transactions on Computers*, vol. 51, no. 9, pp. 1100–1105, 2002.

[10] Y. Luo, "A local multivariate Lagrange interpolation method for constructing shape functions," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 2, pp. 252–261, 2010.

[11] M.-L. T. Lee, *Analysis of Microarray Gene Expression Data*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[12] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-Based skeleton for pairwise biological sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[13] A. S. Jarrah, R. Laubenbacher, B. Stigler, and M. Stillman, "Reverse-engineering of polynomial dynamical systems,"

Advances in Applied Mathematics, vol. 39, no. 4, pp. 477–489, 2007.

[14] R. D. Leclerc, "Survival of the sparsest: robust gene networks are parsimonious," *Molecular Systems Biology*, vol. 4, article 213, 2008.

[15] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," in *Proceedings of the 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL '2000)*, pp. 91–97, May 2000.

[16] E. P. Gribomont and V. V. Dongen, "Generic systolic arrays: a methodology for systolic design," in *Proceedings of the 4th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '93)*, vol. 668 of *Lecture Notes in Computer Science*, pp. 746–761, Springer, Orsay, France, April 1993.

[17] M. I. Davidich and S. Bornholdt, "Boolean network model predicts cell cycle sequence of fission yeast," *PLoS ONE*, vol. 3, no. 2, Article ID e1672, 2008.

[18] L. Tournier and M. Chaves, "Uncovering operational interactions in genetic networks using asynchronous Boolean dynamics," *Journal of Theoretical Biology*, vol. 260, no. 2, pp. 196–209, 2009.

[19] "Convey HC-1 Datasheet," http://www.conveycomputer.com/Resources/HC-1%20Data%20Sheet.pdf.

[20] "DRC Dev System DS2000 Datasheet," http://www.drccomputer.com/pdfs/DRC_DS2000_fall07.pdf.

[21] R. A. Arce-Nazario, E. Orozco, and D. Bollman, "A systolic array based architecture for implementing multivariate polynomial interpolation tasks," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 77–82, December 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 570279, 10 pages
doi:10.1155/2010/570279

Review Article

Reconfigurable Multiprocessor Systems: A Review

Taho Dorta, Jaime Jiménez, José Luis Martín, Unai Bidarte, and Armando Astarloa

Department of Electronics and Telecommunications, University of the Basque Country, UPV/EHU, 48013 Bilbao, Spain

Correspondence should be addressed to Taho Dorta, taho.dorta@gmail.com

Received 28 February 2010; Revised 31 July 2010; Accepted 26 October 2010

Academic Editor: Viktor K. Prasanna

Modern digital systems demand increasing electronic resources, so the multiprocessor platforms are a suitable solution for them. This approach provides better results in terms of area, speed, and power consumption compared to traditional uniprocessor digital systems. Reconfigurable multiprocessor systems are a particular type of embedded system, implemented using reconfigurable hardware. This paper presents a review of this emerging research area. A number of state-of-the-art systems published in this field are presented and classified. Design methods and challenges are also discussed. Advances in FPGA technology are leading to more powerful systems in terms of processing and flexibility. Flexibility is one of the strong points of this kind of system, and multiprocessor systems can even be reconfigured at run time, allowing hardware to be adjusted to the demands of the application.

1. Introduction

Multiprocessor Systems-on-Chip (MPSoC) represent an important trend in digital embedded electronic systems. MPSoC are systems-on-chip with more than one processor. New applications in modern embedded systems require complex multiprocessor designs to reach Real-Time (RT) deadlines while overcoming other critical constraints such as power consumption and low area. MPSoC seem to be the solution for such complex systems. A lot of applications such as networking, multimedia, and control benefit from this type of system. The perfect example of this is a cell phone. Current models must offer low power consumption and integrate a large number of functions such as audio and video encoding, image processing, and Internet access. MPSoC offer better performance with lower energy consumption in this kind of complex systems compared to uniprocessor embedded systems. Traditionally, the trend in uniprocessor systems was to improve performance by increasing clock frequency; now the trend is to work in parallel with lower frequencies, in order to reduce energy consumption [1–3].

In the field of MPSoC, the reconfigurable or FPGA-based multiprocessor is a new and increasingly important trend. It facilitates rapid prototyping and allows research into new architectures and communications techniques without the problems of MPSoC ASIC production. The number of papers published over the last three years has increased significantly. Figure 1 shows the number of publications in the Inspec database using "multiprocessor" and "FPGA" as search keywords.

Reconfigurable Multiprocessor Systems, also known as Multiprocessor-on-Programmable Chip (MPoPC) (or Soft Multiprocessor), are normally presented as a way of making prototype systems for subsequent implementation on an ASIC. Now, not only prototypes are implemented using FPGAs, but final designs too. The growth in FPGA capacity allows designers to implement a complete multiprocessor system in a single FPGA. The main FPGA companies offer the possibility of using softcore processors specially designed to fit well in the FPGA; also, some FPGAs allow the use of hard-core processors. Furthermore, FPGAs are equipped with on-chip memory blocks, peripherals, and interconnection circuitry. Run-time reconfigurability is one of the strong points of FPGA-based multiprocessors systems. This feature allows multiprocessor systems to be adapted to a particular application, gaining flexibility in the designed system.

In the following section, we discuss the viability of FPGA-based Multiprocessors. In Section 3, we provide some examples of FPGA-based multiprocessors implemented by the research community in recent years. In Section 4, we examine the challenges of MPoPC. After that, a number of different methods of design are presented. In the final section of this paper, we highlight a number of important aspects relating to MPoPCs.

2. Viability of FPGA-Based Multiprocessor Systems

The first question we must ask is whether there is any sense in implementing a multiprocessor system on an FPGA. The answer is that it depends. The main disadvantage of this kind of multiprocessor is reduced performance compared with ASIC multiprocessor systems. However, FPGA-Multiprocessor systems have a number of advantages that compensate for this in some way.

(i) Flexibility and reconfiguration. The number of softcore processors that can be included is limited only by the capacity of the FPGA. Also, it is possible to configure each processor independently adding cache, FPU modules, and so forth.

(ii) Less time-to-market. The design process does not include the manufacture of the IC, with a considerable reduction in design time.

(iii) Less cost. The process is cheaper. Nowadays a state-of-the-art FPGA is relatively cheap, enabling own design with a small work team. Furthermore, if there is an error in the system design, this is not decisive.

(iv) Scalability. FPGA-based multiprocessors systems can house an increasing number of microprocessors or peripherals if there are logic resources available in the FPGA. Therefore, using FPGA is the best choice in certain cases.

(i) Low-volume, mission-critical designs (e.g., radar and military applications).

(ii) Rapid design of new, reconfigurable multiprocessor systems.

(iii) Research field. New architectures, memory hierarchies, interprocessor communication, and so forth, can be developed.

(iv) Naturally-grown systems. Systems that have to be able to grow in features depending on the stage of development [4].

The use of FPGA technology provides numerous benefits for the design of embedded systems. These benefits include the ability to fix design bugs in the FPGA hardware, upgrade a system in the field, or simply swap out hardware functionality without redesigning the physical board that contains the FPGA [5]. FPGAs already provide a compelling time-to-market advantage over ASICs, and advanced tools targeting the needs of high-end FPGA designers would help establish clear leadership [6].

Ravindran et al. discuss the viability of soft multiprocessors [7, 8]. According to them, it is necessary to answer these two questions: (a) Can soft multiprocessors achieve performance levels competitive with custom multiprocessor solutions? (b) How do we design efficient systems of soft multiprocessors for a target application? In both papers, they demonstrate the viability of soft multiprocessors.

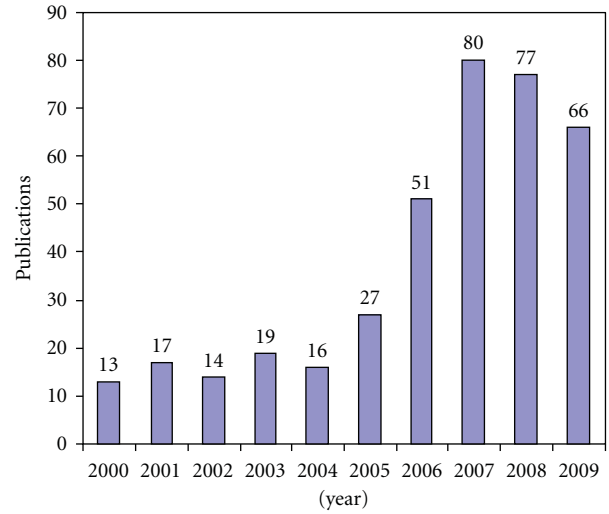


Figure 1: FPGA multiprocessor publications trend (2010 February).

3. FPGA-Based Multiprocessor Systems

In this section, we describe several multiprocessor systems implemented in FPGA. We have tried to include systems that represent the different trends in architecture and applications. First, we provide a little background information about MPSoC and FPGA-based multiprocessor systems. We also present the most widely accepted classification of MPSoCs: homogeneous and heterogeneous. Multiprocessor systems can be also classified as shared-memory systems or distributed-memory systems. We also highlight recent advances in run-time reconfigurable multiprocessor systems and give examples of these systems.

3.1. Main FPGA-Based Processors. In FPGA-based multiprocessor systems, the most widely used FPGA soft processors are made by one of the two main FPGA companies: Xilinx or Altera. The other option is to use open-source soft processors.

Xilinx is the most widely used brand in MPoPC systems. It supplies three main processors: softcore MicroBlaze (MB), PicoBlaze (8 bits reduced soft processor), and hard-core PowerPC. The number of PowerPcs is limited by the FPGA model to a maximum of four. The number of MicroBlaze and PicoBlaze processors is limited only by logic resources.

The MicroBlaze is a 32-bit RISC softcore processor. It uses the Harvard memory architecture, that is, it has a separate instruction memory and data memory. The MicroBlaze can issue a new instruction every cycle, maintaining single-cycle throughput under most circumstances. The shared-bus solution is the CoreConnect On-Chip Peripheral Bus (OPB), and every MicroBlaze has Fast Simplex Link (FSL) ports to make efficient point-to-point connections [9].

PicoBlaze is based on an 8-bit RISC architecture and can reach speeds of up to 100 MIPS in Virtex-4 family FPGAs. The processors have an 8-bit address and data port for access to a wide range of peripherals. The core license allows

Table 1: Main FPGA processor models.

Processor	Company	Type	Bits	Comments
MicroBlaze	Xilinx	RISC	32 bits	Harvard
PowerPC	Xilinx	RISC	32 bits	hard-core
PicoBlaze	Xilinx	RISC	8 bits	open-source
Nios	Altera	RISC	16 bits	obsolete
Nios II	Altera	RISC	32 bits	3 flavours
Leon 3	Gaisler	RISC	32 bits	open-source
OpenRisc 1200	OpenCores	RISC	32 bits	open-source
Mico32	Lattice	RISC	32 bits	open-source

them to be used freely, albeit only on Xilinx devices, and they come with development tools. The PicoBlaze design was originally named KCPSM which stands for "Constant (K) Coded Programmable State Machine" (formerly "Ken Chapman's PSM"). Ken Chapman was the Xilinx systems designer who devised and implemented the microcontroller.

The design flow provided by Xilinx for embedded systems is the Embedded Development Kit (EDK), which involves some limitations when implementing multiprocessor systems, even though it is widely used. These limitations are explained in Section 5.1.

Altera is the second most widely used processor utilized by researchers. It enables the use of Nios II (NII) processors, Avalon bus, and the EDA tool SOPC Builder, which is used as an aid in the development of multiprocessor systems.

Nios II is a 32-bit RISC embedded processor. Nios II is the evolution of the previous 16-bit Nios architecture. It is suitable for a wider range of embedded computing applications, from DSP to system control. Unlike Microblaze, Nios II is licensable for standard-cell ASICs through a third-party IP provider, synopsys designware. Through the designware license, designers can migrate Nios-based designs from an FPGA-platform to a mass production ASIC-device.

Nios II has three mostly unparameterized variations: Nios II/e, a small unpipelined 6-CPI processor with serial shifter and software multiplication, Nios II/s, a 5-stage pipeline with multiplier-based barrel shifter, hardware multiplication, and instruction cache, and Nios II/f, a large 6-stage pipeline with dynamic branch prediction, instruction and data caches, and optional hardware divider.

OpenRisc from OpenCores and Leon 3 from Gaisler are two very common open core soft processors. Leon 3 has more advanced functions than MicroBlaze and Nios II, but has the limitation of occupying a large amount of logic resources, so it is complicated to fit a large number of units in a single FPGA. We believe that it may be a good choice when larger FPGAs appear in the future.

Table 1 summarizes the main FPGA processor models (PowerPC is the only processor in the table that is not softcore).

3.2. Architecture Background. Normally the target application of the FPGA-based multiprocessor determines the architecture. There are three main system architectures: (1) Master-Slave, (2) Pipeline, and (3) Net. Also, it is possible to combine these: master-slave with pipeline, for instance, is very common.

(1) In master-slave systems, one or more processors act as the master processor, controlling the behaviour of the other slave processors.

(2) The pipeline approach is useful with stream applications; the architecture is composite with a chain of processors, every processor acting as a pipeline stage. The tasks are partitioned in time resulting in better performance if the application is adequate.

(3) Finally, net architecture refers to multiprocessor systems where there is no hierarchy between processors, all processors being able to communicate with each other when necessary. One example of this kind of system is the symmetric multiprocessor (SMP). A feature of SMPs is that all the processors are identical, so they are homogeneous multiprocessor systems.

Another important issue is the way the communications connections are implemented physically. There are 3 approaches.

(1) Point-to-point, where the processors are connected directly. High bandwidth is an advantage because it is not necessary to share the communications channel, but when systems grow this is not area efficient.

(2) Shared-bus, the traditional approach that derives from uniprocessor systems. It is the best known mechanism to communicate cores, but it is not effective in terms of performance because the bus can only be used by one processor at a time.

(3) The most recent and promising approach is the network-on-chip (NoC). The basis of this method of interconnecting cores, is to apply network background to on-chip systems. When there are a lot of on-chip cores, it is the solution that best combines area and performance. The idea is to use small routers inside the chip to enable communications between all cores of the system with low latencies.

There are two possible methods for exchanging information between processors: shared-memory and message passing.

(1) Shared-memory is used most frequently, one reason for this being that FPGAs have a limited amount of on-chip memory, so this method allows memory saving. Shared-memory systems, like SMPs, have the problem of synchronization and memory consistency. In FPGA-based designs, it is an important issue under research, because the most widely used soft processors do not have any solution to deal with these problems. Normally, shared-memory multiprocessor systems use a shared bus but there are also some systems with NoC interconnection.

(2) Message passing is mostly used in distributed memory systems [16] and consists of exchanging messages between processors. A message passing protocol is required.

Table 2 summarizes MPSoC architecture.

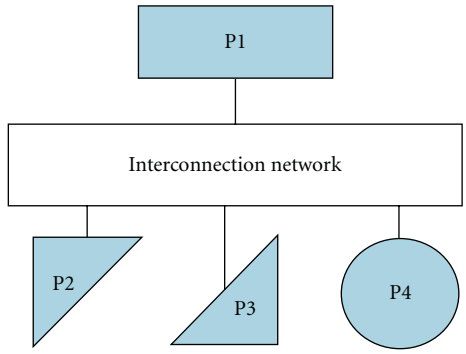


Figure 2: Heterogeneous multiprocessor systems.

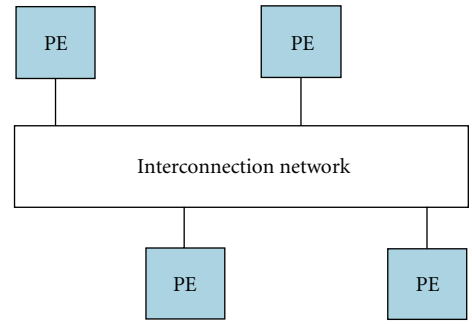


Figure 3: Homogeneous multiprocessor systems.

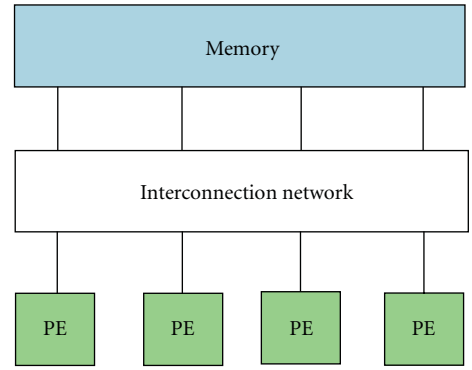


Figure 4: Shared memory multiprocessor systems.

3.3. Classification. The traditional classification of MPSoCs is heterogeneous, when there are different processors or even accelerators in the same system (see Figure 2) and homogeneous, when all the processors in the system are identical (same Instruction Set Architecture) (see Figure 3).

Normally, application-specific systems are heterogeneous. This is the common type of MPSoCs, as in FPGA-based multiprocessor systems. The reason is that MPSoC are usually implemented for embedded applications that behave intrinsically in a heterogeneous manner and require different kinds of processors.

Homogeneous MPSoCs are normally general-purpose systems, where all the processors are identical. In this kind of system, it is possible to increase the number of processors without changing the architecture (scalability property). It is easier to develop software for homogeneous systems.

Different taxonomies can be found in the literature. One is to classify systems in accordance with the memory architecture. There are two types: shared-memory (Figure 4) and distributed memory (Figure 5).

In shared-memory systems, all processors share the same memory resources; therefore, all changes made by a processor to a given memory location become visible to all the other processors in the system.

From an architecture design viewpoint, shared-memory machines are poorly scalable because of the limited bandwidth of the memory.

Shared-memory systems require synchronization mechanisms such as semaphores, barriers, and locks since no explicit communication exists. POSIX threads [17] and OpenMP [18] are two popular implementations of the thread model on shared-memory architectures.

In shared-memory architecture, different processes can easily exchange information through shared variables; however, it requires careful handling of synchronization and memory protection.

In distributed-memory systems, each processor has its own private memory; therefore, one processor cannot read directly in the memory of another processor. Data transfers are implemented using message-passing protocols.

Distributed-memory machines are more scalable since only the communication medium may be shared among processors.

Table 2: MPSoC architecture.

System arch	Comm arch	Comm Method
Master-slave	Point-to-point	Message passing
Pipeline	Shared bus	Shared memory
Net	NoC	

Distributed-memory systems require mechanisms for supporting explicit communications between processes. Usually a library of primitives that allow writing in communication channels is used. The Message Passing Interface (MPI) [19] is the most popular standard.

In distributed-memory architecture, a communications infrastructure is required in order to connect processing elements and their memories and allow the exchange of information.

We can also classify FPGA-MPSoC in (1) static reconfigurable multiprocessor systems and (2) run-time reconfigurable systems. Run-time reconfigurable systems represent state-of-the-art reconfigurable multiprocessors systems. They use the dynamic reconfiguration FPGA feature to adapt hardware at run-time to a specific application. So, we have several dynamic modules which are loaded by an arbiter depending on the target application. In Figure 6, a scheme of a basic run-time reconfigurable system is depicted.

3.3.1. Heterogeneous FPGA-Based Systems. Most MPoPCs are application specific. In this case, the system is application dependent, so the architecture is designed to achieve the best

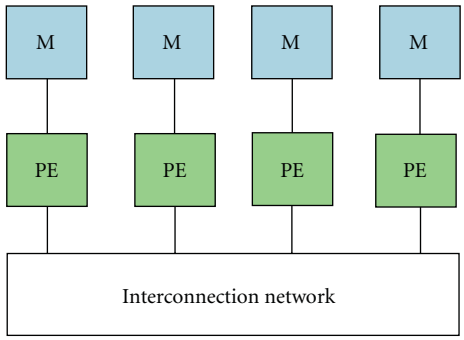


Figure 5: Distributed memory multiprocessor systems.

Table 3: Application-specific FPGA-Based multiprocessor systems.

Ref	Application	Syst arch	Comm meth	Comm arch
[7]	Networking	M-S/pipeline	mess passing	point-to-point
[10]	Control	M-S	mess passing	point-to-point
[11]	MPEG-4	M-S	shared mem	shared bus/HIBI
[12]	Bio	M-S/pipeline	mess passing	point to point
[4]	Industrial	M-S	shared mem	shared bus
[13]	Automotive	Net	shared mem	shared bus
[14]	Automotive	pipeline	mess passing	point-to-point
[15]	RT control	pipeline	mess passing	point-to-point

performance for the specific application. This architecture can be obtained by using design flow tools provided by FPGA vendors (known as hand-tune design) or using custom tools that obtain the architecture automatically from the specifications (known as automatic design) (see Section 5). There are systems that target different areas: multimedia, networking, control, and bioinformatics. First, we present application-specific FPGA-based multiprocessor systems (Table 3).

Network application is implemented in [7]. The solution proposed for IPV4 packet forwarding is a master-slave/pipeline approach. Communications among processors is point to point using MicroBlaze Fast Simplex Link (FSL) ports. There are different pipeline branches replicated in space to increase throughput. To demonstrate the viability of this solution, they compare the FPGA-based system proposed to an ASIC MPSoC with the same functionality. The FPGA-based system only loses a factor of 2.6X in performance normalized to area.

In [10], a master-slave/point-to-point multiprocessor system to control a laser-based transparency meter is presented. The authors are non-FPGA experts but they implemented the system successfully. According to them, the design tools made by the main vendors are easy to use. Tools abstract sufficient low-level details from the system design, allowing designers to implement successful MPSoC in a FPGA.

MPEG-4 Encoder is implemented in [11]. The system has master-slave architecture with support for message passing and shared SDRAM to interconnect NIOS processors. It uses a shared bus to connect instruction-shared memory and Heterogeneous IP Block Interconnection (HIBI) to connect data-shared memory by the plug and play method. It is an easy-to-scale computational system. Scalability is obtained through special parallelization: every image is divided into horizontal slices, and every slice is processed by 4 softcores in a master-slave configuration.

A stream chip multiprocessor (CMP) architecture for accelerating bioinformatic applications is presented in [12]. The architecture is master-slave/pipeline, and the memory hierarchy is customized for the target application. It is easy to increase the number of processors if there are enough logic resources in the FPGA. They compare the use of CPU, GPU, or FPGA for stream applications. GPU implementations achieve an order-of-magnitude speed increase compared to optimized CPU implementations. Custom hardware implementations on FPGA of the test algorithm also exploit data parallelism to achieve speedups over CPUs [25]. They intend to make the prototype in FPGA and after migrate to GPU implementation.

In [4] a master-slave shared-bus/shared-memory architecture is used for industrial applications. They use Nios II softcore processors and an Avalon bus. In this paper, the authors discuss the advantages of using FPGA-based multiprocessor systems in industrial applications. Industrial production machines have to be highly flexible in order to satisfy changes deriving from the demand for new products.

The automotive sector is another area of application for FPGA-based multiprocessor systems. Tumeo et al. present a real-time solution [13]. This is a shared-bus/shared-memory multiprocessor system, but offers the possibility of exchanging small data packets using the message-passing method through a crossbar.

Another solution for the automotive sector is presented in [14], which proposes an NIOS II-based multiprocessor system. The architecture is totally built for the occasion using a pipeline approach, consisting of a chain of 15 processors connected point to point. They use a distributed-memory architecture where each processor executes independent tasks, instead of parallelizing a unique task between the different processors. The reason they give for not using shared-memory architecture is that each task execution time and access latency shared-memory are not the same and therefore the shared bus became a bottleneck. However, with the distributed-memory approach, each microprocessor has local memory and the latency time is lower.

Real-time control applications are the target of the system proposed by Ben Othman et al. [15]. The system has two or three processors connected directly using MicroBlaze Fast Simple Link ports.

A combination of processors and accelerators is proposed by Claus et al. [26]. These systems achieve better results regarding speed and area/power consumption compared to architectures with only processors and no accelerators.

3.3.2. Homogeneous FPGA-Based Systems. While most of today's MPSoC systems are heterogeneous (the same occurred in the case of FPGA-based multiprocessor systems) in order to meet the targeted application requirements, in the near future, homogeneous multiprocessor systems may

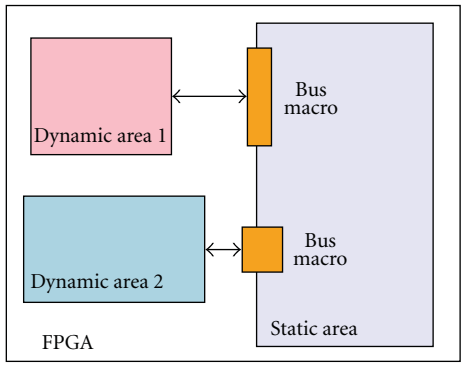


Figure 6: Architecture of a basic run-time reconfigurable system.

become a viable alternative, bringing other benefits such as run-time load balancing and task migration [16].

The homogeneous architectural style is used generally for data-parallel systems. Wireless base stations, in which the same algorithm is applied to several independent data streams, are one example; motion estimation, in which different parts of the image can be treated separately, is another. Normally, homogeneous multiprocessor systems are general purpose. In Table 4, a number of homogeneous MPoPC systems are summarized.

MPLEM [20] is a homogeneous, general-purpose 80-processor FPGA-based multiprocessor system.

Tseng and Chen [21] present a shared-bus/shared-memory multiprocessor system implemented on an Altera Cyclone. They use an instruction cache and Mutex core provided by Altera for synchronization.

A NoC-based homogeneous multiprocessor system is presented in [22]. This has 24 processors, is implemented in a Xilinx Virtex-4, and is external DDR2 constrained. To increase its compatibility and to facilitate the reutilization of the architecture, the IP Open Core Protocol (OCP-IP) standard is used to connect processor elements and NoC. Network interfaces (NIs) translate OCP to the NoC protocol. In this case, every MicroBlaze has an OCP adapter.

Almeida et al. utilizes 16 spartan3 Xilinx FPGAs to validate a homogeneous distributed-memory multiprocessor system [16].

Symmetric Multiprocessors (SMPs) are the best known general purpose multiprocessor system. Huerta et al. [23] explores FPGA capabilities for building Symmetric Multiprocessors systems. They implement a complete SMP system on FPGA using softcore processors. The system has centralized shared-memory architecture. The processors have no cache because of cache coherency problems. Hung et al. present a symmetric multiprocessor system with cache; they propose an HW/SW solution for the cache coherency problem [24]. An operating system with SMP functionalities is presented in [27].

Table 5 presents the area-performance relation of the main FPGA-based multiprocessor systems referenced previously. It is depicted the FPGA model, the use of area (S: Slices; FF: Flip-Flops; LUT: Lookup Tables; LE: Logic Elements), the use of on-chip memory, and the maximum speed of the system.

Table 4: Homogeneous FPGA-Based multiprocessor systems.

Ref	# CPU	Syst Arch	Comm meth	Comm Arch
[20]	80 MB	Net	shared mem	shared-bus
[21]	4 NiosII	Net	shared mem	shared-bus
[22]	24 MB	Net	shared mem	NoC
[23]	4 MB	Net	shared mem	shared bus
[24]	*n* Nios	Net	shared mem	shared bus
[16]	16 NPU	Net	mess passing	NoC

3.3.3. Run-Time Reconfigurable Systems. The new evolution in reconfigurable multiprocessor systems is the run-time reconfigurable system. This type of systems adds the dynamic reconfigurability feature of FPGAs to the power of having multiple processors. It adds a new degree of freedom in the design of multiprocessor systems. This freedom allows designers to adjust system performance at run-time obtaining better efficiency in accordance with the application. Göhringer et al. propose a new taxonomy for reconfigurable multiprocessor systems [29]. This is an extension of the taxonomy proposed and widely accepted by Flynn at 1966 [28] (see Figure 7). The new taxonomy is depicted in Figure 8. The superclass of this classification is the RAMPSoC [30]. Here, they propose a "meet-in-the-middle" design flow, which is to combine traditional top-down design with bottom-up approach. The bottom-up design is possible due to run-time reconfigurability. It allows hardware to be redesigned at run-time in order to obtain better efficiency in terms of speed, area, and power for a specific application.

Hubner et al. presents some pioneering work in [31], where run-time reconfigurability is proposed in multiprocessors systems at an early stage. Depending on the context, a different algorithm will be re-configured. They utilize the Dynamic Partial reconfigurable FPGA feature.

4. Design Challenges

The principal limitation of building a multiprocessor system on a FPGA is the amount of logic resources, specifically the amount of on-chip memory. Therefore, Shared-Memory architectures are widely used. It is possible to share data memory and/or instruction memory. The study by Kulmala et al. [32] examines the efficiency of sharing instruction memory.

Another solution is to use external memory to increase the amount of memory. In this case, the number of external memory blocks is limited by the package and pins of the FPGA [20, 22]. If the multiprocessor system does not fit in a single FPGA, it is possible to implement the system using a multi-FPGA approach [33].

An important issue in shared-memory multiprocessor systems is synchronization. The cores normally used in standard FPGA tool chains do not support atomic instructions and rarely support advanced synchronization mechanisms. For instance, the memory sharing mechanism (Mutex Core) provided by Altera is not efficient for multiprocessor systems [21]. Researchers present a number of ad-hoc solutions to resolve this problem. The GALS approach is adopted by

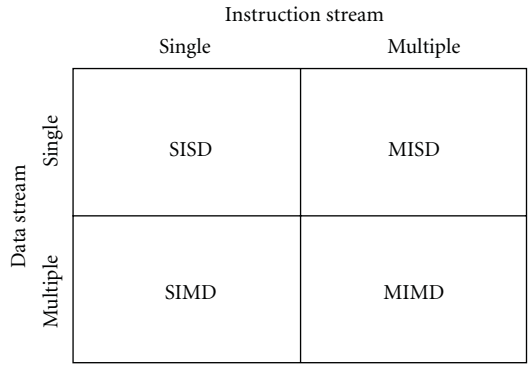


Figure 7: Flynn's taxonomy [28].

using the Bisynchronization method in [22]. Two hardware IP cores to perform lock and barrier functions are presented in [34]. Huerta et al. present an HW Mutex IP solution in [23].

Cache coherency is another critical point in FPGA-based multiprocessor design. Normally, the simple softcores used in such systems do not support any mechanism to guarantee cache coherency. Hung et al. propose an ad-hoc HW/SW solution [24].

Efficient on-chip communications between different cores is required to accommodate the increasing number of processors in FPGA-based multiprocessors. Traditional bus-share approach is not bandwidth efficient. Network-on-chip seems to be the best solution to interconnect cores [35, 36]. There are several papers that examine NoC in FPGA-based multiprocessor systems. The NoC approach is efficient compared to the shared-bus approach when the number of cores and data size increases [37, 38]. In [39], resource-efficient communications architecture is presented. Complete NoC-based multiprocessor systems implemented in a FPGA are presented in [22, 40]. Heterogeneous IP Block Interconnection (HIBI) is an approach designed and used by Salminen et al. [41]. A Multistage Interconnection Network (MIN) solution is presented in [42]. Göhringer et al. present a thorough study of different reconfigurable communication infrastructures targeting FPGA-based multiprocessors systems. They conclude that none of these fulfils all the goals required for use in FPGA-based systems for high-performance computing and present a new model: the star-wheels network on chip. This is a combination of the packet-switching and circuit-switching protocol [43].

Another important research area is parallel software efficiency. To maximize the potential of multiprocessors, it is necessary to develop parallel programming. In [44], Tumeo et al. present a tool to test and validate pipeline applications.

5. Design Methodology

The main two ways of building multiprocessor systems in an FPGA are: (a) manually, using the design flow provided by FPGA companies or (b) automatically, mapping applications to a specific architecture.

5.1. Hand Tuned Design. The first and most common method is manual design using the design flow available from FPGA companies. The two most commonly-used tools are EDK from Xilinx and SOPC Builder from Altera. Their advantage is the familiarity designers have with these tools and their ease of use [10]. The problem is that it is a hand-tuned method so it is difficult to explore all the design space manually to get the best possible architecture for a specific application [8]. This method may be a good choice to build small systems where the architecture to be implemented is known, or where there are not so many possibilities for Design Space Exploration (DSE), so in these cases it is not difficult to perform experiments and then choose the best configuration.

Another important issue is that, with FPGA company design flows, it is not possible to explore all the possibilities for designing multiprocessor systems. Normally, these tools are made to design uniprocessor systems so they have a number of shortcomings when dealing with multiprocessor systems. The most important shortcomings are as follows.

(i) Limitations in architectures for interprocessor communication. The main design flows only allow shared-bus and point-to-point communications. Complex MPSoC may require a higher bandwidth than a bus can offer or may need to be more area-efficient than point-to-point connections. Bafumba-Lokilo et al. [38] present a generic cross-bar network-on-chip for FPGA MPSoCs, and suggest that FPGA manufacturers should integrate such technology in their standard development flow.

(ii) Lack of effective mechanisms to share resources. Mutex Core provided by FPGA vendors seems to be inefficient as a synchronization method for multiprocessor systems [21].

(iii) Limitation in IP cores. The cores that can be included in one's design are restricted by the vendor library.

Researchers try to solve these limitations by creating *ad-hoc* IP blocks. These IP blocks act like add-ons to resolve this specific limitation. Some examples are

(i) Efficient Synchronization. Tumeo et al. [34] introduced two hardware synchronization modules for Xilinx MicroBlaze Systems. The modules can be completely integrated in the system using the EDK tool chain.

(ii) Cache Coherency Problem. A cache coherency module is presented in [24] as an IP Block. This paper assesses the solution and reports good results in performance.

(iii) An EDK-based tool is presented in [40]. It allows the designer to abstract low-level details of EDK and to create NoC-based systems rapidly.

(iv) Tumeo et al. also present a number of interesting modules to improve the performance of multiprocessor systems on FPGA. An Interrupt controller is presented in [45], and a DMA mechanism in [46].

Table 5: Area-performance relation.

Ref	FPGA	#proc	Slices/LE	%	BRam	%	MHz
[7]	Virtex II Pro 50	14 MB	11250 S	48	454 KB	87	100
[10]	Virtex II Pro 30	2 MB	6434 FF 9675 LUT	23 35	80 KB	60	100
[11]	Statrix EP1S40	1 N, 3 NII	20000 LE	50	314 KB	75	70
[12]	Virtex II Pro 50	15 MB	22500 S	96	352 KB	67	80
[4]	Cyclone EP1C20	3 NII	9000 LE	45	—	—	50
[13]	Virtex II Pro 30	4 MB	—	—	—	—	50
[14]	Statrix II 2S60	15 NII	30000 LE	50	—	—	100
[15]	Virtex II Pro 30	3 MB	5268 S	38	108 KB	35	—
[22]	FX-140	24 MB	55266	87	384 KB	69	

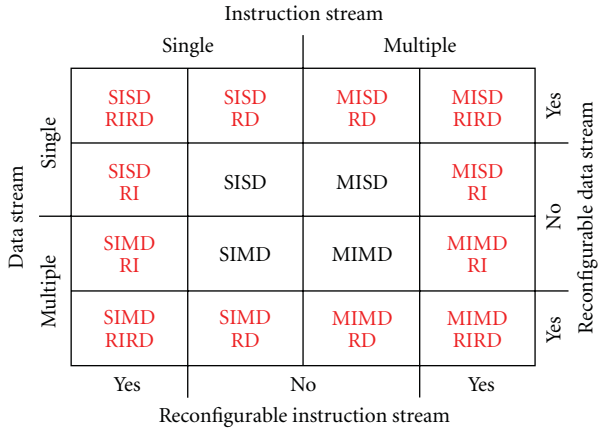


Figure 8: New taxonomy for reconfigurable FPGA-based systems proposed by Göhringer et al. [29].

It may be concluded that Xilinx EDK is the most commonly used design flow [7, 10, 12, 13, 15, 20, 22, 23, 43], possibly due to the fact that Xilinx is the largest FPGA vendor. However, there are also several Multiprocessor systems using Altera SOPC Builder design flow [4, 11, 14, 21, 24].

5.2. Automatic Synthesis Design. Another trend consists of using automatic synthesis tools to map an application to an architecture. The input parameters are: the application and, sometimes, the platform for the architecture and the cores to be used in the design. The result is a synthesized system that fits in the FPGA target.

The problem with this method is that there are no standards. Some automatic tools have appeared in the research community, but they have not been standardized or commercialized. So, they are not widely used. Nevertheless, this is a very important research area, but is still ongoing. Several papers present frameworks to perform this automation process; some of which we discuss here.

A promising possibility to increase designer productivity by automating the simultaneous design tasks of application mappings and IP selection is presented in [47]. Automation is possible because they have found a Solvable Integer Linear Programming (ILP) model that captures all the necessary design trade-off parameters of such systems.

Jin et al. use ILP [8] to solve the exploration problem. They propose an automated framework to assist the designer in exploring the design space (DSE) of soft multiprocessor microarchitectures. The objective is to identify the best multiprocessor on the FPGA for a target application and optimally map the application tasks and communications links to this micro-architecture. The framework proposed is evaluated improving the hand-tuned design presented in [7]. Therefore, DSE automation is important and becomes more critical when the system to be designed consists of many cores, due to in the fact that it offers greater possibilities for architecture configuration.

ESPAM [48] is a tool for automated design, programming, and implementation of multiprocessor systems on FPGAs implemented by Nikolov et al. While state-of-the-art development tools only support shared-bus architectures and point to point, ESPAM is general enough to implement multiprocessor systems with different communications topologies. ESPAM allows automated multiprocessor systems to be programmed in a way which significantly reduces the design time. It uses Kahn Process Networks (KPN) to specify the application.

MAMPS [49] presents a framework similar to ESPAM but with significant improvements. The framework generates application-specific architecture from the description of the applications. This time Kumar et al. use SDF to specify the application. This is the first flow that allows mapping of multiple applications on a single platform. The flow allows the designers to traverse the design space quickly.

Both ESPAM and MAPS are limited by FPGA hardware resources. The amount of on-chip memory is the main limiting factor. It limits the size of the multiprocessor system in ESPAM and the number of applications that can be implemented in MAMPS.

6. Conclusion

This paper surveys recent FPGA-based multiprocessor systems appearing in the literature. There has been a significant increase in publications in this area over the last three years.

After revising the literature, we can draw the conclusion that one of the most important issues with regard to designing multiprocessor systems in FPGA is the use of block RAM. The amount of on-chip RAM is fixed, and it limits the number of processors that can be included in one's design more than logic resources. Therefore, it is important to design memory-efficient systems. When building shared-memory systems, cache coherency and synchronization are critical aspects because the simple softcore processors offered by FPGA vendors have certain shortcomings in these areas. NoC is a weak point in design flows provided by vendors, since they do not include this approach in their tools. In our opinion, it is only a matter of time before they do so. Other important challenges in multiprocessor design are parallel software and automation of the design process. There are a number of important advances in this area but no standards have been established as yet. Run-time reconfigurability uses the dynamic reconfiguration feature of FPGAs to obtain a new degree of freedom in the design of multiprocessor systems, making these systems more flexible to target different applications using the same hardware.

Acknowledgments

This work has been supported by the Department of Education, Universities and Research of the Basque Government within the fund for research groups of the Basque university system IT394-10.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] W. Wolf, "The future of multiprocessor systems-on-chips," in *Proceedings of the 41st Design Automation Conference*, pp. 681–685, ACM, New York, NY, USA, June 2004.

[3] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd ACM/IEEE Design Automation Conference*, pp. 274–279, 2006.

[4] R. Joost and R. Salomon, "Advantages of FPGA-based multiprocessor systems in industrial applications," in *Proceedings of the 31st Annual Conference of the IEEE Industrial Electronics Society (IECON '05)*, pp. 4451–4506, November 2005.

[5] C. Wright and M. Arens, "Fpga-based system-on-module approach cuts time to market, avoids obsolescence," *FPGA and Programmable Logic Journal*, vol. 6, no. 6, 2005.

[6] S. Raje, "Catching the FPGA productivity wave," *Electronic Design*, vol. 52, no. 24, p. 20, 2004.

[7] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based soft multiprocessor system for IPV4 packet forwarding," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 487–492, August 2005.

[8] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (CODES+ISSS '05)*, pp. 273–278, ACM, Jersey City, NJ, USA, September 2005.

[9] P. Huerta, J. Castillo, J. I. Mártinez, and V. López, "A microblaze based multiprocessor SoC," *WSEAS Transactions on Circuits and Systems*, vol. 4, no. 5, pp. 423–430, 2005.

[10] J. Dykes, P. Chan, G. Chapman, and L. Shannon, "A multiprocessor system-on-chip implementation of a laser-based transparency meter on an FPGA," in *Proceedings of the International Conference on Field Programmable Technology (ICFPT '07)*, pp. 373–376, December 2007.

[11] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hännikäinen, and T. D. Hämäläinen, "A parallel MPEG-4 encoder for FPGA based multiprocessor SOC," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 380–385, August 2005.

[12] R. K. Karanam, A. Ravindran, and A. Mukherjee, "A stream chipmultiprocessor for bioinformatics," *SIGARCH Computer Architecture News*, vol. 36, no. 2, pp. 2–9, 2008.

[13] A. Tumeo, M. Branca, L. Camerini et al., "A dual-priority real-time multiprocessor system on FPGA for automotive applications," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 1039–1044, ACM, New York, NY, USA, March 2008.

[14] J. Khan, S. Niar, A. Menhaj, Y. Elhillali, and J. L. Dekeyser, "An MPSoC architecture for the multiple target tracking application in driver assistant system," in *Proceedings of the 19th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '08)*, pp. 126–131, July 2008.

[15] S. Ben Othman, A. K. Ben Salem, and S. Ben Saoud, "MPSoC design of RT control applications based on FPGA SoftCore processors," in *Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS '08)*, pp. 404–409, September 2008.

[16] G. M. Almeida, G. Sassatelli, and P. Benoit, "An adaptive message passing mpsoc framework," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 242981, 20 pages, 2009.

[17] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly & Associates, Sebastopol, Calif, USA, 1996.

[18] "The openmp api specification for parallel progremming," http://openmp.org/wp.

[19] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, Mass, USA, 1994.

[20] G.-G. Mplemenos and I. Papaefstathiou, "MPLEM: an 80-processor FPGA based multiprocessor system," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 273–274, April 2008.

[21] C.-Y. Tseng and Y.-C. Chen, "Design and implementation of multiprocessor system on a chip (mpsoc) based on fpga," in *Proceedings of the International Computer Symposium (ICS '09)*, 2009.

[22] Z. Wang and O. Hammami, "External DDR2-constrained NOC-based 24-processors MPSOC design and implementation on single FPGA," in *Proceedings of the 3rd International Design and Test Workshop (IDT '08)*, pp. 193–197, December 2008.

[23] P. Huerta, J. Castillo, J. I. Martínez, and C. Pedraza, "Exploring FPGA capabilities for building symmetric multiprocessor systems," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL '07)*, pp. 113–118, February 2007.

[24] A. Hung, W. Bishop, and A. Kennings, "Symmetric multiprocessing on programmable chips made easy," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 240–245, March 2005.

[25] T. Oliver, B. Schmidt, D. Maskell, D. Nathan, and R. Clemens, "Multiple sequence alignment on an FPGA," in *Proceedings of the 11th International Conference on Parallel and Distributed Systems Workshops (ICPADS '05)*, vol. 2, pp. 326–330, IEEE Computer Society, Washington, DC, USA, July 2005.

[26] C. Claus, W. Stechele, and A. Herkersdorf, "Autovision: a run-time reconfigurable mpsoc architecture for future driver assistance systems," *Information Technology Journal*, vol. 49, no. 3, pp. 181–187, 2007.

[27] P. Huerta, J. Castillo, C. Sánchez, and J. I. Martínez, "Operating system for symmetric multiprocessors on FPGA," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 157–162, December 2008.

[28] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[29] D. Göhringer, T. Perschke, M. Hübner, and J. Becker, "A taxonomy of reconfigurable single-/multiprocessor systems-on-chip," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 395018, 11 pages, 2009.

[30] D. Göhringer, M. Hübner, T. Perschke, and J. Becker, "New dimensions for multiprocessor architectures: on demand heterogeneity, infrastructure and performance through reconfigurability—rhe RAMPSoC approach," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 495–498, September 2008.

[31] M. Hubner, K. Paulsson, and J. Becker, "Parallel and flexible multi-processor system-on-chip for adaptive automotive applications based on xilinx microblaze soft-cores," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, p. 149a, April 2005.

[32] A. Kulmala, E. Salminen, and T. D. Hämäläinen, "Instruction memory architecture evaluation on multiprocessor FPGA MPEG-4 encoder," in *Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS '07)*, pp. 1–6, April 2007.

[33] A. Kulmala, E. Salminen, and T. D. Hämäläinen, "Evaluating large system-on-chip on multi-FPGA platform," in *Proceedings of the 7th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 4599 of *Lecture Notes in Computer Science*, pp. 179–189, 2007.

[34] A. Tumeo, C. Pilato, G. Palermo, F. Ferrandi, and D. Sciuto, "HW/SW methodologies for synchronization in FPGA," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 265–268, ACM, New York, NY, USA, February 2009.

[35] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[36] S. Kumar, A. Jantsch, M. Millberg et al., "A network on chip architecture and design methodology," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, p. 117, 2002.

[37] H. C. Freitas, D. M. Colombo, F. L. Kastensmidt, and P. O. A. Navaux, "Evaluating network-on-chip for homogeneous embedded multiprocessors in FPGAs," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 3776–3779, May 2007.

[38] D. Bafumba-Lokilo, Y. Savaria, and J.-P. David, "Generic crossbar network on chip for FPGA MPSoCs," in *Proceedings of the Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference (NEWCAS-TAISA '08)*, pp. 269–272, June 2008.

[39] X. Wang and S. Thota, "Design and implementation of a resource-efficient communication architecture for multiprocessors on FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 25–30, December 2008.

[40] S. Lukovic and L. Fiorin, "An automated design flow for NoC-based MPSoCs on FPGA," in *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08)*, pp. 58–64, June 2008.

[41] E. Salminen, A. Kulmala, and T. D. Hämäläinen, "HIBI-based multiprocessor soc on FPGA," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 4, pp. 3351–3354, May 2005.

[42] B. Neji, Y. Aydi, R. Ben-atitallah, S. Meftaly, M. Abid, and J.-L. Dykeyser, "Multistage interconnection network for MPSoC: performances study and prototyping on FPGA," in *Proceedings of the 3rd International Design and Test Workshop (IDT '08)*, pp. 11–16, December 2008.

[43] D. Göhringer, B. Liu, M. Hübner, and J. Becker, "Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit- and a packet-switching communication protocol," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 320–325, September 2009.

[44] A. Tumeo, M. Branca, L. Camerini et al., "Prototyping pipelined applications on a heterogeneous FPGA multiprocessor virtual platform," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, pp. 317–322, January 2009.

[45] A. Tumeo, M. Branca, L. Camerini et al., "An interrupt controller for FPGA-based multiprocessors," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS '07)*, pp. 82–87, July 2007.

[46] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "Lightweight DMA management mechanisms for multiprocessors on FPGA," in *Proceedings of the 19th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '08)*, pp. 275–280, July 2008.

[47] H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub, "Answer set versus integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 863630, 11 pages, 2009.

[48] H. Nikolov, T. Stefanov, and E. Deprettere, "Efficient automated synthesis, programing, and implementation of multi-processor platforms on FPGA chips," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, August 2006.

[49] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–27, 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 159367, 13 pages
doi:10.1155/2010/159367

Research Article

Mechanism of Resource Virtualization in RCS for Multitask Stream Applications

L. Kirischian,[1] V. Dumitriu,[1] P. W. Chun,[2] and G. Okouneva[3]

[1] *Embedded Reconfigurable Systems Laboratory (ERSL), Department of Electrical and Computer Engineering, Ryerson University, Toronto, ON, Canada M5B2K3*
[2] *MDA Space Missions, Brampton, ON, Canada L6S4J3*
[3] *Department of Aerospace Engineering, Ryerson University, Toronto, ON, Canada M5B2K3*

Correspondence should be addressed to V. Dumitriu, vdumitri@ee.ryerson.ca

Received 8 March 2010; Accepted 14 September 2010

Academic Editor: Lionel Torres

Virtualization of logic, routing, and communication resources in recent FPGA devices can provide a dramatic improvement in cost-efficiency for reconfigurable computing systems (RCSs). The presented work is "proof-of-concept" research for the virtualization of the above resources in partially reconfigurable FPGA devices with a tile-based architecture. The following aspects have been investigated, prototyped, tested, and analyzed: (i) platform architecture for hardware support of the dynamic allocation of Application Specific Virtual Processors (ASVPs), (ii) mechanisms for run-time on-chip ASVP assembling using virtual hardware Components (VHCs) as building blocks, and (iii) mechanisms for dynamic on-chip relocation of VHCs to predetermined slots in the target FPGA. All the above mechanisms and procedures have been implemented and tested on a prototype platform—MARS (multitask adaptive reconfigurable system) using a Xilinx Virtex-4 FPGA. The on-chip communication infrastructure has been developed and investigated in detail, and its timing and hardware overhead were analyzed. It was determined that component relocation can be done without affecting the ASVP pipeline cycle time and throughput. The hardware overhead was estimated as relatively small compared to the gain of other performance parameters. Finally, industrial applications associated with next generation space-borne platforms are discussed, where the proposed approach can be beneficial.

1. Introduction and Related Works

During the last decade progress in CMOS technology has allowed an increase in the amount of FPGA resources by almost an order of magnitude. This technological progress has made possible the implementation of relatively complex systems-on-chip (SoC) in a single FPGA but on the other hand, caused several drawbacks. Firstly, increased complexity of the SoCs overcomplicates the routing structure in the FPGA. This, in turn, reduces system performance and dramatically increases design compilation time which often requires many hours or even days. From the application point of view, the increased amount of resources allows effective utilization of FPGA devices for parallel execution of different tasks associated with multimodal and multitask algorithms and streamed data. However, from the reliability point of view, reduction of CMOS gate dimensions increases the probability of hardware faults. These faults can be caused by radiation, hidden manufacturing defects, and many other factors.

The problems above are not new in the field, and one possible approach to solving some of them can be the virtualization of homogenous FPGA resources. First, the computing model which allowed virtualization of resources in a single FPGA has been proposed by Caspi et al. in [1] and is called SCORE. This model assumed segmentation of computation process into fixed-size "pages" and time-multiplexing the virtual pages on available physical hardware. The model, however, did not consider multimodal workloads and was not actually implemented in an FPGA platform. Instead, the hypothetical single-chip SCORE organization was proposed with associated hardware requirements. In [2] Wallner has proposed the model for multithread computation by virtualization of resources. There are also several works reporting

implementation of the concept of resource virtualization in RCS (e.g., [3, 4]). Most of the above implementations are oriented towards coarse-grained or hybrid RCS architectures. However, to the best of our knowledge, none of the above models provides aggregative solution for all the above problems. In contrast with the above, our approach of virtualization of computing resources is oriented towards fine-grained homogenous RCS architectures and multitask, multimodal workloads. There are several advantages which an architecture based on fine-grain homogenous resources could provide. First, it allows rapid and precise run time adaptation for workload variations. At the same time, fine-grain RCS architectures makes possible the restoration of on-chip computing circuits from almost all possible hardware faults. To make use of these advantages two concepts have been proposed. The first is the concept of application specific virtual processor (ASVP). An ASVP is a data stream processing pipelined data-path optimized for a certain algorithm (task segment) and predetermined data-stream structure [5]. Virtualization of resources in ASVP assumes that it can be on-chip (in the FPGA) assembled from virtual hardware components (VHCs). The VHC is the second concept proposed for effective utilization of homogenous logic and routing resources in FPGA-based RCS. VHCs in our consideration are macro-function specific modules [5] represented (and stored) in the form of a configuration bit streams. A VHC consists of two parts: (i) a functional part (similar to an IP-core) which performs function-specific data processing and (ii) an interface part which provides uniform input/output ports as well as synchronization circuits. The uniform interface allows plug-and-play allocation of the VHC in addressed slots of a partially reconfigurable FPGA. This approach makes possible run time on-chip ASVP assembling according to current task and task mode. ASVP assembling, therefore, requires an application-specific static infrastructure to be loaded into the FPGA at start-up time. This static part incorporates the set of slots-and-sockets for VHCs and associated communication routing for all modes of the task. Then, a set of VHCs can be loaded into predetermined slots and combined to create a specific ASVP. Each variant of the ASVP is associated with one of multiple modes of task operation. When the mode of operation is completed or interrupted, the associated ASVP should be modified in run time by replacing one or more VHCs in its architecture. This kind of ASVP reconfiguration dictates the necessity of run time component relocation in the target FPGA. In other words, it may be unknown which slot-and-socket will be available in the ASVP. Thus, the uniform VHC interface should allow allocation and then relocation of the virtual component to an available slot. In recent years a number of research efforts have been directed towards the problem of component relocation in dynamically reconfigurable embedded systems. A number of bit-stream manipulation methods have been proposed [6–8] which alter the bit-stream structure as it is downloaded to the FPGA. However, these works concentrate solely on the bit-stream manipulation process, avoiding consideration of all steps of the real-time relocation process. Becker et al. provide a method for relocation between portions of a device which are not identical in terms of resources [9]. Here, the authors present both the bit-stream manipulation method as well as some of the requirements which must be met for this method to work. However, the system infrastructure requirements are not discussed.

Hübner et al. propose a complete framework for virtual components and component relocation based on the Xilinx Virtex-II family of columnar devices [10]. The authors make use of the network-on-chip interconnect concept to accommodate communication between components. As well, the JBits software package is used to relocate components between slots. Finally, Bobda et al. present the Erlangen Slot Machine (ESM), a reconfigurable computing system based on the Virtex-II family of columnar devices [11]. The authors describe the architecture of the complete platform, including the interconnect infrastructure provided for module communication. Because of the columnar organization of the FPGA device used, the communication infrastructure is either incorporated inside the architecture of each virtual component, or deployed externally, inside a secondary infrastructure FPGA. This, in turn, reduces the effectiveness of the approach compared with a complete on-chip solution because of complicated system infrastructure (e.g., motherboard plus several baby boards, etc.).

In contrast with the above approaches, the component relocation process presented in this paper targets tile-based FPGA devices (i.e., the Xilinx Virtex-4 or Virtex-5 families of FPGA devices), which offer more flexibility in the implementation of ASVPs. For example, the communication infrastructure can now be separate from the components themselves. Thus, the research addresses the architectural requirements needed for various levels of component virtualization, including on-chip assembling and relocation of components. In the case of relocation, the proposed framework provides an example of how complete streaming systems can be constructed using component relocation. This includes both the on-chip infrastructure as well as the behavior of components themselves.

The above mechanism of dynamic VHC relocation allows the implementation of concepts of spatial and temporal locality similar to virtual memory organization in advanced sequential processor architectures. In other words, the sooner a VHC may be requested for operation, the closer to the configuration memory of the target FPGA it should be located. Thus, at the system level of architecture a memory hierarchy for configuration bit streams should be organized. This memory hierarchy needs to contain a cache for VHCs as well as Main Storage for VHC bit streams for all tasks and their modes of operation. This organization of the memory hierarchy allows one to take advantage of the following facts: (i) only one of the modes of each task is active at a time, (ii) only a few of all tasks are active at a time and (iii) the same VHCs can be used for different tasks in different FPGA slots.

Therefore, the proposed approach allows a dramatic increase in cost-effectiveness of RCS through the reduction of computation resources, because nonactive data-processing circuits are stored in the form of configuration bit streams but not in the form of actual circuits in the FPGA.

Furthermore, power consumption in this case will also be reduced accordingly. At the same time, restoration of transient hardware faults can be done much faster because scrubbing procedures should be applied only to the slot affected by single-event-upsets (SEUs) but not the entire FPGA. Permanent hardware faults, on the other hand, can be mitigated by rapid VHC relocation to anther available slot(s) [12]. Thus, virtualization of computing resources based on run time on-chip assembling of ASVP and VHC dynamic relocation may become a promising solution for all problems listed in the beginning of this section. However, there are several issues and uncertainties regarding the implementation of the above ideas and concepts in real RCS. Therefore, the focus of this research was on the following aspects: (i) the development of an RCS architecture with a memory hierarchy for virtual components which will be able to support dynamic allocation and on-chip assembly of application specific virtual processors (ASVP), (ii) investigation of the mechanisms for run time on-chip assembly of ASVP from virtual hardware components (VHCs) and (iii) mechanisms for run time VHC relocation in predetermined regions of the FPGA device and automated reassembly of the associated ASVP in run time.

All the above aspects of the research presented here have been targeted specifically towards streaming applications with high data-rate requirements. The proposed framework at the system level and on-chip was developed with autonomous mobile applications in mind, where minimization of power consumption and high reliability are the major concerns. It was clear that all components of the above framework should be prototyped and tested on hardware platform(s). Therefore, a prototype platform, the multitask adaptive reconfigurable system (MARS) was designed, manufactured and tested. In addition, high-frame rate (up to 200 fps) stereo vision sensors have been developed and incorporated with reconfigurable platforms as well as VHCs for 3D multistream video processing.

The rest of the paper presents the results of the above investigation and implementation. Section 2 provides a specification of multitask and multimode workloads, as well as the system architecture of the multitask adaptive reconfigurable system (MARS) platform. Section 3 presents the on-chip architecture organization and analyzes the effectiveness of this approach. Section 4 presents the mechanism of component relocation. In Section 5, a description of the industrial applications of the proposed mechanism is given. Finally, Section 6 concludes the paper.

2. RCS Architecture and Workload

The architecture of an RSC which is able to support virtualization of on-chip computing resources consists of two parts: (i) system level architecture and (ii) on-chip architecture. Both parts directly depend on the workload specification. Thus, a workload specification is discussed first.

2.1. A Multitask and Multimode Workload. For the purposes of this discussion it was assumed that every application is fundamentally comprised of operations that are arithmetic or logic. Associated operations are grouped together to create a *task* (**Ts**). Each task is considered an independent information object in regard to any data or control dependences with other tasks. Nevertheless, tasks may be linked by input data-streams or share the same resources for the output data. Once all interactions and communication between tasks are arranged, higher levels of computation (i.e., application −**AP**) can be formed. In other words, the application can be considered as a set of tasks that are connected (or disjointed) for the system at times. The connected set of tasks by output(s) of the system is referred to as a *thread* (**Th**). Different functionalities of a task are referred to as modes of the task (**Md**). This means that the task algorithm and/or data structure may have some variations. Each variant of the algorithm and/or data structure can be considered as a mode of the task.

The set of applications over the life time of the system is called the workload (**W**). As per definition made above, any change of the mode **Md***j* of a task **Ts***i* means that the set of VHCs—{VHC*ij* } has to be replaced by a new set—{VHC*ik*} associated with mode **Md***k* of the same **Ts***i*. In most real cases not all VHCs involved in a mode **Md***j* of a task ***Tsi*** should be replaced when the mode changes to **Md***k*. Usually, there are very few new VHCs which have to replace existing VHCs. This is true because in most real cases there is quite a limited difference in functionality between different modes of operation in a task. Furthermore, the request for mode switching is usually associated with a certain response time for switching to mode **Md***k*-**T***SW(ik)*. Therefore, reconfiguration of a number of VHC slots—**N***ij* associated with the new mode **Md***k* of a task **Ts***i* should be done within this period of time. Thus, the granularity of VHC slots directly depends on mode switching time, number and size of VHCs to be replaced and the bandwidth of the FPGA configuration port—**BW**conf.

In this case VHC slot size in a certain FPGA can be estimated by the size (volume) of the configuration bit-stream required for a slot—**BS**slot—and equal to

$$\mathbf{BS}\text{slot} \le \mathbf{T}\text{mode}(ij) * \frac{\mathbf{BW}\text{conf}}{\mathbf{N}ij}. \tag{1}$$

For example, if the mode switching time is *1ms* and the bandwidth of the configuration port is 800 Mb/*s* (e.g., for SelectMap-8 configuration port in Xilinx Virtex FPGA), the size of the VHC's configuration bit-stream will be equal to 780 Kbit for the VHC slot if only one VHC has to be replaced. It is possible to calculate the number of logic slices available for the VHC slot. In the case of the smallest Xilinx Virtex-4 FPGA (XC4VLX15), the slot size will contain 1030 slices and 6 VHC slots will be available in this FPGA. In contrast to this, the largest FPGA in the Xilinx Virtex-4 family may contain 64-VHC slots.

Keeping in mind the size of the configuration bit-stream for the VHC slot, volumes of VHC cache and main VHC memory can be estimated. In all cases the number of different VHCs to be stored in the Main VHC memory and VHC cache is expected to be quite large. Based on the above example, it is possible to estimate the number of VHCs to

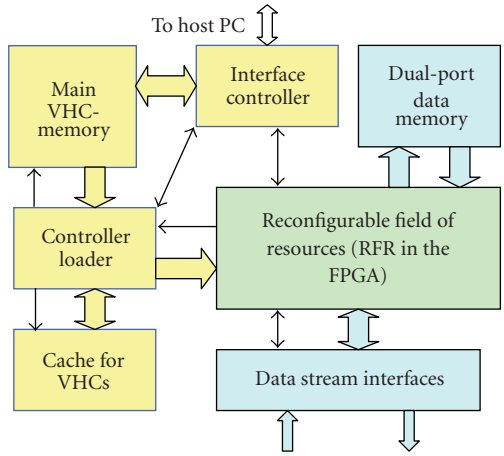


Figure 1: Block diagram of RCS architecture.

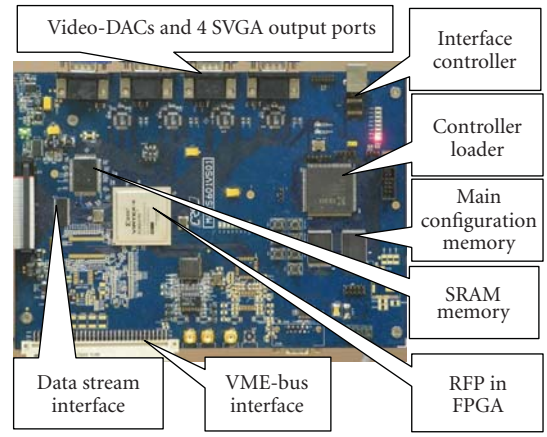


Figure 2: Component layout on MARS platform.

be stored in relatively cheap Flash memory. For example a 10 GB Flash memory module can store up to 107,546 VHCs. It is necessary to mention that this amount of VHCs is associated only with the current application. The bit streams associated with all other applications of the workload—**W** can be stored in remote locations (e.g., host PC). Therefore, the configuration memory hierarchy should consist of the following levels:

(1) *Operational level*: configuration memory of the target FPGA device(s) which contains bit streams of all active VHCs processing the data in current mode of operation,

(2) *Cache level:* VHC cache deployed as close as possible to the target FPGA device(s) which stores bit streams of VHCs that can be requested by other modes of the active tasks,

(3) *Main level:* main VHC memory deployed on the same board as target FPGA device(s) which stores bit streams of all VHCs required for the set of tasks in current application,

(4) *External level:* External (secondary) storage of the VHC bit streams required for all applications in the workload. This storage can be deployed in the hard disk drive of a host PC or any other carrier.

2.2. System Level Architecture of RCS. According to the above memory hierarchy for VHCs, the general architecture of an RCS able to support virtualization of FPGA resources is shown in Figure 1.

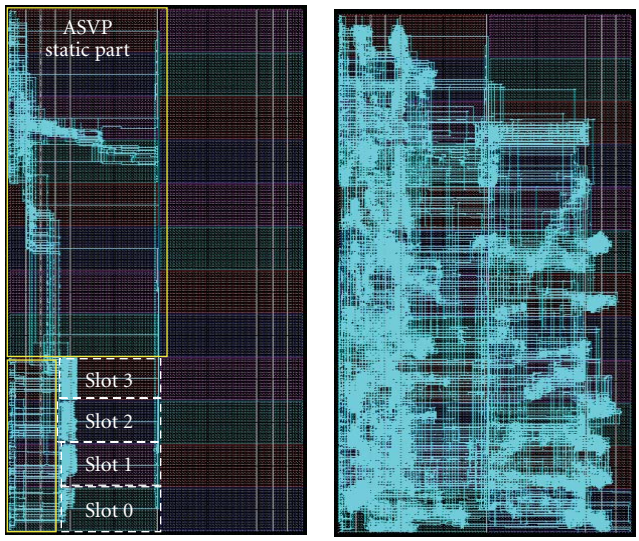
The major elements of this architecture are the following. (i) Reconfigurable field of homogenous resources—RFR (in the target FPGA), (ii) configuration memory hierarchy: consisting of the Main VHC memory, Cache for VHCs and configuration memory of the FPGA, (iii) configuration controller(s) and (iv) interface to the external storage of VHCs (e.g., host PC) with associated interface controller.

The actual system architecture of the RCS with the ability to support the virtualization of homogenous FPGA resources has been implemented in the form of a custom prototype board, the multitask adaptive reconfigurable system (MARS). This RCS has been designed around a Xilinx Virtex-4 FPGA device (XC4VLX160) with tile-based architecture. The platform and layout of major elements of the above architecture is shown in Figure 2. The MARS platform was designed for multistream high data rate video-processing applications. In the current configuration, MARS provides a wide variety of interfaces to video-sources and video-displays. It contains 4 SVGA output ports, one LVDS input port with bandwidth up to 2.128 Gb/s (16 bit × 133 MHz) and a universal VME-bus interface. The Main VHC memory was built on 4 NOR-Flash memory modules. The parallel read of 4 × 16 bit data words from these modules allows the controller-loader to provide the required timing for (a) accelerated parallel load of configuration bit streams for the entire FPGA configuration via SelectMap32 (3.2 Gb/s) and (b) parallel load of VHC bit streams for partial reconfiguration of selected FPGA slots via SelectMap-8 (800 Mb/s). The VHC-cache circuitry was not utilized at this stage of experiments.

3. On-Chip Architecture Organization

The initiation of task **Ts***i* consists of the configuration of the static part of the associated **ASVP***i*. The static part of the ASVP consists of the following elements: (a) Interfaces (IOBs) to external devices (e.g., video sensors, VGA-displays, etc.), (b) clocking and synchronization circuits, (c) data-memory units (BRAM based) and (d) communication infrastructure between VHC slots, I/O ports and data-memory units. Once the static portion of the ASVP is in place, the set of VHCs associated with the required mode of operation are loaded to predetermined VHC slots.

In Figure 3(a) a sample on-chip microarchitecture of a 4-channel parallel video-processing ASVP is shown. The upper and bottom-left boxes in the post-place and route layout show the *static part* of the ASVP design. The dotted boxes (bottom right) indicate the areas reserved for the VHC slots. In this picture VHCs are already loaded into the slots but do



(a) Reconfigurable design (b) Fixed design

Figure 3: (a) Sample of ASVP design for the current mode of a task, (b) Fixed design implementing all modes of a task.

Figure 4: Experimental setup for multistream/multimode Video-processing on the MARS platform.

not use the full amount of reserved resources. To analyze the effectiveness of the system a series of experiments for parallel video stream processing have been performed. One of the examples considered 2 parallel video streams (from a custom stereo vision sensor) to be processed on 4 video processors each of which could work in one of 7 modes of operation. Output streams have been directed to four SVGA monitors. The setup of this example is shown in Figure 4.

All modes and resource utilization for the associated VHCs for this example are listed in Table 1. In this case, each mode was associated with a certain VHC (function-specific video-processing circuits) to be configured in one of the above VHC slots (from Slot 0 to Slot 3).

To analyze the resource utilization, a comparison was made with a fixed (nonrecon19.9figurable) design. The fixed design reflected the traditional approach for static FPGA system-on-chip development. This design incorporates all video processors accommodating all modes of operation in one application-specific circuit. The on-chip layout of the fixed design is shown in Figure 3(b). In the presented example the ASVP based approach never exceeded 27% of the area utilized by a fixed design with the same function-

Table 1: Modes and resources for associated VHCs.

Mode	Required resource 4-input LUTs	Resource utilization 4-input LUTs
Camera display	172	33.6%
Normal edge	177	34.6%
Inverse edge	174	34.0%
PingPong edge	341	66.6%
Color intensity (Red)	102	19.9%
Color intensity (Green)	102	19.9%
Color intensity (Blue)	102	19.9%

alities. An increase in cost-effectiveness is gained because of virtualization of actual processing circuits. In other words, instead of keeping all processing circuits in real hardware (as for the fixed design approach) they can be stored in the form of configuration bit streams in Main VHC memory.

The same is true for power consumption minimization. A 29.8% reduction in power consumption was measured for the ASVP-based design (1.563 W) compared to the fixed system (e.g. 2.03 W). This reduction can also be attributed to lower resource utilization in the temporal domain. The power consumption was measured directly from the power supply (before the on-board DC-DC converters and power regulators). Initially power was measured with an "empty" system before configuration of the design. Then, power consumption was measured on the system with loaded fixed system design and consequently for each mode of the ASVP based design. The difference of power consumption between the "loaded" system and "empty" system was considered as the power consumption overhead associated with the loaded design

Obviously, the effectiveness of the proposed approach directly depends on workload specifics. However, it is necessary to mention that the effectiveness of the resource utilization increases proportionally to the number of modes in the task as well as the number of independent tasks running in parallel and using the same VHCs.

On the other hand, the cost-efficiency of the proposed approach directly depends on the cost of the hardware overhead. The additional cost of the memory hierarchy and associated controllers (Figure 1) increases the cost of the platform. However, the minimization of resource utilization by using the ASVP approach allows processing of the same workload on a smaller FPGA device. This, in turn, can bring a dramatic reduction of system cost. For example, the Xilinx XC4VLX160 (~16 M system gates) unit cost is around $5,600 USD (depending on number of I/O, package type, etc.) while a device from the same family which is four times smaller, the XC4VLX40, costs around $800 USD (http://avnetexpress.avnet.com/—list prices as of March 8th 2010). Comparing with the above cost reduction the hardware overhead cost required for configuration memory hierarchy can be considered as negligible. In our case, the hardware overhead cost did not exceed $200 USD including: 1GB Flash-based Main VHC memory, 4 MB SRAM-based VHC Cache, Xilinx XC95288 CPLD-based

Controller-Loader and PIC/USB interface to host computer. This is about 4.2% of the FPGA cost reduction considering Xilinx Virtex-4 or Virtex-5 FPGA families.

In addition to the cost of the hardware overhead in the system-level architecture, the effective implementation of the above concept of virtualization of FPGA resources directly depends on the mechanism for run time on-chip ASVP assembling. The foundation of this mechanism is a communication infrastructure which allows the allocation of VHCs to any VHC slots and further run time relocation of VHCs to new locations. This aspect of resource virtualization is discussed in the next section.

4. ASVP On-Chip Assembling and VHC Relocation Mechanisms

The major part of the research into the virtualization of hardware resources deals with the topic of run time on-chip ASVP assembly. This process is based on run time allocation and further relocation of VHCs. This research is important in its own right, as it increases the flexibility of a system, and adds additional fault-tolerance capabilities. However, some of the subjects encountered when implementing relocation-enabled systems are applicable to VHC-based systems in general. The remainder of this section will present a series of relocation experiments aimed at tile-based FPGA systems. In particular, the Xilinx Virtex 4 FPGA family is used in the experiments properly. This section will describe the experimental system used for the experiments, and will describe the obtained results. In addition, an on-chip support infrastructure which allows component relocation is introduced and described; as well, some design guidelines are discussed regarding the implementation of the VHCs themselves.

To perform relocation experiments, a dedicated system was designed and constructed from the ground up. The system incorporates many design decisions which were dictated by the requirement for relocation and dynamic partial reconfiguration, as dictated by the xilinx partial reconfiguration flow [13]. The system performs video processing operations on an acquired video stream and displays the results using a VGA interface. The video stream has a resolution of 640×480 pixels and a frame rate from 30 fps up to 200 fps. The stream can be displayed as it is, or a number of spatial color masks can be applied to it prior to display. The number and type of masks are variable, and the mask modules themselves are implemented as Virtual Hardware Components. For the first round of experiments, two types of mask VHCs were implemented: a smoothing mask and a y-dimensional Sobel filter.

4.1. Experimental System-on-Chip Architecture. As mentioned above, the implemented video processing system can display an unaltered VGA video stream, or it can apply two types of spatial masks to the stream, in various configurations. The experimental setup consists of a camera board, the MARS platform and a host computer. A block-diagram of the microarchitecture of a system with 4-VHC slots is shown in Figure 5.

The acquisition and display units are static components of the system, as well as the communication infrastructure. Together, the acquisition and display units act as the interface between the camera board, a VGA capable screen and the video processing components. All I/O signals (with the exception of reset and clock signals) are connected to these two units. The VHC slots in the system can accommodate one dynamic component each, and provide the actual video processing capabilities of the system. Finally, the communication infrastructure routes data between the various components in the system. All the above static elements are incorporated into the initial static architecture of the ASVP and are loaded into the target FPGA at start-up time. The host computer performs two tasks: configuration and control. At this stage of the experiments the FPGA JTAG configuration interface was used to perform all configuration activities, either full or partial. To control the communication interconnect, the host computer makes use of a serial interface and the microcontroller present on the MARS platform.

The complete system is conceived to operate as a deep data-processing pipeline. As virtual components are added or removed from the system, the depth of this pipeline varies. The components themselves are pipelined in nature, and the same basic interface is shared by all components in the system. This ensures that each component can be placed at any location in the system, and the pipeline structure will not change. By adopting this approach, the 30-frames-per-second frame rate of the acquired video is maintained regardless of the number of masks being applied. A deeper pipeline will lead to an increase in initial latency, but the data throughput will remain unaltered.

Two variations of the described system were implemented for experimentation. Both system types are identical in nature, with the only difference being the number of VHC slots present and the structure of the communication infrastructure. The first implemented system contains only four VHC slots; a second version of the system contains eight VHC slots. Two versions of the communication infrastructure were implemented, one for each system, and each will be described in detail in the following section. The post-place-and-route diagram of the four-slot system is shown in Figure 9, while that of the eight-slot system is shown in Figure 13.

4.2. Communication Infrastructure. The communication infrastructure plays a vital role in the operation of the complete system. Most obviously, it provides programmable communication paths between components. There are multiple methods available for providing communication paths between multiple on-chip components. These include the shared bus, the fully and partially connected crossbar, as well as various hybrid architectures.

For the proposed video system a fully connected crossbar was used. This architecture was selected as it provides complete connectivity inside the system, and minimizes latency due to resource sharing. The basic architecture of the cross bar is shown in Figure 6.

The core of the system is composed of a collection of multiplexers, each of which can route any cross bar input to any

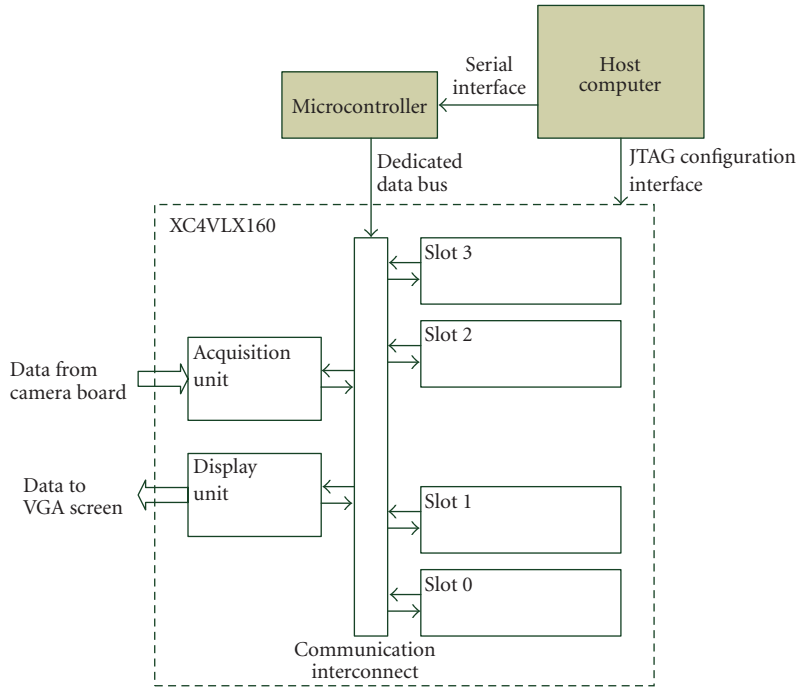


Figure 5: Four VHC slot microarchitecture.

output. As with many other communication protocols, separate ports are used for input and output. The cross bar is controlled by setting the multiplexer control signals to the appropriate value. The problem of communication delay variation is unique in on-chip systems; in a traditional design, the circuit remains static once implemented, and there can be no delay variation. However, in partially reconfigurable systems using VHCs, such delay variations can occur, when a component is relocated. The concept is illustrated in Figure 7, which shows the variation in delay between different slot positions; if a component is relocated from Slot 0 to Slot 3, the communication delay between it and the static part of the system will change. To address this issue, the concept of Latency Insensitive Design [14] has been applied to the architecture of the communication infrastructure. Essentially, the infrastructure is transformed into a pipeline; registers are added at the inputs and outputs of all combinational circuits (the multiplexers in this case), and all nets are segmented into smaller sections, connected to each other using registers. When a component is relocated, the result is that the number of registers (pipeline stages) between it and the rest of the system changes, as illustrated in Figure 8.

The conversion of the communication infrastructure into a pipeline works properly in situations where communications occur in one direction (such as stream situations with no hand-shaking protocols). For further experiments the four VHC slot communications infrastructure based on the architecture shown in Figure 6 was implemented in two versions: (a) a VHDL implementation with automated compilation to the configuration bit-stream, and (b) a mixed implementation combining both VHDL and manually placed and routed circuitry (Figure 9).

The reason for two implementations is that the infrastructure timing is critical to the correct operation of the system. The combinational or net delay between registers must be controlled for all connection paths inside the cross bar. The circuit diagram for the manually implemented interconnect is shown in Figure 10. In both cases, the control circuitry was implemented using VHDL, which consists of a finite-state machine and a collection of registers which store the control settings for every multiplexer in the cross bar.

The communication infrastructure of the eight-slot system was designed using a two-level hierarchical structure. Essentially, a nine-port cross bar was implemented using three five-port cross bars. The structure of this composite cross bar is shown in Figure 11. This design was adopted to determine what effect the increased complexity of the communication infrastructure would have on the performance of the system. The implementation of the eight-slot interconnect is a combination of manual and automatic synthesis. As the number of slots being connected rises, the complexity of the infrastructure also increases.

This leads to a very large increase in implementation time when manual placement and routing are performed.

To alleviate this problem, the interconnect was divided into two parts: the three cross bars used to implement the nine-port cross bar, and a dedicated connection bus, which is used to connect the nine-port cross bar to the ports of the VHC slots. The connection bus was implemented manually, while the hierarchical cross bar was implemented using automatic synthesis, placement and routing. The final structure of the interconnect structure is shown in Figure 12. The post-place-and-route circuit of the 8-VHC slot communication infrastructure is shown in Figure 13.

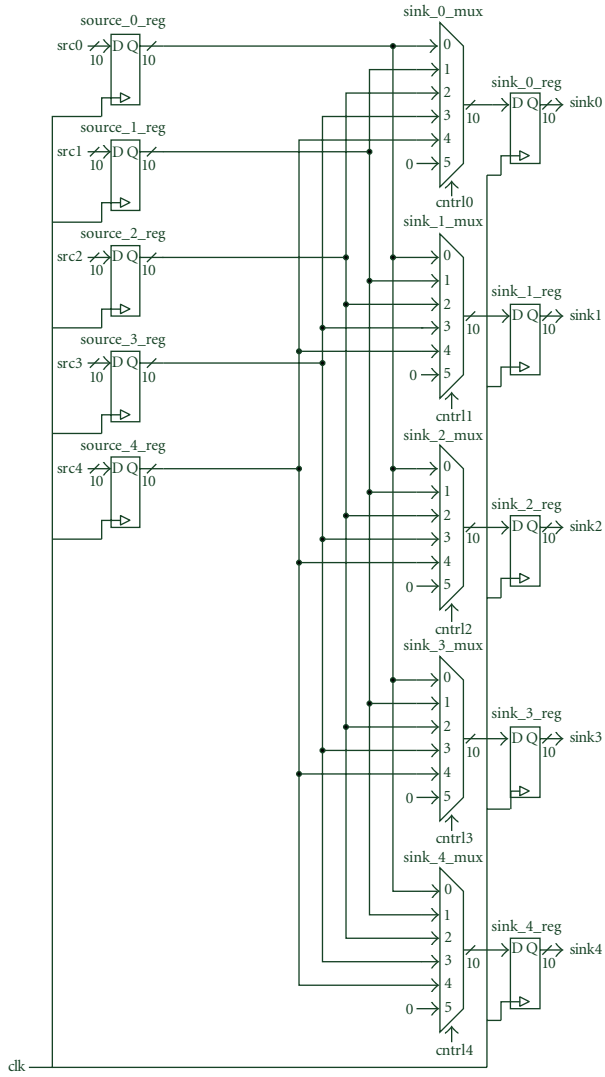


Figure 6: Five-port crossbar switch.

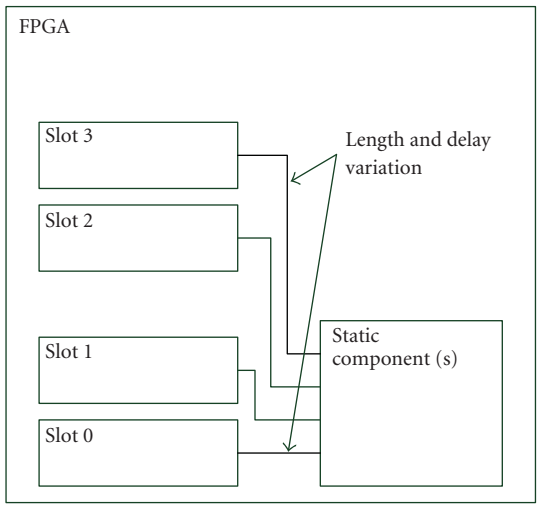


Figure 7: Delay variation in different slots.

4.3. Virtual Hardware Component Organization. This section will concentrate on the architecture organization of VHCs as dictated by the requirements for relocation. However, the internal circuitry of the particular VHCs will not be analyzed in detail here, as that would be outside the scope of this paper. In this consideration VHCs are determined as self-contained data-processing circuits which could meet the partial reconfiguration requirements of Xilinx Virtex FPGA devices [13]. The VHCs are composed of a pipelined data-path, a control unit, and local storage consisting of four RAMB16 blocks [15]. This structure implies that all VHC slots must contain the required memory resources, in addition to the homogenous resources used for general logic. By the same token, however, none of the components require external support ports, a fact which simplifies the design of the communication infrastructure (and the whole system). This approach, however, imposes certain limitation on the application.

This design constraint is adopted only temporarily and was accepted only insofar as it simplifies VHC design at the current stage of research. In further research stages this limitation is going to be removed.

The interfaces used by the components are reduced to the bare minimum needed for the system requirements. Each component has a sink interface and a source interface, the sink acting as the input to the component, and the source as the output from the component. Both the sinks and the ports have the same structure: each port consists of an 8-bit data bus and two synchronization signals. The data bus is used to transfer pixel information from one component to the next; one pixel color sample is transmitted every two clock cycles. The beginnings of each line and of each frame are signaled by the horizontal and vertical synchronization signals, respectively. The communication interconnect described above simply routes these signals as an aggregated 10-bit channel between various slots. This type of communication protocol allows self-synchronization of VHCs during run time relocation. In other words, if the data execution process stops between frames (as indicated by the vertical synchronization signal) VHC relocation can be initiated without data corruption. When the relocation process is complete the relocated VHC(s) can be activated by the same vertical synchronization signal. In video-processing systems the period of VHC relocation can be limited to the interframe period (seamless relocation) or to the period of one frame.

In the case of the video-processing applications considered in our experiments, the mask components are built to automatically synchronize themselves to the rest of the system using the vertical and horizontal synchronization signals. Once the vertical synchronization signal is received by a mask, it will start reading in data, one row at a time. After every row, the mask will stop and wait for the horizontal synchronization signal to arrive, before it will start reading in the next row. At the same time, once two rows have been buffered, the mask component will start generating output information, at the same rate that data is coming in; similar synchronization signals will be generated as outputs for the component downstream. Because of this, a component will stop operating if no vertical and horizontal signals are

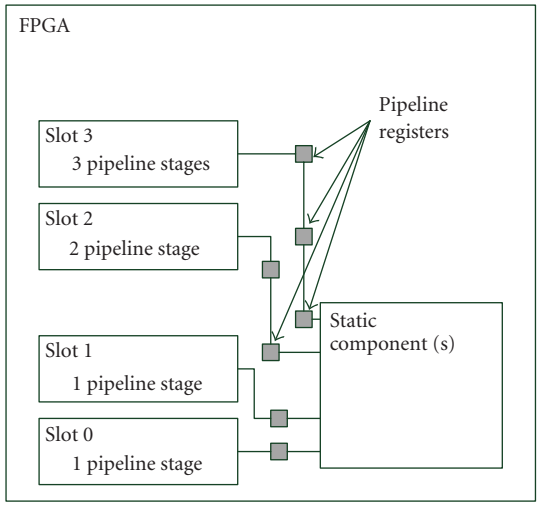


Figure 8: Latency variation in different slots.

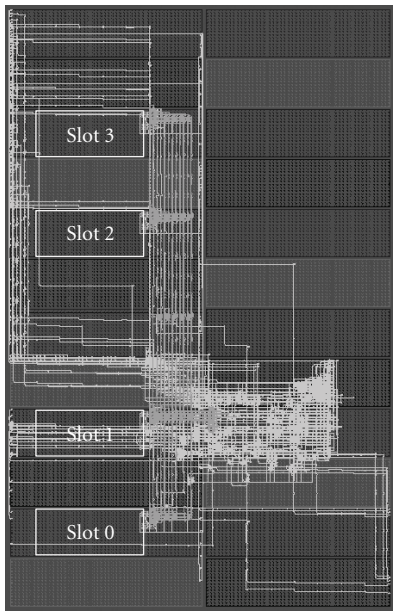


Figure 9: Four-VHC slot interconnect (VHDL and manual implementation).

received. In essence, by driving constant 0 values on the two synchronization signals, the components will become disconnected from the rest of the system, and stall. Similarly, once relocated and reconnected back to the system, the components will wait for the next frame before beginning operation.

It is necessary to mention that self-synchronization of VHCs is not limited to video stream applications. Generally speaking, any application dealing with streamed data can use this approach. This is possible because the data-stream usually has a specific structure which including synchronization periods, packet length, and so forth. However, since the data-stream structure depends on a given application, the self-synchronization mechanism should be customized to address this specific structure.

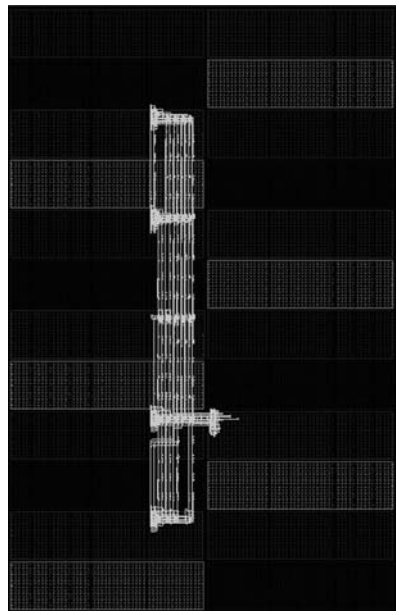


Figure 10: Four-VHC slot Interconnect (VHDL and manual implementation).

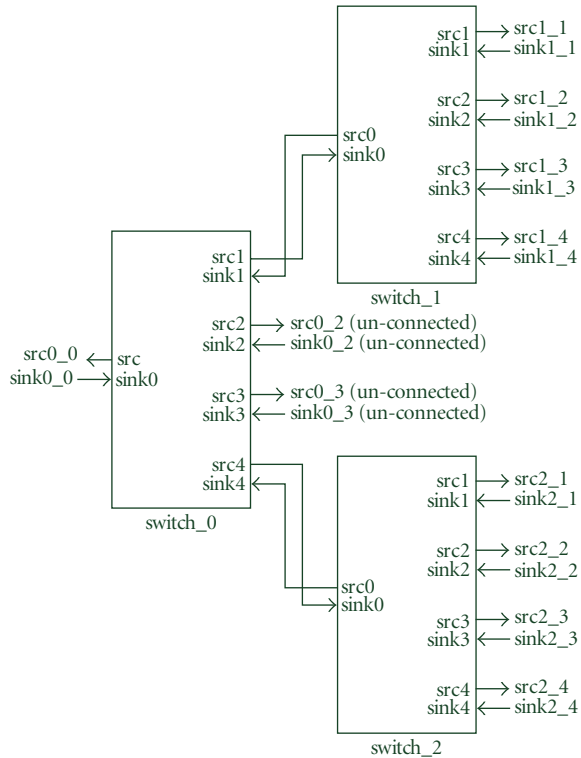


Figure 11: Eight-slot hierarchical switch.

4.4. Relocation Process Analysis. Once the on-chip communication infrastructure and VHC organization have been described, the relocation process can be analyzed in detail. Based on the systems described above, a series of relocation experiments were conducted. These experiments consisted of loading VHCs (masks) into all VHC slots to ensure

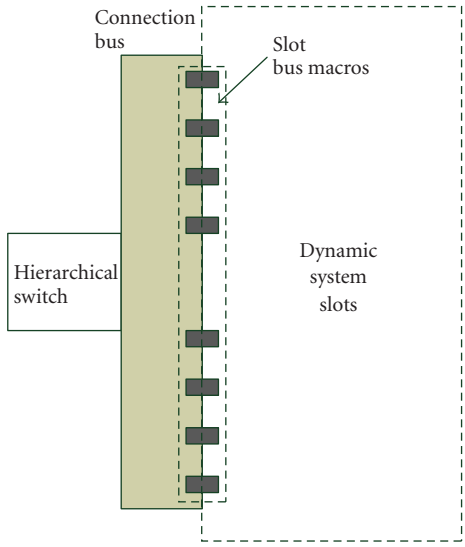


Figure 12: Eight-slot interconnect structure.

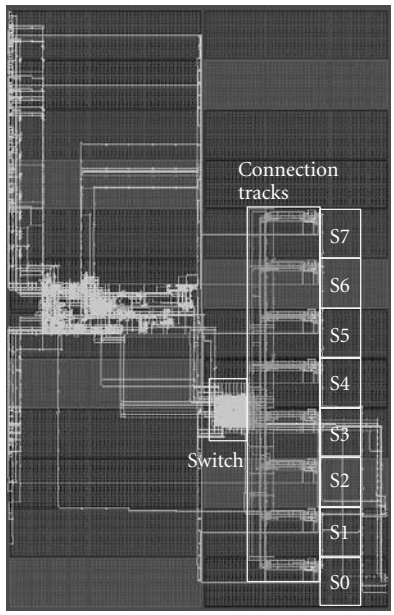


Figure 13: Post-place-and-route circuit of 8-VHC slot interconnects structure.

each mask/slot combination worked. As well, masks were relocated during system operation, to ensure that the process works correctly. For this round of experiments, the approach used was to generate multiple configuration bit streams for every VHC, one for each VHC slot in the system. The results of these experiments consist of timing information regarding the relocation process and the performance of the communication infrastructure, as well as resource utilization figures for the additional infrastructure needed to support VHCs and relocation.

4.4.1. Relocation Process Timing Overhead. The relocation process itself consists of a number of steps. To determine the total timing overhead needed to relocate a component, each step will be analyzed, and the timing information will be extracted. The steps needed for the relocation process are listed below.

(1) Disconnect (and stall) the downstream component from the source of the component being relocated.

(2) Configure the FPGA using the VHC bit-stream targeting the destination slot.

(3) Connect the relocated VHC sink to the source of the upstream component.

(4) Connect the downstream component source to the sink of the relocated component.

Of the above steps, 1, 3 and 4 are identical, essentially consisting of sending control information from the host computer to the reconfigurable platform, and waiting for the communication infrastructure itself to react. For the four-slot system the control information consists of two bytes of information, which are processed by the on-board controller and transmitted to the FPGA system. The time needed to transmit and process a control sequence is of 156 μs. The infrastructure itself requires less than 1 μs to react to the received control information. Steps 1, 3 and 4, require less than 160 μs to complete individually. All three steps together require less than 480 μs. It must be noted that this delay is due in very large part to the communication delay over the serial connection. If the control mechanism resided on-board, and the serial transmission times were eliminated, as much as 85% of this overhead could be eliminated. In the case of the eight-slot system, three bytes of information are transmitted from the host computer to the system for each port being set in the communication infrastructure. Since a hierarchical architecture is used, two ports must be set for each of the three steps described. In total, 422 μs are needed for each step, leading to a total time overhead of 1.27 ms. Once again, this overhead is due primarily to the use of a host computer and serial communication. If an on-board control mechanism were used, and serial communication eliminated, this overhead would be reduced by more than 95%.

The only remaining step to consider is step 2, the partial configuration of the FPGA itself. As mentioned previously, the JTAG configuration interface was initially used for this round of experiments. The sizes of both a complete bit-stream and a partial bit-stream are listed in Table 2. As well, the configuration times associated with each bit-stream are listed. Finally, to show the impact that configuration time has on systems using VHCs, the estimated configuration time for SelectMAP32 configuration is also listed, assuming a maximum configuration frequency of 100 MHz [16]. As can be seen, the configuration time currently dominates the relocation time. However, if SelectMAP32 were used, steps 1, 3 and 4 would require more time than the actual configuration process for a partial bit-stream. In such a situation, using an on-board controller would be beneficial.

Table 2: Bit-stream size and associated configuration time.

	Size (bytes)	JTAG (s)	SelectMAP 32 (ms)
System	5,043,464	6.73	12.6
VHC	90,578	0.12	0.23

Table 3: Communication interconnect hardware resource requirements.

Component	System gates	4-input LUTs	Slice Flip-Flops
INT2	660	30	60
INT4	2,173	154	154
INT4_M	—	394	362
INT8	6,032	569	337
INT8_M	—	569	817

4.4.2. Communication Interconnect Performance. A second set of experiments associated with the timing characteristics of the communication infrastructure were performed. In this case, the aim was to determine how much of a timing limitation the infrastructure would be to a system. This was done by determining the maximum operating frequency (the maximum clock frequency at which the communication infrastructure would still yield correct results). The experiment was conducted by building a test system consisting of the communication infrastructure under test, a simple pattern generator and an on-chip logic analyzer (ChipScope). This test system was implemented as a physical circuit on FPGA and the logic analyzer was used to observe data entering and leaving the communication infrastructure. The clock frequency of the system was increased by constant increments until synthesis could no longer be completed or until incorrect results were observed using the on-chip logic analyzer. It was found that both four-slot infrastructures (manual and automatic) as well as the eight-slot infrastructure were able to operate up to a frequency of 425 MHz. In the case of the eight-slot infrastructure, the two furthest slots in the system (slot 0 and slot 7 in Figure 13) were used to house the pattern generator and the logic analyzer. Despite this, the system offered the same performance as the four-slot system thanks to the use of additional registers in the longer communication paths.

4.4.3. Communication Interconnect Hardware Overhead. To support VHCs, a system must incorporate dedicated infrastructure circuitry, as was shown in the system above. This infrastructure fulfills an important role for the system, while at the same time not contributing anything to the actual processing performed by the system. Because of this, it is important to ensure that the infrastructure itself does not add too large an overhead to the rest of the system. In the case of the systems presented here, the communication infrastructure provides the support infrastructure needed by the system. Table 3 lists resource figures for various implementations of the communication infrastructure; 2, 4 and 8 slot versions are listed. In addition, for the 4 and 8 slot versions, data is provided for both the VHDL-only as well as VHDL and manual placement versions of the circuit. In the table, INT2 refers to a two-slot infrastructure (3 ports), INT4 refers to the four-slot infrastructure and INT8 refers to the eight-slot infrastructure; all these circuits are implemented in VHDL only, and use automatic synthesis. INT4_M refers to the four-slot infrastructure implementation which uses VHDL as well as manual place-and-route. INT8_M refers to the eight-slot infrastructure implementation using VHDL and manual place-and-route.

To determine how large the infrastructure variations are we can compare them with one of the system slots. One slot in the four-slot architecture contains 1,664 slices, or 3,328 LUTs and 3,328 1-bit flip-flops. In addition to this, each slot contains eight RMAB16 blocks and four DSP blocks; however, since the infrastructure variations do not use these resources, they will not be considered in this analysis. The most resource-heavy infrastructure in the above table is the eight-slot variation implemented using both VHDL and manual place-and-route. Nonetheless, when compared with the resources available in one system slot (which takes up less than one sixteenth of the device), we see that the communication infrastructure would account for only 21% of the total resources available in one VHC slot (counting all LUTs and slice flip-flops together) or less than 2.7% of the total resources reserved for all eight slots.

Summarizing the above, the proposed organization of on-chip communication infrastructure can provide run time adaptation of the FPGA-based stream processing system on the current set of tasks (current workload) as well as run time mitigation of hardware faults caused by various factors. On the other hand the cost-effectiveness of multimode/multistream processing systems can be increased by reducing power consumption compared with existing design methodologies. To make possible the above features the following aspects have been analyzed and tested: (i) flexible allocation and run time on-chip relocation of virtual hardware components, providing a mechanism for run time on-chip assembling of ASVPs; (ii) an operational frequency range up to 425 MHz for both 1-level (4-VHC slot) and 2-level (8 and 16 VHC slot) hierarchical organizations of the communication infrastructure in tile-based Xilinx Virtex FPGA devices; (iii) seamless relocation of VHCs using the mechanism of self-synchronization embedded in the VHC architecture; (iv) relatively small hardware overhead for the communication infrastructure (not exceeding 2.7% of resources allocated for 8-VHC slot structures).

All of the above gives us the strong belief that the proposed approach can be an effective solution for a wide variety of applications associated with high data rate processing of streamed data. Some of these applications will be discussed in the next section.

5. Industrial Applications

The presented research was oriented towards several industrial applications which required high data rate processing in many different modes. Furthermore, high reliability and low power consumption were important requirements for

these applications. In addition to that, minimal dimensions and weight (mass) were necessary. One of these applications is the next generation of space-born computing platforms for orbital and interplanetary missions. The initial focus was on high-frame rate machine-vision platforms for automated docking of satellites as well as automated object grasping using space manipulators (e.g., CanadArm-2 on ISS). This type of applications requires (i) multimode parallel video stream processing with high-frame-rate requirements (from 30 to 200 fps), (ii) automated switching from one mode to another within a relatively short time (milliseconds), (iii) parallel processing of several tasks allowing run time distance measurement, object tracking, image stabilization, video-compression and many other algorithms, (iv) minimum mass, dimensions, power consumption and power dissipation, (v) the ability to mitigate transient hardware faults caused by cosmic radiation effects (e.g., SEU—Single Event Upset) as well as restoration from permanent faults. Additionally, the platform architecture should allow remote upgrade of hardware and software components and rapid adaptation for new task algorithms and data structures.

The above requirements are usual for any autonomous embedded system oriented to work in a harsh environment (e.g., control and robotic systems for nuclear power stations, etc.).

Similar platform specifications can be found for satellite communication and broadcasting systems. Recently, these systems are oriented on broadcasting digital video streams on mobile and handheld devices. This requires utilization of complex COFDM modulators, which usually are developed around special chipsets and a number of statically configured large FPGA devices. The example here is DVB-S/H (Digital Video Broadcasting—Satellite-to-Handheld) systems. It is necessary to mention that the MARS platform was developed and manufactured as part of an RandD project oriented towards the creation of the next generation of platforms for DVB-S/H COFDM modulators.

Most recently, our implementation efforts of the above research are concentrated on the creation of the next generation of space-borne platforms for machine-vision. Several stereo vision sensors and associated VHCs have been developed and tested during a Collaborative RandD project associated with high-frame rate tracking of 3D objects. A special experimental setup has been recently designed and assembled on the basis of a 5-axis space platform where the high-frame-rate stereo vision sensor was installed for experiments on automated pose recognition and tracking. This setup is shown in Figure 14.

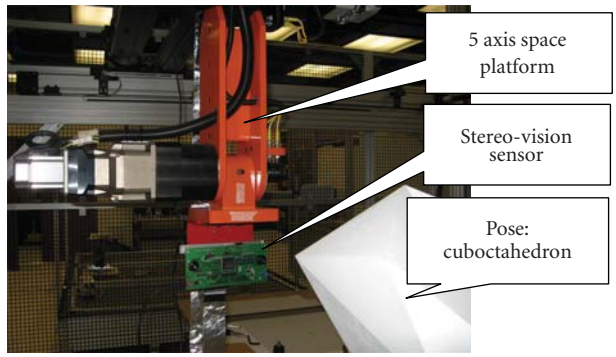


Figure 14: Experimental setup for autonomous docking of space structures.

6. Summary

A mechanism for the virtualization of computing resources for multitask and multimode stream applications has been developed and investigated in detail in this work. The major concept of the proposed approach is to assemble application specific virtual processors (ASVPs) at run time inside the FPGA rather than the traditional approach based on off-line compilation. The ASVP can be assembled using a field of homogenous computing resources in the FPGA and a configuration which reflects the current set of tasks and their modes. The building blocks for the ASVP are virtual hardware components (VHCs)—macro-function-specific stream processing pipelines. The VHCs are self-containing modules with embedded SRAM blocks, control FSM and synchronization circuits. A VHC becomes a real processing module only when configured in a specific slot of the target FPGA. Otherwise, the VHC is stored in one of multiple levels of a memory hierarchy: VHC cache, main VHC-memory or secondary storage. This organization of the system level architecture allows a dramatic minimization of actual hardware resources and power consumption as well as cost and dimensions of the RCS. The comparative analysis made on the multitask adaptive reconfigurable system (MARS) demonstrated negligible hardware overhead offset by sufficient gains in the main performance parameters. This experimental platform incorporates all required components needed to provide hardware support for virtualization of FPGA resources. At the same time, the on-chip communication infrastructure has been designed to provide the same hardware support at the FPGA microarchitecture level. Different schemes and cross bar switch configurations have been analyzed and a hierarchical organization (2 levels) of the communication infrastructure with pipeline transition registers was selected. It was experimentally determined that this system can support operating frequencies up to 425 MHz and eliminates propagation delay variations even between the farthest slots. At the same time, utilization of self-synchronization circuits in the VHCs allowed seamless run time component relocation which provides more flexibility for a system. This same relocation procedure allows rapid restoration of any transient faults as well as mitigation of permanent hardware faults. The above relocation procedures have been experimentally tested using different FPGA ports for configuration as well as on-board/external control mechanisms. It was determined that using on-board configuration controllers in conjunction with parallel configuration ports of the target FPGA, the relocation process could be done in a relatively short time (less than 1 ms). The on-chip hardware overhead was also analyzed in comparison with the amount of resources available for VHC slots. In all experiments the on-chip communication

infrastructure never exceeded an overhead of 2.7% compared with the resources dedicated to data-processing.

Finally, the proposed mechanism has been implemented for different industrial applications and a high potential for increasing system cost-efficiency has been demonstrated. The major applications for the above approach are mobile autonomous embedded systems where multiple tasks associated with streamed data should be performed, each with high data-rate requirements.

Acknowledgment

The authors wish to acknowledge the support of NSERC of Canada, Canadian Space Agency (CSA), Ontario Centers of Excellence (OCE-CITO), MDA Space Missions, UBS Ltd., CMC Microsystems, and Xilinx Inc.

References

[1] E. Caspi, M. Chu, R. Huang, J. Wawrzynek, J. Yeh, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE)," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pp. 605–614, Springer, Berlin, Germany, 2000.

[2] S. Wallner, "A reconfigurable multi-threaded architecture model," in *Advances in Computer Systems Architecture*, pp. 193–207, Springer, Berlin, Germany, 2003.

[3] M. Mishra and S. C. Goldstein, "Virtualization on the Tartan reconfigurable architecture," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 323–330, Amsterdam, The Netherlands, August 2007.

[4] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[5] L. Kirischian, V. Geurkov, V. Kirischian, and I. Terterian, "Multi-parametric optimisation of the modular computer architecture," *International Journal of Technology, Policy and Management*, vol. 6, no. 3, pp. 327–346, 2006.

[6] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, "REPLICA: a bitstream manipulation filter for module relocation in partial reconfigurable systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pp. 151–158, Denver, Colo, USA, 2005.

[7] F. Ferrandi, M. Novati, M. Morandi, M. D. Santambrogio, and D. Sciuto, "Dynamic reconfiguration: core relocation via partial bitstreams filtering with minimal overhead," in *Proceedings of the International Symposium on System-on-Chip (SOC '06)*, pp. 1–4, Tampere, Finland, 2006.

[8] S. Corbetta, F. Ferrandi, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto, "Two novel approaches to online partial bitstream relocation in a dynamically reconfigurable system," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI: Emerging VLSI Technologies and Architectures (VLSI '07)*, pp. 457–458, Porto Alegre, Brazil, March 2007.

[9] T. Becker, W. Luk, and P. Y.K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Proceedings of the IEEE Symposium on Field-Programme Custom Computing Machines (FCCM '07)*, pp. 35–44, Napa, Calif, USA, 2007.

[10] M. Hübner, C. Schuck, M. Kühnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, vol. 2006, pp. 97–102, Karlsruhe, Germany, 2006.

[11] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, pp. 37–42, Singapore, December 2005.

[12] L. Kirischian, V. Geurkov, I. Terterian, V. Kirischian, and J. Kleiman, "Multilevel radiation protection of partially reconfigurable field programmable gate array devices," *Journal of Spacecraft and Rockets*, vol. 43, no. 3, pp. 523–529, 2006.

[13] Xilinx, "Early access partial reconfiguration user guide, 1.2 edn," 2008.

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[15] Xilinx, "Virtex 4 FPGA user guide, 2.5 edn," 2008.

[16] Xilinx, "Virtex-4 FPGA configuration user guide, 1.1 edn," 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 652620, 16 pages
doi:10.1155/2010/652620

Research Article

Traversal Caches: A Framework for FPGA Acceleration of Pointer Data Structures

James Coole and Greg Stitt

Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611-6550, USA

Correspondence should be addressed to Greg Stitt, gstitt@ece.ufl.edu

Received 9 March 2010; Revised 10 October 2010; Accepted 13 December 2010

Academic Editor: Viktor K. Prasanna

Field-programmable gate arrays (FPGAs) and other reconfigurable computing (RC) devices have been widely shown to have numerous advantages including order of magnitude performance and power improvements compared to microprocessors for some applications. Unfortunately, FPGA usage has largely been limited to applications exhibiting sequential memory access patterns, thereby prohibiting acceleration of important applications with irregular patterns (e.g., pointer-based data structures). In this paper, we present a design pattern for RC application development that serializes irregular data structure traversals online into a traversal cache, which allows the corresponding data to be efficiently streamed to the FPGA. The paper presents a generalized framework that benefits applications with repeated traversals, which we show can achieve between 7x and 29x speedup over pointer-based software. For applications without strictly repeated traversals, we present application-specialized extensions that benefit applications with highly similar traversals by exploiting similarity to improve memory bandwidth and execute multiple traversals in parallel. We show that these extensions can achieve a speedup between 11x and 70x on a Virtex4 LX100 for Barnes-Hut n-body simulation.

1. Introduction

Numerous studies have shown that field-programmable gate arrays (FPGAs) and other reconfigurable computing (RC) devices can achieve order of magnitude or larger performance improvements over microprocessors [1, 2] for application domains including embedded systems, digital signal processing, and scientific computing. In addition to providing superior performance, FPGAs have also been shown to improve power consumption and energy efficiency compared to microprocessors and alternative accelerator technologies such as graphics processing units (GPUs) [3].

The advantages of FPGAs result from the ability to implement custom circuits that exploit tremendous amounts of parallelism, often using deep pipelines with additional parallelism ranging from the bit level up to the task level. As a consequence of enabling large amounts of parallelism, however, FPGA circuits require high memory bandwidth to avoid frequent pipeline stalls that can prevent potential speedups from being realized [4].

One significant limitation of FPGAs is that due to the requirement for high memory bandwidth, FPGAs are typically unable to accelerate applications with irregular access patterns [5]. In this paper, we define an irregular access pattern as a sequence of memory accesses that are not sequential in memory, or patterns that cannot be buffered based on compile time analysis [6]. Although irregular access patterns can result in many different ways, in this paper, we focus on the common example of pointer-based data structure traversals, such as lists and trees. Traversals of pointer-based structures reduce effective memory bandwidth through indirection, which requires multiple memory accesses to fetch a single item of data. In addition, consecutively accessed nodes of pointer-based structures are rarely stored at consecutive memory locations, resulting in poor cache performance and requiring frequent expensive row address strobes (RAS) in SDRAM memories [7]. Nonsequential accesses also prevent the use of specialized burst memory access modes [7] that provide maximal memory performance.

To enable usage of FPGAs on more pointer-based applications, we present a framework for improving memory bandwidth of traversals of pointer-based data structures. As illustrated in Figure 1, the presented framework dynamically serializes data accessed during traversal of

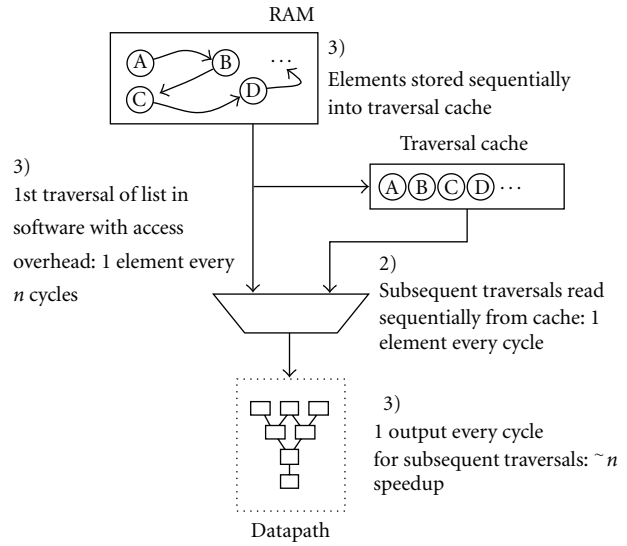


Figure 1: Conceptual idea of *traversal caches*, where repeated traversals of pointer-based structures are stored sequentially to improve memory bandwidth to custom circuits.

large pointer-based data structures and stores the serialized representation in a local memory, referred to as a traversal cache, where it can be more efficiently accessed by the FPGA. By serializing the elements accessed in a traversal, traversal caches eliminate the bandwidth reduction caused by indirection and nonsequential data storage while also enabling memory burst modes.

The simplest use of traversal caches involves storing repeated traversals of a pointer data structure in the cache memory. When the cache is larger than the typical traversal, multiple traversals may be stored at different locations in the cache to reduce the miss rate, with software managing placement and bookkeeping. This usage model is most analogous to traditional caches, with traditional cache lines corresponding to entire traversals.

In this repetition-based model, the concept of a cache hit corresponds to an FPGA repeating an identical traversal that is already stored in the cache. Traversal cache misses result from two situations. First, if an application requires a traversal not currently stored in the cache, the framework must first load the corresponding traversal (i.e., a conflict or compulsory miss). Second, if any part of the application modifies the data involved with a traversal stored in the cache, the cache representation must be marked invalid, resulting in a compulsory miss on the next access. Therefore, the overall performance improvement resulting from the use of a traversal cache is highly dependent on the invalidation rate. For applications with high invalidation rates (or otherwise low repetition), the framework provides the least benefit, behaving similarly to a system with no traversal cache with additional overhead due to frequent data structure serialization. In Section 5, we show that these techniques can achieve large speedups for applications with different rates of repeated traversals and are capable of matching sequential-access (array-based) software in some situations.

Though this model is simple and generic, it achieves limited speedup for applications that never or very rarely have identically repeated traversals resulting in a high invalidation rate and constant thrashing. However, even applications that rarely repeat a traversal often exhibit a large similarity between traversals. This can be the case for applications, where a large data structure remains static over a (relatively) long period of time and traversals occur in the same order through the data structure (e.g., preorder for a tree). In these situations, we show that additional data can be included in the traversal cache to describe the data structure itself, allowing hardware to generate the multiple traversals from the cache without additional software intervention.

This similarity-exploiting use model has the added benefit of allowing hardware to generate and process multiple traversals in parallel, which can enable a large amount of data reuse when the similarity between traversals is high, resulting in additional speedup. This approach to traversal caches also handles repeated traversals as a special case, automatically handling the repeated traversals in parallel, enabling for example greater loop unrolling by improving access to memory. In this paper, we evaluate these extensions exploiting similarity between traversals in the Barnes-Hut n-body simulation algorithm [8] and discuss how the framework could be used for other applications. The experimental results show that the memory bandwidth bottleneck is almost completely eliminated for highly similar traversals, resulting in kernel speedup that increases approximately linearly with area. In Section 5, we show speedups from 11x to 70x for instances of the Barnes-Hut n-body simulation algorithm, which essentially never repeats traversals.

Besides requiring a high level of similarity between traversals, current implementations require manual creation of the traversal cache, including application-specific logic in the case of the similarity-based framework and modification of existing application codes to perform cache management. These steps are mostly data structure specific, enabling the creation of traversal cache compatible libraries that can be used by multiple applications; however, future work will introduce techniques for performing these tasks as part of high-level synthesis. The frameworks also generally assume that at least one entire traversal fits in the traversal cache. Since the cache is typically implemented using off-chip SDRAM (commonly available on commercial FPGA accelerator boards or shared with the host processor), this assumption is not unreasonable. Otherwise, ensuring this condition is generally a matter of partitioning the application to run within the platform's resource constraints, as would also be required by multinode computation typical for such large problems. Other considerations are discussed where appropriate later in this document.

The paper is formatted as follows. Section 2 discusses previous work. Section 3 describes the traversal cache framework supporting repeated traversals. Section 4 presents a generalization of the framework to support parallel processing of similar traversals through a case study on Barnes-Hut n-body simulation. Section 5 presents experimental results. Conclusions and future work are discussed in Section 6.

2. Previous Work

Previous work has investigated hardware synthesis techniques for code utilizing pointers. In [9], Semeria integrated alias analysis techniques into a high-level synthesis tool flow to help resolve aliases at compile time, thus enabling further optimization and utilization of multiple memories. These techniques were later extended in [10] to support dynamic memory allocation by integrating a memory manager into the synthesized circuit. The traversal cache framework is a complementary approach that targets hardware/software codesign by not restricting the hardware to using a separate memory management unit—a situation that may not be practical or efficient for all FPGA accelerators.

Diniz and Park [5] utilized FPGAs to create smart memory engines capable of reorganizing data from pointer-based data structures to improve data locality and cache performance. The traversal cache framework has a similar goal, but does not reorder data in main memory, and thus does not have the alias restrictions of [5]. Furthermore, the traversal cache framework can be applied to any FPGA accelerator, including those that access memory via DMA.

Impulse [11] introduced a memory controller that remaps physical addresses to improve cache performance and memory bandwidth. Impulse remapped data using specialized languages and operating system support. The traversal cache framework does not have these restrictions, and only requires use of a specific library.

Specialized cache architectures and memory allocation techniques have also been introduced to better handle pointer operations. Collins et al. [12] introduced the pointer cache to efficiently handle chained pointer traversals by prefetching data based on pointer transitions. Weinberg [13] eliminates pointer-based memory accesses at runtime by caching previous evaluation results. Hu et al. [14] predicts memory behavior and utilizes a time-based victim cache to improve hit rate. Chilimbi et al. [15] discuss manual programming practices that can improve data locality of pointer structures, in addition to automatic data layout optimizations that are integrated into garbage collection. Calder et al. [16] considers cache-conscious data placement for heap and stack objects. The traversal cache framework provides similar optimizations for FPGAs, which commonly have direct access to memory and, therefore, do not benefit from traditional cache optimization.

Smart buffers [6] are a data-caching scheme for FPGAs that prevent reused data from being read multiple times from memory. Smart buffers greatly improve memory bandwidth and FPGA performance but do not support pointer-based data structures.

Numerous compiler optimizations [7, 17, 18] modify data layout at compile time based on memory access patterns. Traversal caches improve on these previous approaches by supporting pointer-based data structures. Baradaran and Diniz [19] and some parallelizing compilers [20] also consider mapping accesses to multiple FPGA-internal memory resources, improving parallelism. The similarity-based traversal cache also optimizes for parallelism but seeks it through access coalescing, which enables efficient use of large external memories such as SDRAMs that are optimized for sequential accesses.

Traversal caches were originally introduced in [21], which presented a general framework that exploits identically repeated traversals. In [22], the approach was extended to exploit similarity. This paper integrates the contributions of previous work and evaluates the framework for additional examples.

3. Traversal Cache Framework

In this section, we present the traversal cache framework for caching and reusing repeated traversals. Section 4 discusses extensions for exploiting nonidentical traversals with high similarity. The following subsections discuss the functionality required to enable the traversal cache framework, which includes the system architecture, hardware/software communication, and traversal cache software library.

3.1. System Architecture. Figure 2 illustrates the system architecture used by the traversal cache framework, which consists of two main components: the microprocessor and the FPGA accelerator. The microprocessor executes all software regions of the code and assists in fetching traversals whenever the traversal cache is empty or invalidated. The FPGA accelerator implements custom circuits to speed up computationally intensive kernels of the targeted application, typically implemented as deeply pipelined datapaths, which read data from the traversal cache to efficiently handle pointer-based data structures. Future work will look at automatically partitioning traversal fetch and serialization logic as part of high level synthesis.

In addition to the processor's main memory, the framework assumes two separate memories local to the FPGA that are used to simultaneously stream data in and out of the accelerator—a common architecture in commercially available FPGA accelerator cards and other accelerators. The microprocessor also has read/write access to the FPGA's memories, either through an external controller or a controller implemented by the FPGA. Communication details are discussed in the next section.

Details of the FPGA accelerator are shown in the expanded box in Figure 2. The controller interprets messages from the microprocessor and enables the address generators when the microprocessor activates the accelerator. The address generators control the input and output memories to read a stored traversal or write datapath results. Since elements within a traversal are stored in order in the traversal cache, the address generators typically need only to support linear access patterns, possibly with a fixed stride. In other situations, more complicated address generators with buffering [6] can be substituted. The datapath is a pipeline implementing an application kernel, which can usually be isolated from the specifics of the framework allowing reuse from previous design efforts or the use of existing HLS techniques.

The input memory hierarchy used by the accelerator consists of three possible data sources. The accelerator uses

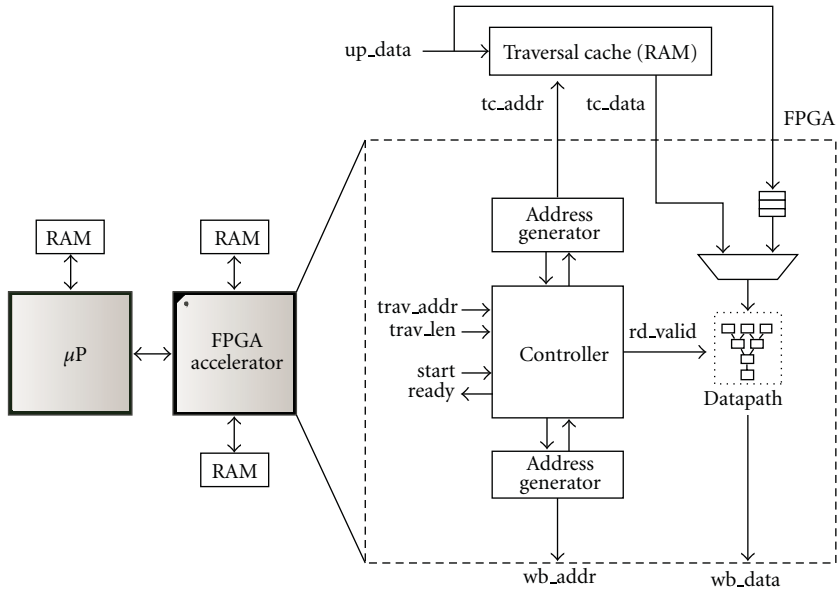


Figure 2: System architecture of the generalized traversal cache framework. On the first of a repeated traversal of a pointer-based data structure, software sends the elements to the FPGA on *up_data*, which is simultaneously written to the cache memory and consumed by the accelerator datapath. On subsequent traversals, the accelerator reuses the sequence of data in the cache, which can be read sequentially using simple constant stride address generators.

one of the two external RAMs shown in Figure 2, leaving the other for buffering results. In our implementation, processor writes to the FPGA's traversal cache (RAM) are handled by logic on the FPGA itself, allowing incoming data to be used directly while a traversal is being transferred to the traversal cache, which can help mitigate the effect of cache misses.

There is obviously flexibility in this architecture for systems with different needs. Our implementation uses SDRAMs for the traversal cache (read memory) and results buffer (write memory); however, systems with more modest memory requirements might be able to use external SRAM or distributed memory on the FPGA itself. The high latency of nonsequential SDRAM accesses impacts the framework's performance for similar traversals (discussed in Section 6) so that the use of lower latency memories could also result in performance improvements. Although not a focus of this paper, potential optimizations in this area are discussed in more detail in Section 4.

Systems with tighter integration between the FPGA and CPU could also read or write directly from main memory, with the traversal cache occupying a dedicated section of main memory. In such tightly coupled systems, the performance overhead involved with copying traversals to the FPGA's memories would be reduced, possibly improving speedup for applications with shorter or less frequently repeated traversals.

3.2. Hardware/Software Communication. Communication between the microprocessor and FPGA occurs through the signals shown on the left side of the controller in Figure 2. For simplicity, we do not show signals or control logic for moving data between the microprocessor and local FPGA memories (which could also be external to the FPGA). Other application-specific signals used to set up the datapath, for example, selecting which calculation to perform during this traversal, are also not shown.

When the microprocessor reaches a kernel requiring traversal of a pointer-based data structure, software makes a determination about whether an up-to-date serialized version of the traversal already exists in the cache. This determination is left to software, since it may be arbitrarily complex and is frequently the result of modifications to the data structure, which is also likely to be a complicated operation and, therefore, best handled in software.

If the cache does not contain the current traversal, representing a cache miss, the microprocessor transfers a serialized version of the traversal to the cache (mediating controller not shown), placed at an address in the cache chosen by software given by *trav_addr*. In the process of loading the cache, the accelerator's controller is notified to expect the valid traversal elements to begin appearing on the buffered processor write bus *μp_data*, and the controller is primed by asserting *start*. Once the first elements appear on *μp_data*, the controller manages the address generators and datapath, stalling the datapaths as necessary for new elements to arrive from the microprocessor while writing elements into the cache. In the unlikely event that the transfer rate exceeds the datapath's rate of consumption, the *μp_data* FIFO will eventually overflow. Upon overflow, the *μp_data* path is disabled after flushing the FIFO, and the controller begins sourcing data from the traversal cache memory instead, beginning with the element that overflowed in the FIFO (address known to the controller by observing the FIFO's overflow signal).

In the case of a cache hit, software tells the controller to begin processing the traversal from address *trav_addr* and asserts *start*. Elements of the traversal are simply read out of the cache in order, starting at address *trav_addr* and ending at address *trav_addr* + *trav_len*.

Once all the elements in the traversal have been processed through the datapath, the accelerator is complete and notifies the controller by asserting *ready*, after which the processor can safely issue new work for the accelerator. Our implementation also provides a signal to the microprocessor with the results memory's current fill state (not shown), which can allow software to overlap additional processing of the results with lengthy calculations on the accelerator in some applications.

The framework currently implements all control and synchronization signals using memory mapped registers inside the FPGA. Communication currently occurs over a PCI-X interface. Note that the framework is independent of the communication architecture, potentially supporting any underlying architecture that can implement the discussed control signals.

3.3. Software Library. To utilize the traversal cache, the accelerator requires assistance from a software library running on the microprocessor. The library is responsible for specifying when a traversal occurs, detecting invalidations of a traversal, and passing data to the accelerator when the traversal cache is empty or needs different or updated traversals. We currently implement this functionality using a library of wrapper functions around standard data structures. This process could also potentially be automated as part of a high-level synthesis and hardware/software partitioning tool, which could theoretically allow a user of the framework to utilize any library.

To specify a traversal to be processed by the accelerator, the wrapper functions first check if a valid serialization already exists in the traversal cache, only refetching and serializing traversal elements if the traversal is not already in the cache. The library uses a different wrapper function for each type of traversal, such as an in-order tree traversal, a depth-first search of a graph, etc. The most challenging task required by the software is to detect traversal invalidations. Currently, the framework invalidates traversals when elements in active traversals (traversals whose serializations are currently in the traversal cache) are changed. This often requires that the data structures be augmented with additional data identifying membership in currently active traversals. Detecting changes to the data structure is a challenging problem in general due to aliasing issues that may exist in the code. To avoid these issues, the framework requires that any changes made to the data structure be made through the use of the wrapper functions. This requirement guarantees that the traversal cache will be invalidated for any modification to the data structure. Furthermore, this requirement does not restrict the use of aliases outside the wrappers because those aliases cannot modify the structure.

Since multiple traversals might be kept active (in the traversal cache) at the same time, in the hope that they will be reused later, the libraries also do bookkeeping to track which traversals are in the cache and where they are located. This data is also required for placing traversals when being added to the cache and for carefully evicting individual traversals as necessary. This cache-management functionality is needed by all libraries and can be implemented in a way that allows sharing between libraries for different data structures. This is currently implemented as a mapping from library-defined keys (usually incorporating order and other data that uniquely generates the corresponding traversal) to the base address and extent of that traversal in the cache.

The penalty for evicting the wrong traversal (equal to the time required to generate and add a traversal) is high compared to traditional caches, and the total number of active traversals tends to be low, since accelerators are more effective on longer traversals. This suggests that otherwise expensive eviction strategies might be useful for traversal caches. Direct management of the cache by libraries also allows for the use of strategies based on an individual algorithm's dynamics. Traversals also generally vary in length and must be allocated in the cache contiguously, complicating placement and leading to fragmentation issues. Here too, the cost of a poor policy outweighs the relatively small cost associated with even complex strategies like heap compaction. In this paper, we evaluate different possibilities abstractly by using an invalidation rate (IR) which includes the effects of any such strategies. Future work will address these issues in more depth.

3.4. Limitations. The main limitation of the general traversal cache framework is that not all applications using pointer-based data structures are amenable to speedup. To achieve speedup typically requires that an application have frequently repeated traversals. However, as shown in Section 5, for some applications, speedups are possible even with relatively few repetitions. In fact, for some computation-intensive applications, speedup can be obtained even if the traversal cache is invalidated for every traversal due to the speedup provided by the accelerator's deeply pipelined datapaths. Furthermore, as discussed in Section 4, the framework can be extended for some applications to support traversal of similar but nonidentical traversals without suffering the penalty incurred by invalidations.

Another limitation is that the traversal cache must be manually created and a specialized software library must be used, or existing code modified for new applications. Ideally, a hardware/software partitioning tool could partition the application automatically, high-level synthesis could determine the appropriate size and amount of traversal caches, and also modify the software source code appropriately for use with any data structure library. These issues are outside the scope of this paper, but we plan to introduce synthesis techniques for traversal caches as part of future work.

In situations where the local FPGA memory is too small to store an entire traversal, software could load only the first part of the traversal, paging in later parts as needed by the accelerator. Though this would impact performance, depending on how fast the accelerator consumes data out of

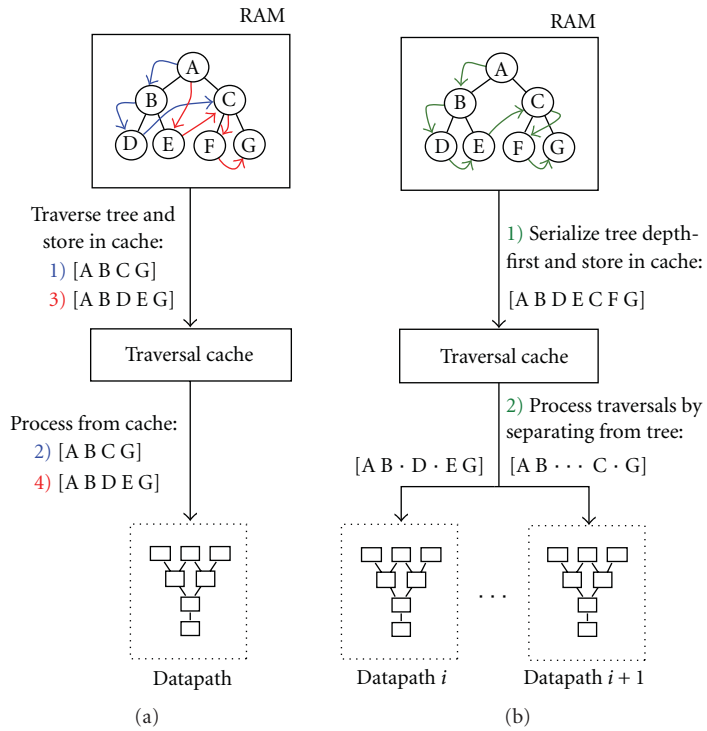


FIGURE 3: (a) Despite large amounts of similarity, nonrepeated traversals result in a 100% miss rate using the general framework, transferring a new traversal each time. (b) Extensions to the framework exploit the similarity between traversals by storing the entire tree in the cache and separating out individual traversals while streaming, also enabling datapath replication. (Elements excluded from separated traversals are shown as $(\cdot)$.)

the cache, at least some of the time required for paging could be hidden by consuming data in stream, as is done when the cache is populated with a new traversal. For example, in Section 5.1, we present a speedup of 15.5x for a linear search benchmark (*search,* IR = 20). Reducing the size of the traversal cache so that only one half of the traversal fits at a time and paging in the second half reduces the speedup to 8x without any effort to overlap computation paging and computation.

4. Exploiting Traversal Similarity: A Case Study on Barnes-Hut

The previous section discussed a generalized traversal caching framework that improves memory bandwidth for applications with repeated traversals. Unfortunately, many applications do not have this characteristic, limiting the achievable speedup. However, some important applications that lack sufficient repeated traversals have or can be made to have large amounts of similarity between traversals over time, including many applications that involve searching trees. In this section, we discuss modifications to the traversal cache framework to exploit this similarity in order to improve memory bandwidth for a broader range of applications.

Unlike the general framework in the previous section, the extended version of the traversal cache framework relies on the cooperation of hardware in generating traversals for processing in the accelerator's datapaths. This tighter integration of hardware and software provides new opportunities for optimization, but it has the effect of requiring application-specific control logic in the accelerator, which increases the complexity of accelerator design and hardware/software partitioning.

We evaluate these extensions for a type of Barnes-Hut n-body simulation application, summarized in Section 4.1. Barnes-Hut is a motivational example for the traversal cache framework because for n-body simulations using FPGAs, Barnes-Hut is typically avoided due to irregular memory access patterns resulting from its use of a quad- or octtree. FPGA implementations instead usually use a straightforward $O(n^2)$ implementation that does not require a tree data structure. Although this straightforward implementation enables highly parallel FPGA execution, the reduced $O(n \log n)$ complexity of Barnes-Hut often negates any performance advantage of the FPGA accelerated implementation. One key result of the traversal cache framework is that it can enable highly parallel FPGA implementations of the $O(n \log n)$ versions of Barnes-Hut as well as other tree-based algorithms.

4.1. Overview. Figure 3 compares the general traversal cache framework with the extensions for exploiting similarity. In the general framework shown in Figure 3(a), two different (but highly similar) traversals of the pointer-based data structure would have required two invalidations, causing

each new traversal to be serialized in software, stored in the traversal cache, and then streamed to the datapath in the FPGA. The extended framework exploits the similarity between the traversals by combining the multiple traversals into a data structure (in this case the entire tree serialized in depth-first order) that is stored in the traversal cache as shown in Figure 3(b). The FPGA can then stream the data from multiple traversals into datapaths by using additional logic, discussed in more detail in Section 4.4, to separate the elements needed by each traversal into the corresponding datapath. Since multiple traversals can be separated out from the same stream, the figure shows how these extensions further improve performance by enabling the two traversals to be processed in parallel in replicated datapaths.

In general, the improvement in memory bandwidth and the corresponding increase in useful datapath replication depend on the similarity between traversals. For circuits with n replicated datapaths performing n parallel traversals, the memory bandwidth provided by the framework becomes close to n times higher than the physical memory as the similarity between the traversals approaches 100%. Therefore, the framework can enable near-linear performance increases for linear increases in area when applications have highly similar traversals—a situation that overcomes the common memory bandwidth bottleneck of many FPGA applications [4, 23].

4.2. Barnes-Hut N-Body Simulation. N-body simulation is a common scientific computing problem that simulates the movement of n bodies under the influence of physical forces (e.g., gravitational or electrostatic forces). N-body simulation is used in a key step for a number of scientific applications including molecular dynamics, cosmological simulations, and problems in statistical learning.

A straightforward algorithm for an n-body gravity simulation consists of computing the forces exerted on each body by all the other bodies, which has a time complexity of $O(n^2)$. The Barnes-Hut algorithm [8] can reduce this complexity by treating groups of distant bodies as a single body centered at the corresponding center of mass. The algorithm creates this approximated solution by recursively subdividing space into a quadtree or octtree, with internal nodes representing an average of its leaf bodies (e.g., center of mass or electric multipoles). For concentrated collections of distant objects, set by a threshold ratio *theta*, the force is computed due to these average nodes instead of the individual bodies. Depending on the value chosen for *theta*, Barnes-Hut has a complexity ranging from $O(n \log n)$ to $O(n^2)$, with $O(n \log n)$ versions for *theta* < 0.5 being common in practice [8].

Pseudocode for the Barnes-Hut n-body algorithm is shown in Algorithm 1. The input is the number of time steps to simulate in addition to the set of bodies, where each body is typically represented by a mass, position, and velocity. At the beginning of each time step, Barnes-Hut creates a new quad/octtree based on the current position of the bodies. Then, for each body, the algorithm traverses the tree to calculate the force on that body, using the threshold *theta* to determine when to treat distant body clusters as a single body. The algorithm then updates the state (position, velocity, etc.) of the body based on this force. The algorithm then returns to the first loop, builds a new tree for the new body positions, and repeats.

```
def BarnesHut (TimeSteps, Bodies, theta):
    for # of TimeSteps:
        Tree = BuildTree (Bodies)
        for each body b_i in Bodies:
            force = CalculateForce (b_i, Tree, theta)
            b_i = UpdateState (force, b_i)
        Bodies = (b_0, b_i,. . ., b_N)
    return Bodies
```

Algorithm 1: Pseudocode of software Barnes-Hut implementation.

```
def BarnesHut_TC_SW (TimeSteps, Bodies):
    Tree = BuildTree (Bodies)
    for # of TimeSteps:
        Tree_Serial = SerializeTree (Tree)
        Bodies_Serial = SerializeOrderedBodies (Tree)
        PopulateCache (Bodies_Serial, Tree_Serial)
        ClearTree (Tree)
        StartFPGA ()
        while FPGABusy ():
            if FPGAHasData ():
                PartialBuildTree (Tree, ReadFPGAResults())
    return GetBodies (Tree)
```

Algorithm 2: Pseudocode of the software portion of the (similarity-based) traversal cache implementation of Barnes-Hut.

4.3. Software Extensions. The software for the extended framework has several responsibilities in addition to those required by the general framework. Pseudocode is shown in Algorithm 2. The basic structure of the code is similar to the Barnes-Hut code, with code that creates a new tree structure based on the body positions at each time step. This is done once initially for the initial positions of all bodies and is done during each time step in parallel with the accelerator's calculations by adding the updated bodies as they are made available by the accelerator.

After constructing the tree, the software serializes the entire tree in the order used by the Barnes-Hut traversal (i.e., preorder), while including address offset information necessary for the accelerator to execute skips that may occur due to grouping of distant bodies (i.e., *theta* threshold comparisons). These skips are processed by the FPGA, but because we cannot statically determine when the skips occur, the serialized tree representation must include all nodes of the tree and information at each node about how to skip over the node's subtree. After creating the serialized tree, the software transfers the serialized form to the traversal cache. This behavior is, in a sense, a generalization of the previous approach which provided the data for a single traversal at a time, in the order of just that traversal.

The software also provides the traversal cache with any data outside the data structure necessary to generate each traversal, which we refer to as *traversal inputs.* For Barnes-Hut, the traversal inputs are the individual bodies used by each datapath (the bodies subject to the net force being computed by that datapath). In general, the performance of the FPGA is maximized by the traversal cache when the traversal inputs are ordered to maximize the similarity between the traversals for adjacent traversal inputs. Our Barnes-Hut implementation determines this ordering (shown in the figure as *SerializedOrderedBodies*) by dividing the bodies into fixed-sized *clusters* of closely-located bodies, which are likely to traverse the tree in similar ways because of similar threshold comparisons at a distance. For Barnes-Hut, this clustering has little overhead because particles are already spatially grouped in the tree structure. Bodies within each cluster are then processed by the accelerator in parallel. Note that for some applications including Barnes-Hut, increasing the cluster size can still result in additional speedup even when further datapath replication is limited by area constraints, as discussed in the next section.

Once it has populated the traversal cache, software starts the accelerator, which processes the clusters one at a time, generating traversals and accumulating forces for the individual bodies. After completing each cluster, the accelerator adds results data to the results memory and asserts a *cluster_ready* signal, which is checked by software (shown in the figure as *FPGAHasData*). This allows software to construct the next version of the Barnes-Hut tree while the accelerator is still working. When all clusters have been processed by the accelerator, the *ready* signal is asserted and software increments the current time step, serializes the quadtree and clusters, and restarts the process.

4.4. FPGA Framework. The FPGA portion of the traversal cache framework is shown in Figure 4. The majority of the computation is handled by the datapaths, which are replicated p times up to the area of the target device, with each replication handling a separate traversal stream. Before work is begun on a cluster, the traversal inputs are read into registers (not shown) in the datapath logic, beginning from the address *clusters_addr*. The data structure is streamed out of the traversal cache, starting with the element at *ds_addr*, with elements appearing on the common data bus *tc_data*. The elements belonging to each individual traversal are identified from the common traversal cache stream by logic referred to as a *generator kernel*. The decisions made by these generator kernels are also used by a common controller called a *traversal generator* to steer further exploration of the data structure.

More precisely, the decision about whether a particular element from the cache is included in a given traversal (the traversal corresponding to a given traversal input) is determined by application-specific logic, referred to as a *generator kernel*, which is replicated once for each of the p parallel traversals. This logic is rarely the same between applications since the exploration of pointer data structures is almost always driven by computation unique to the application. For

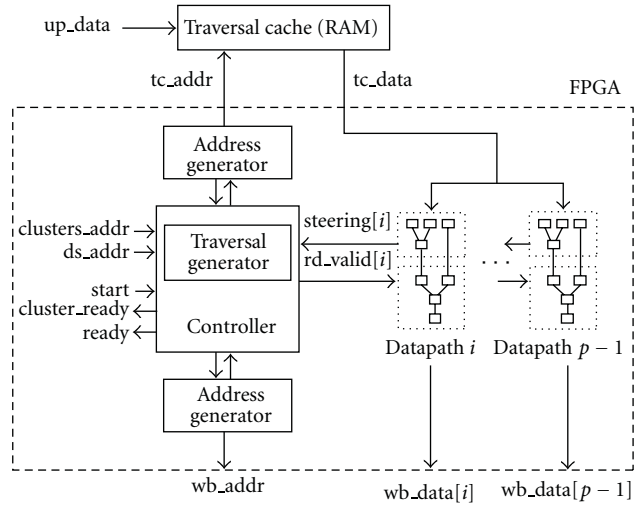


Figure 4: FPGA portion of the extended traversal cache framework with unrolled datapaths, each provided with separate traversals through selectively enabled *rd_valid[i]* signals. Element membership in each traversal is determined by application-specific logic, *generator kernels*, as part of each datapath (shown here as a dotted segment) and typically data structure-specific logic in the *traversal generator*, shown here as part of the controller.

example, although Barnes-Hut generally explores the tree in depth-first order, which elements are included in the depth-first traversal for a given body is determined by the *theta* threshold calculation for internal nodes. Thus, the Barnes-Hut generator kernel is this threshold calculation, which is dependent on the traversal's body (the corresponding *traversal input*) and the current element. In Barnes-Hut, as is often the case, the generator kernel logic produces many intermediate values needed for datapath computation and is itself rather computation intensive. This tight relationship between generator kernel and datapath is illustrated in Figure 4 by showing the generator kernel as an upstream section of each datapath.

Depending on the data structure being explored, a generator kernel may also decide *steering*, in addition to the element *membership* signal, describing how its traversal should proceed exploring the data structure. In Barnes-Hut, the only steering data is whether to skip over the current subtree, which is logically equivalent to the membership signal. However, membership and steering are collectively shown as a *steering* signal in Figure 4 for generality.

The *traversal generator* is application-specific control logic that explores the in-cache data structure (by controlling the read address generator) to stream all the elements required to process a cluster of traversals. The generator uses the steering data provided by the generator kernels to decide how to traverse the data structure minimally, skipping over any elements not required by any traversals in the cluster, while ensuring that even elements required by any single traversal are still read. In the case of Barnes-Hut (or any depth-first traversal) this condition is met by only skipping when all generator kernels agree to skip the current section of tree.

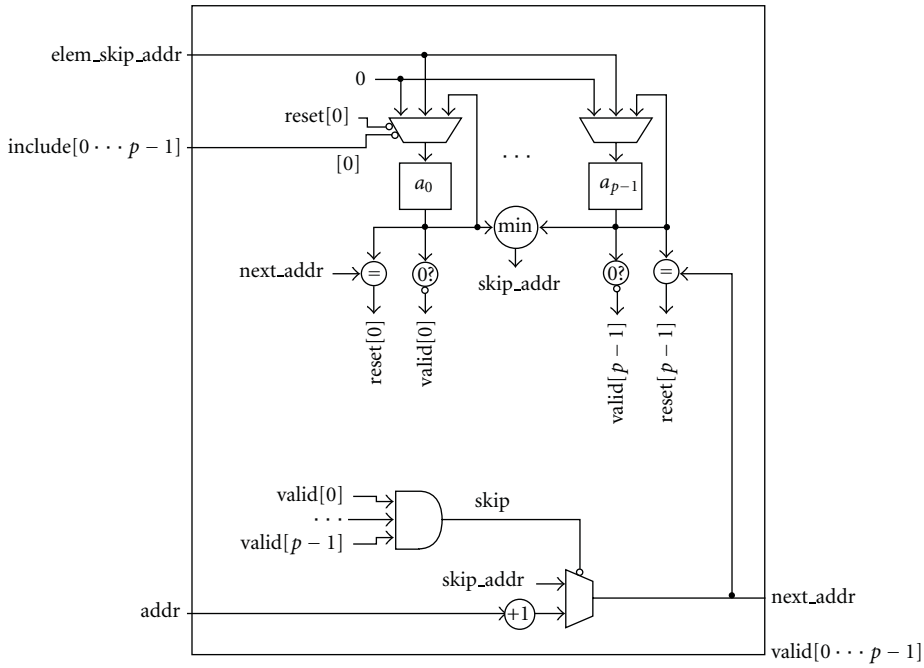


Figure 5: Simplified *traversal generator* logic for preorder tree traversal used to implement Barnes-Hut. The traversal generator guides exploration of the data structure for p simultaneous traversals, minimizing accesses by coordinating multiple datapaths by selectively asserting *valid[ ]* using membership data from each datapath's *generator kernel* on *include[ ]*.

Handling a skip is implemented by augmenting the in-cache representation of the data structure with additional data. For example, Barnes-Hut skips over subtrees when an internal node can be treated as a center of mass. Our implementation of Barnes-Hut implements these skips by including a child node count on each internal node in the octtree's serialized representation. Since the tree is serialized in preorder, skipping over the current subtree is handled by simply incrementing the current address by the current (head) node's child count multiplied by the size of a serialized node (a constant).

The traversal generator must also ensure that elements read for the benefit of only some traversals in the cluster are not delivered for traversals (generator kernels or datapaths) that have previously opted to steer around them. For example, in our Barnes-Hut implementation, if all but one datapath can skip a portion of the tree, the generator must still explore the section for the holdout while blocking that data from reaching the other datapaths.

Note that although the traversal generator is application-specific in general, a single generator can often be reused between multiple applications, since it only requires knowledge of the data structure and how to move through it (e.g., what types of skips are possible). This reuse is the primary reason for separating traversal generator logic from the generator kernels. Barnes-Hut, for example, performs a preorder traversal of an n-ary tree. However, besides the logic governing when to skip a subtree, which is implemented in the generator kernels, only the node size and n-ary size of the tree are specific to Barnes-Hut and can be made parameters of a generic preorder generator component.

The traversal generator implementation for Barnes-Hut is illustrated with simplifications in Figure 5. The generator takes as inputs *addr*, the address of the current element being streamed from the cache, *elem_skip_addr*, the address of the next element after the current subtree (part of the current element record as stored in the cache), and *include[ ]*, a vector of membership bits from the generator kernels shown in Figure 5. The generator outputs *next_addr*, the next address to read from the traversal cache, and *valid[ ]*, a vector of data valid bits for the datapaths used to separate the individual traversals.

Registers a_0 through a_{p-1} are used to track the progress of each traversal through the preorder serialized data structure. Each register starts at value 0, indicating that the corresponding traversal is not skipping the current node. When a generator kernel determines the current subtree should be skipped, the corresponding *include* signal is deasserted, storing *elem_skip_addr* (the address of the next potentially included element after the current subtree) in the register. Each register is *reset* to 0 on the cycle before its stored address (if any) is accessed in the cache. As long as the register is nonzero, the corresponding *valid* signal is deasserted, skipping over the current element for that traversal by preventing processing in the datapath. The skip signals for individual traversals are ANDed together to enable the generator to jump over sections of the data structure by conditionally setting *next_addr* to the minimum nonskipped element address stored in the registers. The tree of comparators and MUXs implementing this min operation is the primary source of overhead due to the traversal cache in our Barnes-Hut implementation, growing roughly

as $n \log n$ in area and $\log n$ in performance. In general, the overhead from a given traversal cache implementation is application dependent, depending heavily on the logic required by the traversal generator. However, since an efficient FPGA implementation is often not possible without caching for applications that would use the framework, including Barnes-Hut, this "overhead" is not really in conflict with accelerator speedup though it must still be minimized to optimize performance.

Sometimes generator kernels may be complex enough to require pipelining. In this case, the *include* signals in Figure 5 are delayed to the generator logic by as long as the generator kernel's latency. In order to avoid stalling excessively the generator must continue to fetch elements from the cache, predicting a skip will not occur. Since the generator kernels may continue to receive elements they will eventually steer around, they must either be stateless (i.e., no dependence on past data) or must be capable of reverting their state when it is determined that previously determined elements were invalid. This was not an issue for Barnes-Hut, since the *theta* threshold calculation that determines membership is a function of the current element only.

In the simplest case, the number of traversals handled by the framework in parallel is limited to p, the number of replicated datapath-generator kernel pairs, which is limited by the resources available on the FPGA. The number of parallel traversals can be increased without increasing datapath replication by creating clustered traversal inputs that can be divided into s-sized subclusters. For each subcluster, the traversal generator computes traversals in parallel, then sequentially swaps in a new subcluster and repeats until the entire cluster is processed for the current serialized tree element. The amount of datapath replication p and the number of subclusters s are architectural parameters of the framework that can be varied for different devices and inputs. The total number of traversals considered simultaneously is then given by $p \times s$, which is equivalent to the cluster size. This approach requires additional logic not shown in Figures 4 or 5 for clarity. The performance of this type of clustering is evaluated in Section 5 for our implementation of Barnes-Hut.

4.5. Extensions for Other Applications. The previous sections discuss how the traversal cache framework exploits similarity between traversals for an n-body simulation. The framework can potentially be extended to other applications using the following methodology.

First, the software must be able to serialize the data structure used by all traversals. Serialization is straightforward if the same basic traversal order is used for all traversals. For example, Barnes-Hut used a preorder traversal of the tree, while occasionally skipping nodes. Therefore, we serialized the tree using a preorder traversal. In addition, the traversal generators in the FPGA must be able to identify when incoming data is part of the traversal for a given input. For example, the traversals for Barnes-Hut differed for every input, but the traversal generator could detect when there was overlap by applying the threshold comparison used for skipping over nodes. If an application uses completely different traversal orderings, the framework could potentially still achieve speedup, but only if the traversal generator in the FPGA can detect when overlap occurs.

After serializing the tree, the software must also be able to determine an ordering of traversal inputs that provides for good similarity. For Barnes-Hut, this ordering was based on clustering close particles. If it is not clear what order would produce high similarity, or if determining that order would have a large overhead, then the framework may not benefit the corresponding application.

4.6. Limitations. The maximum speedup using the similarity-based traversal cache is achieved for algorithms with independent traversals in the same order, for example depth first, across a data structure. In the case that an algorithm's traversals are dependent on previous traversals, the framework cannot process traversals in parallel ($p = s = 1$). For algorithms whose traversals occur in different orders across a data structure resulting in no data reuse, the framework is equivalent to the general framework and requires software intervention between processing traversals.

The potential for speedup from the framework increases with the similarity between successive traversals. Because not all applications exhibit high similarity between traversals, the framework does not always improve performance. Low similarity between traversals results in disagreement among generator kernels about which regions of the data structure can be ignored. Since the generator must satisfy the needs of all the kernels to maintain correctness, it must include regions needed by any single kernel, stalling any other kernels (and datapaths) that do not need those elements and reducing parallelism. Since element accesses are grouped for traversals computed in parallel, it can also be shown that the total number of accesses is minimized when similarity is maximized. However, because efforts to maximize the similarity between traversals would also benefit pure software implementations due to caches on main memory, implementations using the framework might benefit from existing work along those lines.

In this paper, we assume the data structure fits in the traversal cache, which we currently store in a memory on the FPGA board. In situations where the size of memory prevents storing the entire data structure in the cache, software could load only part of the data structure, paging other parts into the cache as they are reached by the accelerator. However, the effect on performance would depend heavily on the algorithm being implemented due to the overhead of loading potentially many unused elements included in the data structure's cached representation. If the number of elements within each traversal is comparable to the size of the data structure itself (e.g., instances of Barnes-Hut with low *theta*), paging in and out the entire data structure might provide acceptable performance. Otherwise, application-specific approaches would probably be necessary for acceptable performance. One possibility is prefiltering the data structure to remove elements that it is known will not be required by the current window of traversals.

However, implementations not using a traversal cache would also be limited by the size of available memory, and possibly sooner since the ordering assumed by the framework allows the data structure's serialized form to often be smaller than the original form in memory. More detailed analysis of the effects of cache size on the performance of implementations using the framework is left as future work.

5. Experiments

5.1. General Framework

5.1.1. Experimental Setup. To evaluate the traversal caches, we implemented the framework on a system combining a 3.2 GHz Xeon with a Nallatech H101-PCIXM [24], which consists of a Xilinx Virtex 4 LX100 FPGA with 4 SRAM banks and 1 SDRAM. We mapped the framework onto this target architecture in the following way. All software uses the 3.2 GHz Xeon, and all custom circuits execute on the FPGA. All communication between the microprocessor and FPGA is sent over a PCI-X bus. Control and synchronization signals are read from and written to memory mapped registers inside of the LX100 using memory map nodes provided by Nallatech. For each example, the traversal cache is implemented in either an SRAM bank or on the SDRAM depending on the required size. To investigate effects of invalidation rate on performance, we do not base the invalidation rate on a specific input stream, and instead manually test different invalidation rates. This approach allows us to test different input possibilities ranging from the worst case to the best case. For each example, we test invalidation rates of 1 (invalidate every traversal), 5 (invalidate every 5 traversals), 10, 20, 40, and 80.

We evaluated the framework using the following benchmarks, which we developed. For each benchmark, we describe the pointer-based data structure and justify tested invalidation rates. For each software baseline, GCC's prefetch builtin was used as soon as the next node's address was available to begin loading the next node's data as early as possible.

Search scans a linked list of 1 million 16-bit integers and determines the number of occurrences of a specified value. The FPGA implementation performs 16, 16-bit comparisons and 15 additions every cycle. The invalidation rate for *search* is likely to be low for any application that searches a data structure multiple times without changing, such as a database application.

Audio performs convolution of an input signal consisting of 16-bit audio samples with a 64 sample impulse response. The data structure used by audio is a linked list of audio streams, which may likely occur in a digital audio workstation or a video game. Repeated traversals are likely since the actual audio stored in these applications does not change frequently. The circuit implementation performs 64 multiplications and 63 additions every cycle.

Graphics performs 3-dimensional vertex transformations by multiplying 4×4 transformation matrices with 4×1 vertex matrices. *Graphics* uses a list of vertex arrays, where each node of the list represents an object to be rendered. The implementation performs 16 multiplications and 28 additions every cycle. All operations are floating point.

Hardware/software communication and bookkeeping common to all the benchmarks was implemented in standalone C and VHDL libraries. These libraries required under a week to develop, test, and refine, using a number of components provided by Nallatech including a PCI-X interface, SRAM/SDRAM controllers, and memory map interfaces. The software implementation of each benchmark was completed in one day on average. The application-specific data structure serialization, invalidation bookkeeping, and VHDL datapaths collectively took on average between 1 and 2 additional days for each case study to develop and optimize.

We executed each example at the maximum possible clock frequency obtained after placement and routing using Xilinx ISE 9.2, which ranged from 115 MHz for *graphics* to 135 MHz for *search*. For all experiments, we compare traversal cache performance to software running on a 3.2 GHz Xeon. We compiled each benchmark using GCC 3.4.6 with –O3 optimizations to ensure that the baseline in the comparisons was as fast as possible.

5.1.2. Speedup Compared to Pointer-Based Software. This section presents performance advantages of the general traversal cache framework (not handling similarity) compared to software running on a 3.2 GHz Xeon. Figure 6(a) shows speedup in execution time obtained by the traversal cache compared with the original pointer-based software. The speedup is given for a number of invalidation rates (abbreviated IR). Comparisons are not shown for an FPGA implementation without the traversal cache as in most cases the cache less implementation did not provide a speedup over software.

Each benchmark was also rewritten to use an analogous approach to the traversal cache in software by maintaining traversed elements in an array "cache", enabling sequential access. The speedup of this approach is shown in Figure 6(a) as IR (sw). In the interest of space, we only show these results for the lowest and highest invalidation rates. Finally, the benchmarks were rewritten to use arrays natively, shown as array (sw) in the figure.

For the *search* example, only the highest invalidation rate (IR1) was slower than the pointer-based software. All other invalidation rates achieved large speedup compared to pointer-based software, reaching as high as 29x for IR80. Array-based software performance was better than the traversal cache framework for all invalidation rates under 40. For the IR80 case, the traversal cache was 1.3x faster than the software array implementation.

For *audio*, all invalidation rates, including IR1, were faster than both the pointer-based software implementation and the array implementation. Speedup ranged from 6.2x to 8.2x. The reason for the increased speedup at higher invalidation rates was because *audio* performed more computation for each piece of data, minimizing the effects of transferring the data to the FPGA.

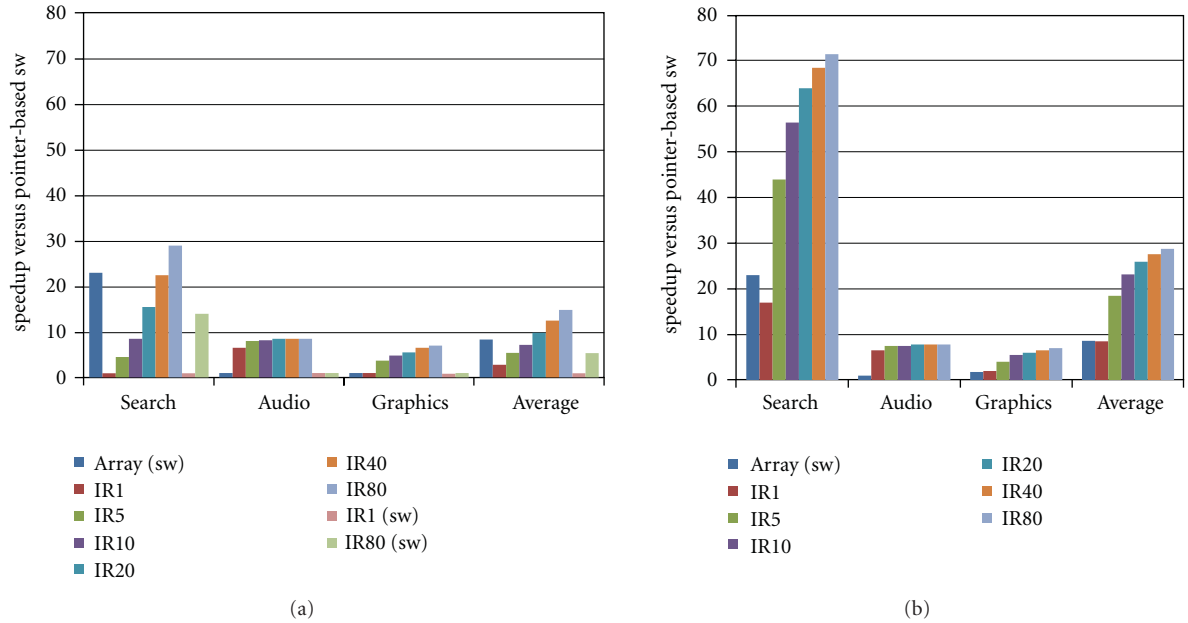


Figure 6: (a) Speedups versus the original pointer-based software implementation obtained for: array-based software (array) and an accelerator using the traversal cache framework for invalidation rates ranging from 1 to 80 (IR1-80). The results show that for some algorithms even high invalidation rates can achieve large speedup. (b) The same comparisons against pointer-based software after recoding the traversal cache implementations to use arrays. A similar speedup for *audio* and *graphics* shows that traversal caches can sometimes nearly completely hide the overhead of pointer-based structures.

Graphics achieved similar results, always outperforming the pointer and array software implementations with speedup ranging from 1.2x to 6.7x.

5.1.3. Speedup after Recoding. To determine how much improvement could be obtained by using arrays within the traversal cache framework instead of pointer-based data structures, in this section, we report the speedups assuming a designer were to recode the benchmarks to use arrays.

Figure 6(b) illustrates the speedup, again compared to pointer-based software, after recoding. For *search*, speedup more than doubled, reaching 70x for IR80. The increased speedup resulted from the significantly slower pointer-based software, which was more than 20x slower than the array-based software. We believe this performance difference is due to page faults caused by the large list size. The other examples achieved almost identical performances after being implemented with arrays. This surprising result implies that for certain examples, traversal caches make pointer-based code just as efficient on FPGAs as array-based code—a significant achievement considering the traditionally bad performance that has resulted from pointer-based structures.

5.2. Similarity Extensions: Barnes-Hut Case Study

5.2.1. Experimental Setup. We implemented Barnes-Hut for classical gravitational forces in two and three dimensions. The quad/octtree traversal logic was implemented as a generator supporting generic preorder traversal and search over an n-ary tree, as illustrated in Figure 5, with Barnes-Hut generator kernels skipping subtrees depending on a programmable *theta*. Similarity between traversals is increased by loosely ordering the bodies by spatial locality in the universe, which is available inexpensively as the order of the leaves in the fully constructed octtree. Our software implementations also use this ordering, which we found provides a speedup of up to 50% due to improved use of the processor's cache. Therefore, speedup estimates are slightly pessimistic compared to normal software execution. In the hardware implementation, we also construct the next time step's tree concurrently with running the accelerator by streaming outputs as they become available although our simple implementation of tree construction remains a factor limiting speedup in our full-application studies (Figures 7 and 8). Implementing parallel tree construction methods [25] would likely improve total application speedup.

To evaluate the similarity extensions for the traversal cache, we implemented fixed-point force calculation datapaths and preorder control logic, using generics for the number of traversals computed in parallel p and the number of p-sized subsets computed sequentially, s, in each pass. In the case that $s = 1$, the design is nearly identical to the block diagram of Figure 4. The framework was configured to use a single SDRAM input memory and an SRAM results buffer. Cycle counts were extracted from an HDL test bench that used a memory model of the SDRAM available on the Nallatech H101-PCIXM, accessed through Nallatech's SDRAM controller. Timing was measured and verified using a hardware implementation on the Nallatech H101-PCIXM FPGA accelerator card.

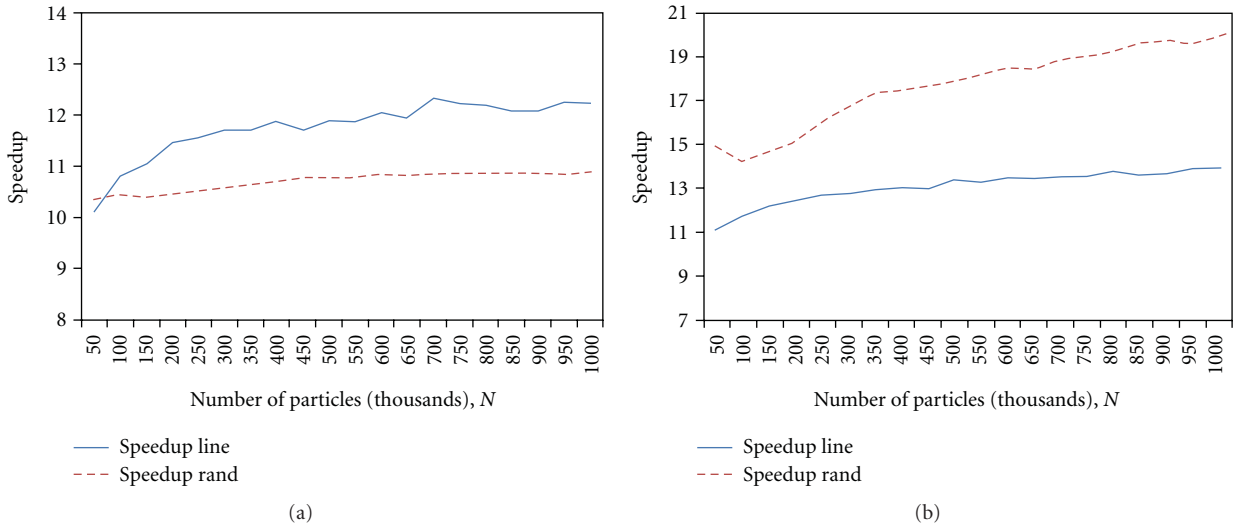


Figure 7: 2D (a) and 3D (b) Barnes-Hut application speedup achieved by the traversal cache framework on a Virtex 4 LX100 compared to a 3.2 GHz Xeon for various numbers of particles and representative distributions. The speedup is larger for the 3D algorithm, despite approximately equal similarity for each distribution, indicating that the increased computational intensity of the 3D algorithm takes better advantage of the FPGAs resources. *Theta* = 0.5.

The HDL simulation and timing data were combined to create a cycle-accurate C-based simulator of the hardware's behavior on arbitrary input data. Because synthesizing each test case would have been very time consuming, this C-based simulator enabled rapid exploration of the framework's design space, while also enabling consideration of configurations that would not fit on the Virtex 4 LX100 FPGA. The data provided for all configurations tested assumes the clock rate achieved for the largest design that would fit on the LX100 ($s = 1$, $p = 25$), which underestimates the performance of the smaller configurations tested.

Our software implementation of Barnes-Hut and the naïve n^2 algorithm took about a day to develop and test, with an additional day to implement optimizations. By comparison, the traversal cache implementation was developed and tested over about 2 weeks from start to finish, including all hardware and software components.

5.2.2. Performance Results. Our traversal cache Barnes-Hut implementation was compared to software running on a 3.2 GHz Xeon. The framework was tested with the maximum parallelism supported by the LX100 with no sequencing within a cluster ($s = 1$, $p = 25$). Sequencing is discussed in the next section. The time required to serialize the quad/octtree was included in the execution time for the hardware implementations and is reflected in the speedups reported. The serialization process required additional host memory equal to the size of the serialized data structure although this could obviously change significantly with other implementations (e.g., by generating and sending in chunks).

Because the traversal generator provides maximum bandwidth for high similarity traversals, and for Barnes-Hut the similarity between traversals is data dependent, we compared the system's performance on two data sets

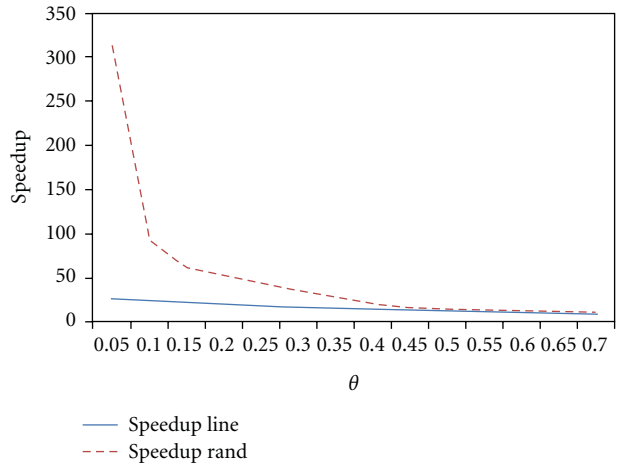


Figure 8: For lower values of theta (i.e., less approximation in Barnes-Hut), the traversal cache framework achieves larger application speedup. Higher values of *theta* than shown are not common in practice. $N = 100$ k.

representing extremes in similarity. The lowest similarity data set, *random*, has bodies distributed randomly in space, with on average 83% (for *theta* = 0.5) of elements in common between traversals for adjacent bodies. The highest similarity data set, *line*, has bodies evenly distributed in a line between opposite corners of space, with on average 90% of elements in common between subsequent traversals.

Figure 7 shows the application speedup of the framework relative to software for different problems sizes N and *theta* = 0.5, in two and three dimensions. Even though this value of *theta* is high for most practical n-body applications, it is used here to estimate worst-case performance, since higher values of *theta* result in shorter traversals. This is discussed in more

detail below in relation to the data in Figure 8, which shows speedup for different values of *theta*.

For the problem sizes tested, a speedup of between 10x and 12x was achieved for the 2D implementation, and between 11x and 21x for the 3D implementation, depending on the data set. The 3D simulations are more representative of practical n-body applications, which tend to be much more computation intensive than our gravitational force calculations and are usually physically three dimensional as well (e.g., molecular dynamics). The 2D simulations are still useful to illustrate the effectiveness of caching to enable speedup even for relatively low computation/memory ratios.

In both cases, speedup increased with problem size due to longer individual traversals resulting in more computational work being done before loading the next cluster, which better utilizes the FPGA's deep datapaths. The speedups for the 2D case were generally lower for the same reason, with less work being performed in the datapaths for each memory access. Execution times were universally lower for *line* due to better processor cache performance in software and improved parallelism in hardware due to better data reuse keeping more of the datapaths active.

Since real-world n-body problems tend to be more computationally complex than classical gravitational calculations, these results suggest even better speedups for real n-body applications. Although more involved calculations can entail larger pipelines and more limited unrolling due to area constraints, other techniques discussed in Section 3 can mitigate this effect.

However, the time required to construct the next tree structure confounds making some observations about the behavior of the traversal cache itself. The reduced effectiveness of increasing N is actually primarily due to the time required to construct the next tree, which quickly approaches the time spent doing actual computation. In the 3D case, application speedup is lower for the *line* data set despite a greater kernel speedup, because the lower total runtime in both hardware and software means tree construction figures more prominently in the total runtime. The cache's behavior is analyzed in more detail in Section 3 by focusing on kernel execution time.

The effect of the algorithm's precision parameter *theta* is shown in Figure 8 for a 3D simulation with 100 k bodies. The framework achieves higher application speedup for smaller values of *theta*, where the algorithm's complexity approaches $O(N^2)$. Traversals include more elements for these instances of the problem, resulting in a greater reduction in total memory accesses (improved bandwidth) due to the traversal cache. Including more elements per traversal also eventually results in fewer pipeline stalls due to fewer skips, contributing to a greater speedup. Note that comparing speedups of different *theta* values is different than comparing execution times. As *theta* increases, the complexity and execution time of the algorithm decreases, which contributes to the decreasing speedup due to the time required for the next tree construction. The value of *theta* chosen in practice depends on the particular application doing n-body, as it represents a tradeoff of simulation accuracy for execution time, but *theta*s less than 0.5 are common in real-world applications.

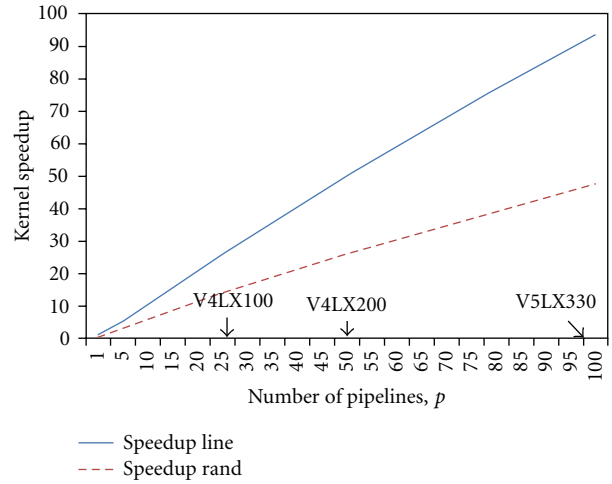


Figure 9: N-body kernel speedup obtained by the traversal cache framework for different amounts of datapath replication p. Note *that for high similarity, speedup increases almost linearly due to the framework exploiting similarity between traversals to eliminate the memory bandwidth bottleneck.* $N = 100$ k, *theta* = 0.5.

Previous FPGA implementations have focused primarily on n^2 implementations of n-body (*theta* = 0), which can usually only provide speedups for extremely precise simulations.

5.2.3. Effect of Framework Parameters. The traversal cache framework can be configured for an application through the parameters p and s, as discussed in Section 4.4. In this section, we explore the effect of these parameters for our traversal cache implementation of Barnes-Hut. Simulation data is provided in Figures 9 and 10 for a 3D system of 100 k bodies with *theta* = 0.5. In order to focus on the behavior of the framework itself, the data in these figures deals only with execution times and speedups for the force calculation and traversal kernels, referred to as the *kernel speedup*, which corresponds to the portion of Barnes-Hut implemented on the accelerator. The results for the Barnes-Hut application as a whole are discussed at the end of this section.

Figure 9 demonstrates that the kernel speedup provided by the framework increases with p by an amount determined by the similarity between traversals and grows nearly linearly for the high-similarity *line* case. The traversal cache is able to achieve this speedup *without increased demands on physical memory bandwidth by increasing effective memory bandwidth*, requiring only a single access for any single data structure element within a batch of traversals. Since p is limited by the size of the device, labels were added showing the amount of unrolling achievable on some common FPGAs.

Figure 10 shows the effect of s, the number of p-sized groups of traversals computed in sequence within a cluster, for Barnes-Hut force calculation implemented with 5 datapaths ($p = 5$). The simulation demonstrates that additional kernel speedup can be obtained after maximum unrolling by increasing s, up to a maximum, with reducing speedups after the maximum. This maximum speedup is also shown to be limited by the similarity between traversals,

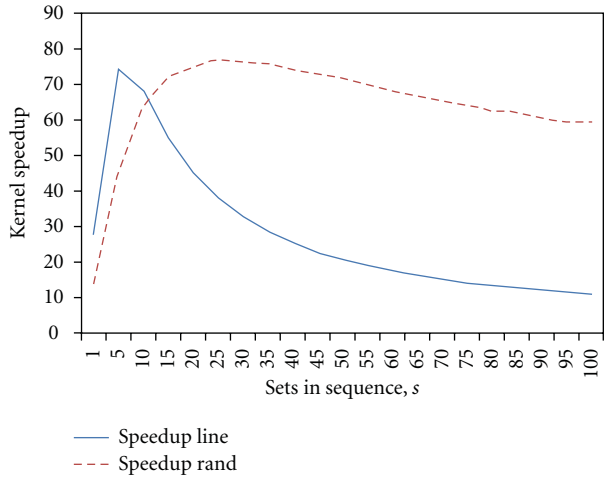


Figure 10: N-body kernel speedup obtained by the traversal cache framework for different amounts of datapath replication s. N = 100 k, *theta* = 0.5.

with higher similarities allowing a greater speedup for lower values of s. However, since the optimum amount of sequential processing s depends on the amount of similarity, which, as is the case for Barnes-Hut, is likely not constant or known *a priori*, implementations using sequencing would likely need to be adaptive, adjusting s according to an observed or predicted amount of similarity.

The speedup that can be achieved for a full application is limited by the execution time for the software parts of the application. For our implementation, application speedup was limited to 11x (*line*) to 21x (*random*) for N = 1 M and *theta* = 0.5. Using variable cluster sizes by optimizing s for the data set, our implementation could achieve a maximum application speedup of 14x (*line*) to 70x (*random*) for N = 100 k and *theta* = 0.5, being limited by tree construction. Methods for adaptively varying s are left as future work.

6. Conclusions

In this paper, we introduced a traversal cache framework that enables efficient FPGA execution of applications with irregular memory access patterns. The general framework dynamically serializes pointer-based data structure traversals and stores them in a memory local to the FPGA as a cache, enabling reuse of repeated traversals. Such serialization greatly improves effective memory bandwidth for repeated traversals. We also presented extensions that exploit similarity between nonidentical traversals that enable multiple traversals to be processed in parallel. For a Barnes-Hut n-body case study, the framework was shown to achieve speedups ranging from 11x to 21x compared to software on a 3.2 GHz Xeon processor using a Virtex4 LX100, with higher speedups of over 70x possible through runtime-adaptive techniques. More importantly, by exploiting similarity between traversals, the framework can nearly eliminate the common memory bandwidth bottleneck, leading to nearly linear increases in speedup with additional area.

Future work includes automating designer specified portions of the frameworks for integration with high-level synthesis and hardware/software partitioning tools. For the generalized traversal cache framework, additional work on cache eviction placement strategies should also be explored as a means to reduce invalidation rates and provide even larger speedups.

References

[1] A. DeHon, "Density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[2] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *Proceedings of the ACM/SIGDA 12th ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, pp. 162–170, 2004.

[3] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Computational density of fixed and reconfigurable multi-core devices for application acceleration," in *Proceedigns of Reconfigurable Systems Summer Institute 2008 (RSSI '08)*, 2008.

[4] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George, "Rat: a methodology for predicting performance in application design migration to fpgas," in *Proceedings of the 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '07)*, pp. 1–10, ACM, New York, NY, USA, 2007.

[5] P. Diniz and J. Park, "Data search and reorganization using FPGAs: application to spatial pointer-based data structures," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 207–217, 2003.

[6] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*, pp. 249–256, 2004.

[7] P. Grun, N. Dutt, and A. Nicolau, "Access pattern based local memory customization for low power embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 778–784, IEEE Press, Piscataway, NJ, USA, 2001.

[8] A. Y. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the Barnes-Hut method for n-body simulations," in *Proceedings of the ACM/IEEE conference on Supercomputing*, pp. 439–448, ACM, New York, NY, USA, 1994.

[9] L. Semeria and G. De Micheli, "SpC: synthesis of pointers in C. Application of pointer analysis to the behavioral synthesis from C," in *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 340–346, 1998.

[10] L. Semeria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 743–756, 2001.

[11] L. Zhang, Z. Fang, M. Parker et al., "The impulse memory controller," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1117–1132, 2001.

[12] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proceedings of the Proceedings of the 35th Annual ACM/IEEE International Symposium on*

Microarchitecture, pp. 62–73, Computer Society Press, Los Alamitos, Calif, USA, 2002.

[13] N. Weinberg and D. Nagle, "Dynamic elimination of pointer expressions," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, p. 142, IEEE Computer Society, Washington, DC, USA, 1998.

[14] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behaviour," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 209–220, May 2002.

[15] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[16] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pp. 139–149, 1998.

[17] P. Grun, N. Dutt, and A. Nicolau, "Memory aware compilation through accurate timing extraction," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 316–321, ACM, New York, NY, USA, 2000.

[18] P. R. Panda, F. Catthoor, N. D. Dutt et al., "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[19] N. Baradaran and P. C. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 4, article no. 61, 2008.

[20] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, pp. 165–176, 2002.

[21] G. Stitt, G. Chaudhari, and J. Coole, "Traversal Caches: a first step towards FPGA acceleration of pointer-based data structures," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS '08)*, pp. 61–66, ACM, New York, NY, USA, October 2008.

[22] J. Coole, J. Wernsing, and G. Stitt, "A traversal cache framework for FPGA acceleration of pointer data structures: a case study on Barnes-Hut N-body simulation," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '09)*, pp. 143–148, December 2009.

[23] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 145–154, 1999.

[24] Nallatech Inc. Nallatech PCIXM FPGA accelerator card, 2010, http://www.nallatech.com/~nallatech/index.php/PCI-Express-Cards/h101-pcixm.html.

[25] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes," *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 953693, 15 pages
doi:10.1155/2010/953693

Research Article

Low-Complexity Online Synthesis for AMIDAR Processors

Stefan Döbrich and Christian Hochberger

Chair for Embedded Systems, University of Technology, Nöthnitzer Straße 46, 01187 Dresden, Germany

Correspondence should be addressed to Stefan Döbrich, stefan.doebrich@tu-dresden.de

Received 4 March 2010; Revised 27 September 2010; Accepted 17 December 2010

Academic Editor: Lionel Torres

Future chip technologies will change the way we deal with hardware design. First of all, logic resources will be available in vast amount. Furthermore, engineering specialized designs for particular applications will no longer be the general approach as the nonrecurring expenses will grow tremendously. Reconfigurable logic has often been promoted as a solution to these problems. Today, it can be found in two varieties: field programmable gate arrays or coarse-grained reconfigurable arrays. Using this type of technology typically requires a lot of expert knowledge, which is not sufficiently available. Thus, we believe that online synthesis that takes place during the execution of an application is one way to broaden the applicability of reconfigurable architectures. In this paper, we show that even a relative simplistic synthesis approach with low computational complexity can have a strong impact on the performance of compute intensive applications.

1. Introduction

Following the road of Moore's law, the number of transistors on a chip doubles every 24 months. After being valid for more than 40 years, the end of Moore's law has been forecast many times now. Yet, technological advances have kept the progress intact.

Further shrinking of the feature size of traditionally manufactured chips will give us two challenges. Firstly, exponentially increased mask costs will make it prohibitively expensive to produce small quantities of chips for a particular design. Also, the question comes up: how to make use of the vast amounts of logic resources without building individual chip designs for each application?

MPSoCs currently receive a lot of attention in this area. They can be software programmed, and by the use of large amounts of processing cores, they can make (relatively) efficient use of large quantities of transistors. Yet, their big drawback is that all the software has to be restructured and reprogrammed in order to use many cores. Currently, this still seems to be a major problem.

Reconfigurable logic in different granularities has been proposed to solve both problems [1]. It allows us to build large quantities of chips and yet use them individually. Field programmable gate arrays (FPGAs) are in use for this purpose for more than two decades. Yet, it requires much expert knowledge to implement applications or part of them on an FPGA. Also, reconfiguring FPGAs takes a lot of time due to the large amount of configuration information.

Coarse-Grained Reconfigurable Arrays (CGRAs) try to solve this last problem by working on word level instead of bit level. The amount of configuration information is dramatically reduced, and also the programming of such architectures can be considered more *software style*. The problem with CGRAs is typically the tool situation. Currently available tools require an adaptation of the source code and typically have very high runtime so that they need to be run by experts and only for very few selected applications.

Our approach tries to make the reconfigurable resources available for all applications in the embedded systems domain, particularly mobile devices or networked devices where the requirements change. Thus, synthesis of accelerating circuits takes place during the applications execution as we do not have access to the applications beforehand. No hand crafted adaptation of the source code will be required, although it is clear that manual fine-tuning of the code can lead to better results. The intention of the AMIDAR concept is to provide a general purpose processor

that can automatically take advantage of reconfigurable resources.

In this contribution, we want to show that even a relatively simple approach to online circuit synthesis can achieve substantial application acceleration.

The remainder of this paper is organized as follows. In Section 2, we will give an overview of related work. In Section 3, we will present the model of our processor which allows an easy integration of synthesized functional units at runtime. In Section 4, we will detail how we figure out the performance sensitive parts of the application by means of profiling. Section 5 explains our online synthesis approach. Results for some benchmark applications are presented in Section 6. Finally, we give a short conclusion and an outlook onto future work.

2. Related Work

Fine-grained reconfigurable logic for application improvement has been used for more than two decades. Early examples are the CEPRA-1X which was developed to speed up cellular automata simulations. It gained a speedup of more than 1000 compared with state of the art workstations [2]. This level of speedup still persists for many application areas, for example, the BLAST algorithm [3]. Unfortunately, these speedups require highly specialized HW architectures and domain specific modelling languages.

Combining FPGAs with processor cores seems to be a natural idea. Compute intense parts can be realized in the FPGA and the control intense parts can be implemented in the CPU. GARP was one of the first approaches following this scheme [4]. It was accompanied by the synthesizing C compiler NIMBLE [5] that automatically partitions and maps the application.

Static transformation from high level languages like C into fine-grained reconfigurable logic is still the research focus of a number of academic and commercial research groups. Only very few of them support the full programming language [6].

Also, Java as a base language for mapping has been investigated in the past. Customized accelerators to speed up the execution of Java bytecode have been developed [7]. In this case, only a small part of the bytecode execution is implemented in hardware and the main execution is done on a conventional processor. Thus, the effect was very limited.

CGRAs have also been used to speed up applications. They typically depend on compile time analysis and generate a single-datapath configuration for an application beforehand: RaPiD [8], PipeRench [9], Kress-Arrays [10], or the PACT-XPP [11]. In most cases, specialized tool sets and special-purpose design languages had to be employed to gain substantial speedups. Whenever general-purpose languages could be used to program these architectures, the programmer had to restrict himself to a subset of the language and the speedup was very limited.

Efficient static transformation from high level languages into CGRAs is also investigated by several groups. The DRESC [12] tool chain targeting the ADRES [13, 14] architecture is one of the most advanced tools. Yet, it requires hand-written annotations to the source code and in some cases even some hand crafted rewriting of the source code. Also, the compilation times easily get into the range of days.

Several processor concepts provide special features to simplify the adaptation of the processor to the needs of particular applications. The academic approaches like XiRISC/PicoGA [15] or SpartanMC [16] register in this category as well as commercial products like the Stretch processor [17]. Typically, these processors are accompanied by a toolset that statically determines performance critical parts of the application and maps them to the configurable part of the device.

The RISPP architecture [18] lies between static and dynamic approaches. Here, a set of candidate instructions are evaluated at compile time. These candidates are implemented dynamically at runtime by varying sets of so called atoms. Thus, alternative design points are chosen depending on the actual execution characteristics.

Dynamic transformation from software to hardware has been investigated already by other researchers. Warp processors dynamically transform assembly instruction sequences into fine-grained reconfigurable logic [19]. This happens by synthesis of bitstreams for the targeted WARP-FPGA platform. Furthermore, dynamic synthesis of Java bytecode has been evaluated [20]. Nonetheless, this approach is only capable of synthesizing combinational hardware.

Dynamic binary translation can also be seen as a runtime mapping of applications to spatial hardware. This has been explored in the past, for example, in the Transmeta Crusoe processor [21], where x86 instructions were dynamically translated into VLIW code. The main purpose of this approach was to execute legacy x86 code with much better energy efficiency.

Newer approaches start from binary RISC code like MIPS [22] or SimpleScalar [23]. In both cases, sequences of machine instructions are mapped onto an array of ALU elements in order to parallelize the computations. Nonetheless, it must be mentioned that both approaches up to now offer only limited speed ups. Yet, the approach taken in [22] enables the usage of partially defect arrays which makes it an interesting approach for future unreliable devices.

The token distribution principle of AMIDAR processors has some similarities with Transport Triggered Architectures [24]. These processors are sometimes referred to as *move processors*, because they are programmed by specifying the transport (move) of data between different components of the processor. This approach allows a relatively simple introduction of new processor components for specialized applications. TTAs, are customized statically at design time. In AMIDAR processors, the tokens also control the movement of data between components. Yet, in TTAs an application is transformed directly into a set of tokens. Storing the full token set of an application leads to a very high memory overhead and makes an analysis of the executed code extremely difficult. In contrast to this approach, AMIDAR processors compute the token set dynamically.

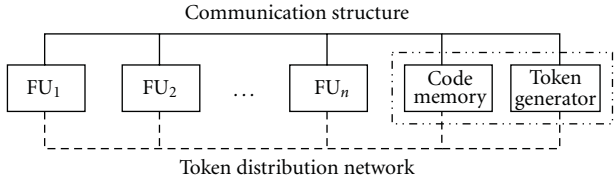


Figure 1: Abstract Model of an AMIDAR processor.

3. The AMIDAR Processing Model

In this section, we will give an overview of the AMIDAR processor model. We describe the basic principles of operation. This includes the architecture of an AMIDAR processor in general, as well as specifics of its components. Furthermore, we discuss the applicability of the AMIDAR model to different instruction sets. Afterwards, an overview of an example implementation of an AMIDAR-based Java machine is given. Finally, we discuss several mechanisms of the model, that allow the processor to adapt to the requirements of a given application at runtime.

3.1. Overview. An AMIDAR processor consists of three main parts. A set of functional units, a token network, and a communication structure.

Two functional units, which are common to all AMIDAR implementations, are the code memory and the token generator. As its name tells, the code memory holds the applications code. The token generator controls the other components of the processor by means of tokens. Therefore, it translates each instruction into a set of tokens, which are distributed to the functional units over the token distribution network. The tokens tell the functional units what to do with input data and where to send the results. Specific AMIDAR implementations may allow the combination of the code memory and the token generator as a single functional unit. This would allow the utilization of several additional side effects, such as instruction folding. Functional units can have a very wide range of meanings: ALUs, register files, data memory, specialized address calculation units, and so forth. Data is passed between the functional units over the communication structure. This data can have various meanings: program information (instructions), address information, or application data. Figure 1 sketches the abstract structure of an AMIDAR processor.

3.2. Principle of Operation. Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor explicit pipelining is used to execute instructions. Instead, instructions are broken down to a set of tokens which are distributed to a set of functional units. These tokens are 5 tuples, where a token is defined as $T = \{\text{UID}, \text{OP}, \text{TAG}, \text{DP}, \text{INC}\}$. It carries the information about the type of operation (OP) that will be executed by the functional unit with the specified id (UID). Furthermore, the version information of the input data (TAG) that will be processed and the destination port of the result (DP) are part of the token. Finally, every token contains a tag increment flag (INC). By default, the result of an operation is tagged equally to the input data. In case the TAG-flag is set, the output tag is increased by one.

The token generator can be built such that every functional unit which will receive a token is able to receive it in one clock cycle. A functional unit begins the execution of a specific token as soon as the data ports receive the data with the corresponding tag. Tokens which do not require input data can be executed immediately. Once the appropriately tagged data is available, the operation starts. Upon completion of an operation, the result is sent to the destination port that was denoted in the token. An instruction is completed, when all the corresponding tokens are executed. To keep the processor executing instructions, one of the tokens must be responsible for sending a new instruction to the token generator.

A more detailed explanation of the model, its application to Java bytecode execution, and its specific features can be found in [25, 26].

3.3. Applicability. In general, the presented model can be applied to any kind of instruction set. Therefore, a composition of microinstructions has to be defined for each instruction. Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimal. The model does not produce good results, if there is a strict order of those microinstructions, since in this case no parallel execution of microinstructions can occur. The great advantage of this model is that the execution of an instruction depends on the token sequence, not on the timing of the functional units. Thus, functional units can be replaced at runtime with other versions of different characterizations. The same holds for the communication structure, which can be adapted to the requirements of the running application. Thus, this model allows us to optimize global goals like performance or energy consumption. Intermediate virtual assembly languages like Java bytecode, LLVM bitcode, or the .NET common intermediate language are good candidates for instruction sets. The range of functional unit implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and/or basic operations are sufficiently complex. Finally, the data-driven approach makes it possible to easily integrate new functional units and create new instructions to use these functional units.

3.4. Implementation of an AMIDAR-Based Java Processor. The structure of a sample implementation of an AMIDAR-based Java host-processor is displayed in Figure 2. This section will give a brief description of the processors structure and the functionality of its contained functional units. The central units of the processor are the code memory and the token generator. The code memory holds the applications code. In case of a Java processor, this includes all class files and interfaces, as well as their corresponding constant pools and attributes. The Java runtime model separates local variables and the operand stack from each other. Thus,

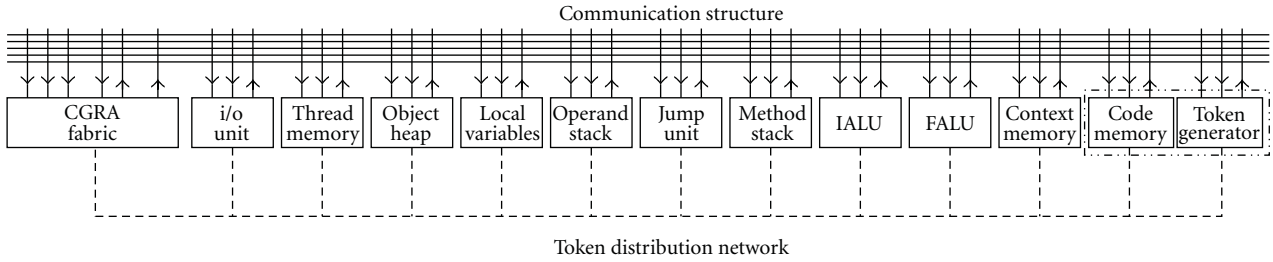


Figure 2: Model of a Java (non-)virtual machine on AMIDAR basis.

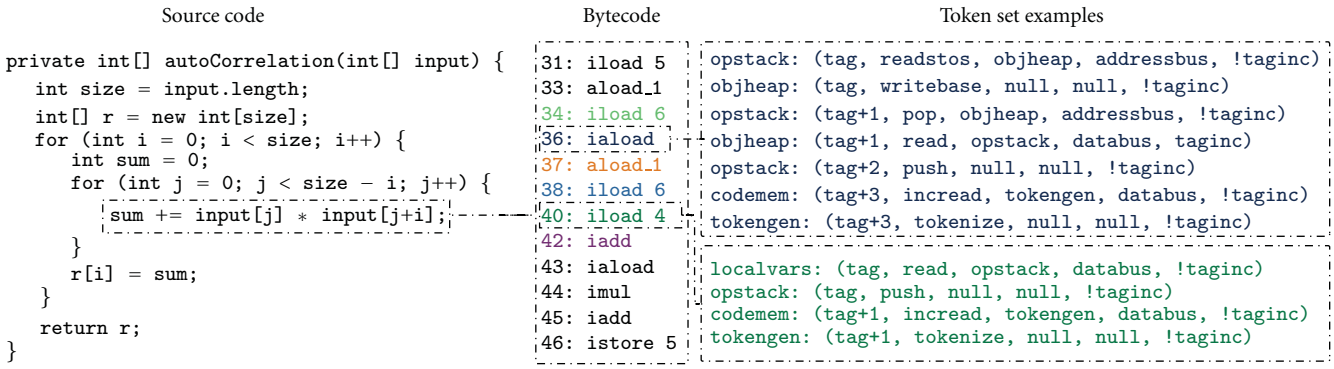


Figure 3: Example source code sequence and the resulting bytecode and exemplified token sequences.

a functional unit that provides the functionality of a stack memory represents the operand stack. Furthermore, an additional functional unit holds all local variables.

A local variable may be of three different types. It may be an array reference type or an object reference type, and furthermore, it may represent a native data type such as `int` or `float`. All native data types are stored directly in the local variable memory, while all reference types point to an object or array located on the heap memory. Thus, the processor contains another memory unit incorporating the so-called object heap. Additionally, the processor contains a method stack. This memory is used to store information about the current program counter and stack frame in case of a method invocation.

In order to process arithmetic operations, the processor will contain at least one ALU functional unit. Nonetheless, it is possible to separate integer and floating point operations into two disjoint functional units, which improves the throughput. Furthermore, the processor contains a jump unit which processes all conditional jumps. Therefore, the condition is evaluated and the resulting jump offset is transferred to the code memory. Additionally, the context memory contains the states of all sleeping threads in the system.

3.5. Example Token Sequence and Execution Trace. In order to give a more detailed picture of an actual applications execution on an AMIDAR processor, we have chosen an autocorrelation function as an example. The source code of the autocorrelation function, its resulting bytecode, and sample token sequences for two of its bytecodes are displayed in Figure 3. The `iaload` instruction at program counter `36` is focussed in the further descriptions.

The `iaload` bytecode loads an integer value from an array at the heap and pushes it onto the operand stack. Initially, the arrays address on the heap and the offset of the actual value are positioned at the top of the stack. Firstly, the arrays address is read from the second position of the stack and is sent to the heap where it is written to the base address register. Afterwards, the actual offset is popped of the stack, sent to the heap, and used as address for a read operation. The read value is sent back to the operand stack and pushed on top of the stack.

Figure 4 shows an excerpt of the execution of the autocorrelation function. Each line of the diagram represents the internal state of the displayed functional units in the corresponding clock cycle. Furthermore, all operations that belong to the same instruction are colored identically, which visualizes the overlapping execution of instructions.

3.6. Adaptivity in the AMIDAR Model. The AMIDAR model exposes different types of adaptivity. All adaptive operations covered by the model are intended to dynamically respond to the running applications behavior. Therefore, we identified adaptive operations that adapt the communication structure to the actual interaction scheme between functional units. Furthermore, a functional unit may be the bottleneck of the processor. Hence, we included similar adaptive operations for functional units. The following subsections will give an overview of the adaptive operations provided by the

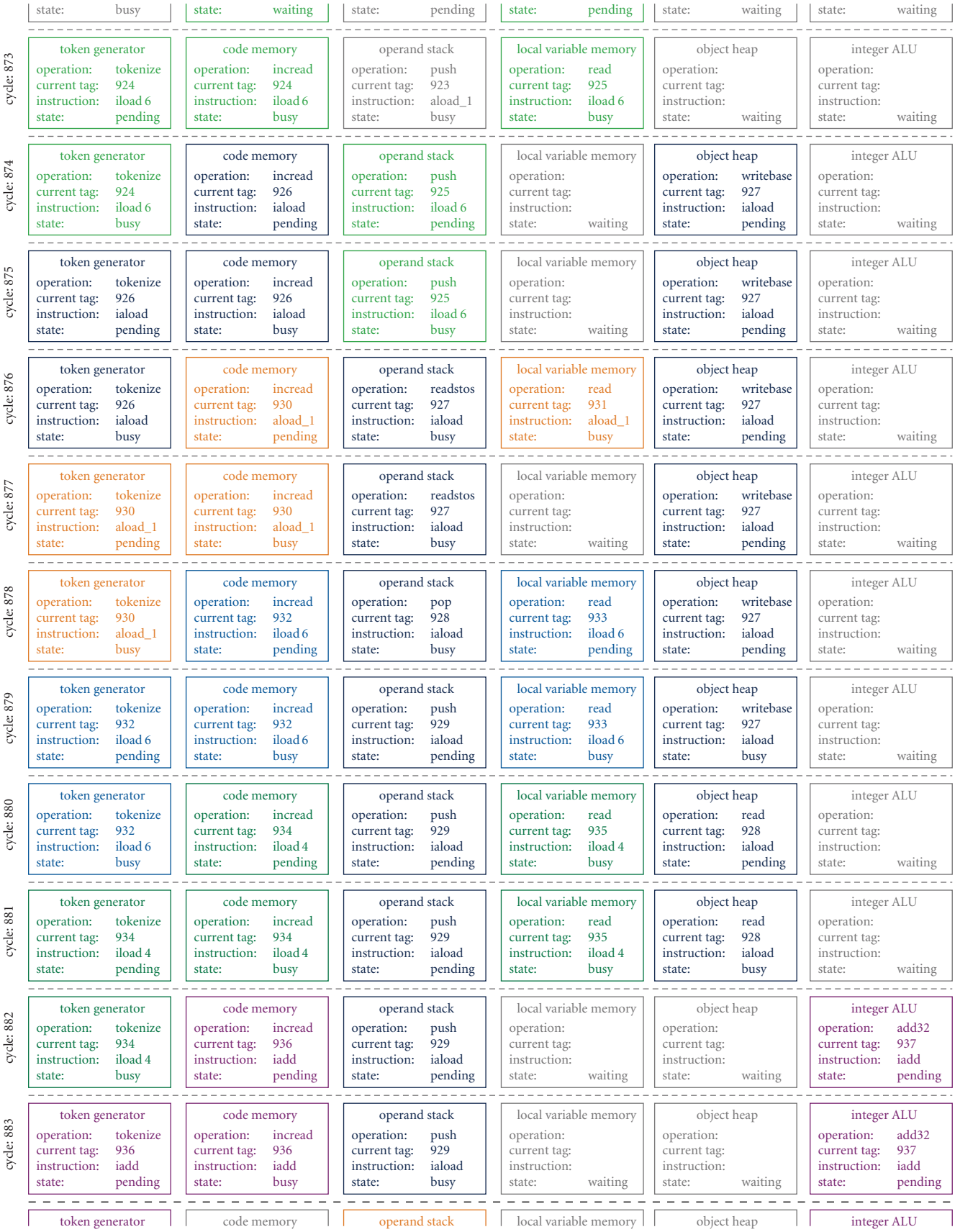


Figure 4: Visualized excerpt of an execution trace of the autocorrelation example.

AMIDAR model. Most of the currently available reconfigurable devices do not fully support the described adaptive operations (e.g., addition or removal of bus structures). Yet, the model itself contains these possibilities, and so may benefit from future hardware designs.

3.7. Adaptive Communication Structures. The bus conflicts that occur during the data transports between functional units can be minimized by adapting the communication structure. Therefore, we designed a set of several adaptive operations that may be applied to it.

In order to exchange data between two functional units, both units have to be connected to the same bus structure. Thus, it is possible to connect a functional unit to a bus in case it will send data to/receive data from another functional unit. This may happen if the two functional units do not have a connection yet. Furthermore, the two units may have an interconnection, but the bus arbiter assigned the related bus structure to another sending functional unit. In this case, a new interconnection could be created as well.

As functional units may be connected to a bus structure, they may also be disconnected. For example, this may happen in case many arbitration collisions occur on a specific bus. As a result, one connection may be transferred to another bus structure by disconnecting the participants from one bus and connecting them to a bus structure with sparse capacity.

In case the whole communication structure is heavily utilized and many arbitration collisions occur, it is possible to split a bus structure. Therefore, a new bus structure is added to the processor. One of the connections participating in many collisions is migrated to the new bus. This reduces collisions and improves the applications runtime and the processors throughput. Vice versa, it is possible to fold two bus structures in case they are used rarely. As a special case, a bus may be removed completely from the processor. This operation has a lower complexity than the folding operation, and thus may be used in special cases.

In [27], we have shown how to identify the conflicting bus taps and we have also shown a heuristics to modify the bus structure to minimize the conflicts.

3.8. Adaptive Functional Units. In addition to the adaptive operations regarding the communication structure, there are three different categories of adaptive operations that may be applied to functional units.

Firstly, variations of a specific functional unit may be available. This means, for example, that optimized versions regarding chip size, latency, and throughput are available for a functional unit. The most appropriate implementation is chosen dynamically at runtime and may change throughout the lifetime of the application. The AMIDAR model allows the processor to adapt to the actual workload by substitution of two versions of a functional unit at runtime. In [26], we have shown that the characteristics of the functional units can be changed to optimally suit the needs of the running application.

Secondly, the number of instances of a specific functional unit may be increased or decreased dynamically. In case a functional unit is heavily utilized, but cannot be replaced by a specialized version with a higher throughput or shorter latency, it may be duplicated. The distribution of tokens has to be adapted to this new situation, as the token generator has to balance the workload between identical functional units.

Finally, dynamically synthesized functional units may be added to the processors datapath. It is possible to identify heavily utilized instruction sequences of an application at runtime. A large share of applications for embedded systems rely on runtime intensive computation kernels. These kernels are typically wrapped by loop structures and iterate over a given array or stream of input data. Both cases are mostly identical, as every stream can be wrapped by a buffer, which leads back to the handling of arrays by the computation itself. In [28], we have shown a hardware circuit that is capable of profiling an applications loop structures at runtime. The profiles gained by this circuit can be used to identify candidate sequences for online synthesis of functional units. These functional units would replace the software execution of the related code.

It should be mentioned that in this work only dynamically synthesizing funcational units is taken into account as adaptive operation.

3.9. Synthesizing Functional Units in AMIDAR. AMIDAR processors need to include some reconfigurable fabric in order to allow the dynamic synthesis and inclusion of functional units. Since fine-grained logic (like FPGAs) requires large amount of configuration data to be computed and also since the fine-grained structure is neither required nor helpful for the implementation of most code sequences, we focus on CGRAs for the inclusion into AMIDAR processors.

The model includes many features to support the integration of newly synthesized functional units into the running application. It allows bulk data transfers from and to data memories, it allows the token generator to synchronize with functional unit operations that take multiple clock cycles, and finally it allows synthesized functional units to inject tokens in order to influence the data transport required for the computation of a code sequence.

3.10. Latency of Runtime Adaptivity. Currently, we cannot fully determine the latencies regarding the runtime behavior of the adaptive features of the AMIDAR model. The feature which is currently examined in our studies is the runtime synthesis of new functional units. Right now, the synthesis process itself is not executed as a separate Java thread within our processor, but only as part of the running simulator. Thus, the process of creating new functional units is transparent to the processor. Hence, a runtime prediction is not possible yet. It should be mentioned that the code currently used for synthesis could be run on the target processor as it is written in Java.

Nonetheless, the usefulness of synthesizing new functional units can be determined in two ways. In one case, there is no spare time concurrently to the executed task. Then, the runtime of the synthesis process for new functional units slows down the current operation, but after finishing

the synthesis, the functional units execute much faster. Thus, eventually, the runtime lost to the synthesis process will be gained back. In the other case, there is enough spare time, the synthesis process did not slow down the application anyway and there are no objections against this type of adaptation.

4. Runtime Application Profiling

A major task in synthesizing hardware functional units for AMIDAR processors is runtime application profiling. This allows the identification of candidate instruction sequences for hardware acceleration. Plausible candidates are the runtime critical parts of the current application.

In previous work [28], we have shown a profiling algorithm and corresponding hardware implementation which generate detailed information about every executed loop structure. Those profiles contain the total number of executed instructions inside the affected loop, the loops start program counter, its end program counter, and the total number of executions of this loop. The profiling circuitry is also capable to profile nested loops, not only simple ones. The whole process of profiling is carried out in hardware and does not need a software intervention. As the exact method to evaluate the profiles is not of further interest for the following considerations, we will not explain it in detail here. The interested reader is referred to [28] for a full account of the resulting hardware structures.

A profiled loop structure becomes a synthesis candidate in case its number of executed instructions surmounts a given threshold. The size of this threshold can be configured dynamically for each application.

Furthermore, an instruction sequence has to match specific constraints in order to be synthesized. Currently, we are not capable of synthesizing every given instruction sequence. Bytecode containing the following instruction types cannot be handled as our synthesis algorithm has not evolved to this point yet

(i) memory allocation operations,

(ii) exception handling,

(iii) thread synchronization,

(iv) some special instructions, for example, `lookupswitch`

(v) access operations to multidimensional arrays,

(vi) method invocation operations.

Of these bytecodes memory allocation operations, exception handling, thread synchronization and the special instructions do not matter much as they typically do not occur in compute intensive instruction sequences.

Access to multidimensional arrays does indeed occur in compute kernels and would be important for a broad applicability of the synthesis. The AMIDAR model itself does not prevent this type of array access from synthesized functional units. Unfortunately, our synthesis algorithm does not handle the multistage access up to now. Multidimensional arrays are constructed as arrays of arrays in Java. Thus, such an array may not be allocated as a single-block and the detection of its actual address at runtime is not possible with a simplistic approach. Yet, a manual rewrite of the code is possible to map multidimensional arrays to one-dimensional arrays.

Finally, method invocations in general are not mappable to synthesized functional units. Nevertheless, object-oriented languages often contain getter/setter methods in objects which would break the synthesis of methods. Thus, method inlining should be applied to improve the applicability of the synthesis algorithm. However, care must be taken to deal with polymorphism appropriately. Here, techniques which are usually applied in JIT compilation can be used as well for the synthesized functional unit. In general, the approach is to assume a particular class for the called method. At runtime, the synthesized functional unit has to check whether the given object really belongs to this class. Hardware execution may only occur if this condition is fulfilled. Otherwise, the execution of the synthesized functional unit must be aborted and the original bytecode sequence must be executed instead. Currently, we do not support this special handling in the synthesis algorithm, but we plan to do so in the future.

5. Online Synthesis of Application-Specific Functional Units

The captured data of the profiling unit is evaluated periodically. In case, an instruction sequence exceeds the given runtime threshold, the synthesis is triggered. The synthesis runs as a low-priority process concurrently to the application. Thus, it can only occur if spare computing time remains in the system. Also, the synthesis process cannot interfere with the running application.

5.1. Synthesis Algorithm. An overview of the synthesis steps is given in Figure 5. The parts of the figure drawn in grey are not yet implemented.

Firstly, an instruction graph of the given sequence is created. In case an unsupported instruction is detected, the synthesis is aborted. Furthermore, a marker of a previously synthesized functional unit may be found. If this is the case, it is necessary to restore the original instruction information and then proceed with the synthesis. This may happen if an inner loop has been mapped to hardware before, and then the wrapping loop will be synthesized as well.

Afterwards, all nodes of the graph are scanned for their number of predecessors. In case a node has more than one predecessor, it is necessary to introduce specific Φ-nodes to the graph. These structures occur at the entry of loops or in typical if-else-structures. Furthermore, the graph is annotated with branching information. This will allow the identification of the actually executed branch and the selection of the valid data when merging two or more branches by multiplexers. For if-else-structures, this approach reflects a speculative execution of the alternative branches. The condition of the if-statement is used to control the selection of one set of result values. Loop entry points are treated differently, as no overlapping or software pipelining of loop kernels is employed up to now.

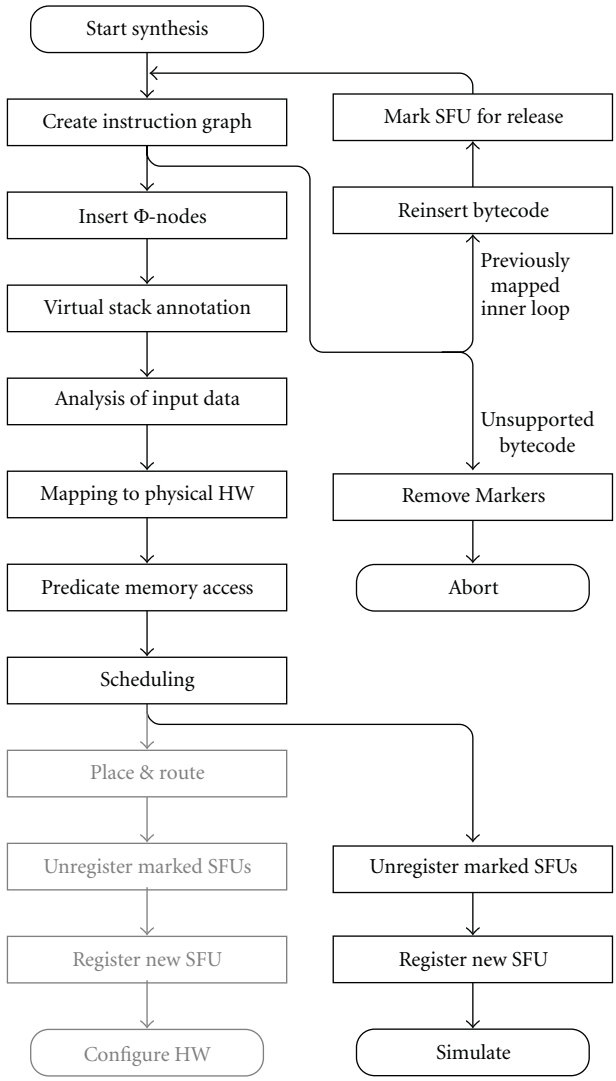


Figure 5: Overview of synthesis steps.

In a second step, the graph is annotated with a virtual stack. This stack does not contain specific data, but the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the immediate predecessor of an instruction may not be the producer of its input.

Afterwards an analysis of access operations to local variables, arrays and objects takes place. This aims at loading data into the functional unit and storing it back to its appropriate memory after the functional units execution. Therefore, a list of data that has to be loaded and a list of data that has to be stored is created.

The next step transforms the instruction graph into a hardware circuit. This representation fits precisely into our simulation. All arithmetic or logic operations are transformed into their abstract hardware equivalent. The introduced Φ-nodes are transformed into multiplexer structures. The annotated branching information helps to connect the different branches correctly and to determine the appropriate control signal. Furthermore, registers and memory structures are introduced. Registers are used to hold values at the beginning and the end of branches in order to synchronize different branches. Localization of memory accesses is an important measure to improve the performance of many potential applications. In general, synthesized functional units could also access the heap to read or write array elements, but this access would incur an overhead of several clocks. The memory structures are connected to the consumer/producer components of their corresponding arrays or objects. A datapath equivalent to the instruction sequence is the result of this step.

Execution of consecutive loop kernels is strictly separated. Thus, all variables and object fields altered in the loop kernel are stored in registers at the beginning of each iteration of the loop.

Arrays and objects may be accessed from different branches that are executed in parallel branches. Thus, it is necessary to synchronize access to the affected memory regions. Furthermore, only valid results may be stored into arrays or objects. This is realized by special enable signals for all write operations. The access synchronization is realized through a controller synthesis. This step takes the created datapath and all information about timing and dependency of array and object access operations as input. The synthesis algorithm has a generic interface which allows to work with different scheduling algorithms.

Currently, we have implemented a modified ASAP scheduling which can handle resource constraints and list scheduling. The result of this step is a finite state machine (FSM) which controls the datapath and synchronizes all array and object access operations. Also the FSM takes care of the appropriate execution of simple and nested loops.

As mentioned above, we do not have a full hardware implementation yet, but we use a cycle accurate simulation. Hence, placement and routing for the CGRA structure are not required, as we can simulate the abstract datapath created in the previous steps.

In case the synthesis has been successful, the new functional unit needs to be integrated into the existing processor. If one or more marker instructions of previously synthesized functional units where found, the original instruction sequence has to be restored. Furthermore, the affected synthesized functional units have to be unregistered from the processor and the hardware used by them has to be released for further use.

The synthesis process is depicted in Figure 6. It shows the initial bytecode sequence and the resulting instruction graph, as well as data dependencies between the instructions and the final configuration of the reconfigurable fabric. The autocorrelation function achieves a speedup of 12.42 on an array with four operators and an input vector of 32 integer values.

5.2. Functional Unit Integration. The integration of the synthesized functional unit (SFU) into the running application consist of three major steps. (1) A token set has to be generated which allows the token generator to use the SFU.

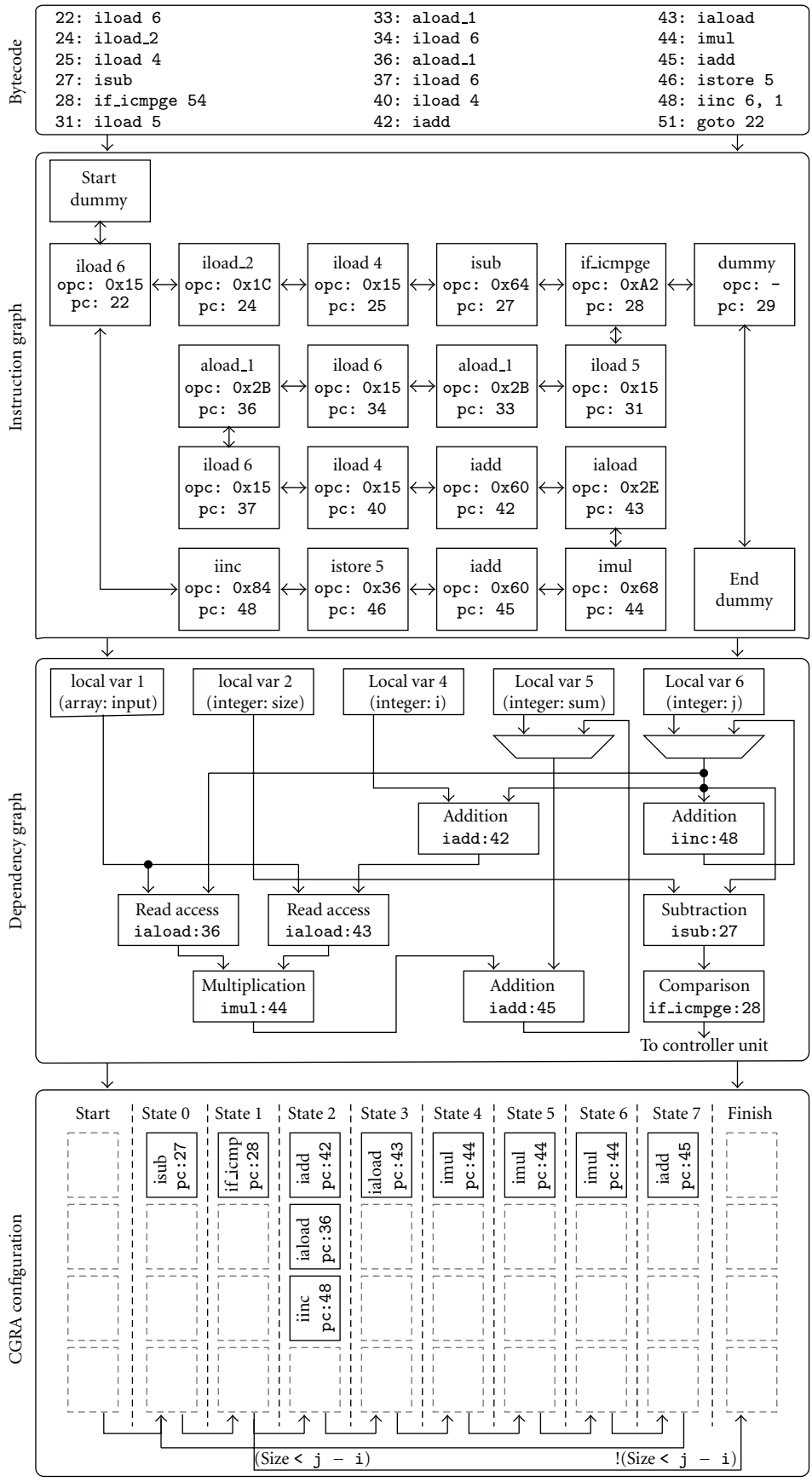


Figure 6: Example of intermediate synthesis results.

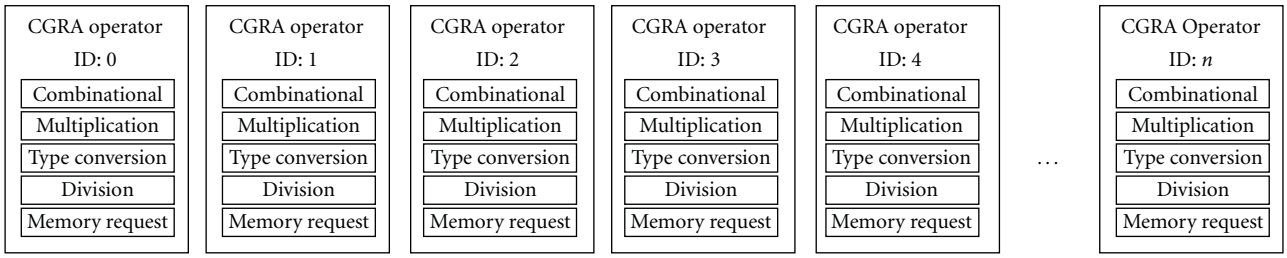


Figure 7: Configuration of the homogeneous coarse grain reconfigurable array.

(2) The SFU has to be integrated into the existing circuit and (3) The synthesized code sequence has to be patched in order to access the SFU. This last step comes with an adjustment of the profiling data which led to the decision of synthesizing an FU.

The generated set must contain all tokens and constant values that are necessary to transfer input data to the SFU, process it, and write back possible output data. All tokens are stored in first-in-first-out structures. This handling assures that tokens and constants are distributed and processed in their determined order. The lists of input and output local variables now can be used to feed the token generation algorithm.

In a second step, it is necessary to create tokens for the controlling of the SFU itself. One token has to be created in order to trigger the SFUs processing when all input data has arrived. A second token triggers the transfer of output data to its determined receiver.

In a last step, the output token sequence for the SFU is created. This set of tokens will be distributed to the functional units which will consume the generated output. The sequence must specify all operations that have to be processed in order to write the functional units output data to its receivers corresponding memory position.

In a next step, it is necessary to make the SFU known to the circuit components. Firstly, is has to be registered to the bus-arbiter. This allows the assignment of bus structures to the SFU. Furthermore, the SFU must be accessible by the token generator. The token generator works on a table which holds a token set for each instruction. Thus, the token generator needs to know a set of tokens which are necessary to control the SFU. This information is passed to the token generator in special data structures which are characterized by the ID of the SFU. This ID is necessary to distribute the correct set of tokens once the SFU is triggered by the new instruction.

The SFU needs to be accessed through a bytecode sequence in order to use it. Therefore, it is necessary to patch the affected sequence. The first instruction of the loop is patched to a specific newly introduced bytecode which signals the use of an SFU. The next byte represents the global identification number of the new SFU. It is followed by two bytes representing the offset to the successor instruction which is used by the token generator to skip the remaining instructions of the loop. In order to be able to revoke the synthesis, it is necessary to store the patched four bytes with the token data. Patching the bytecode must not be done as long as the program counter points to one of the patched locations.

Now, the sequence is not processed in software anymore but by a hardware SFU. Thus, it is necessary to adjust the profiling data which led to the decision of synthesizing a functional unit for a specific code sequence. The profile related to this sequence now has to be deleted, as the sequence itself does no longer exist as a software bytecode fragment.

In [29], we have given further information and a more detailed description of the integration process.

6. Evaluation

In order to evaluate the potential of our synthesis approach, we evaluated a set of benchmark applications. In previous research [30], we have evaluated the potential speedup of a simplistic online synthesis with unlimited resources. To be more realistic, we assumed a CGRA with 16 operators. Every operator is capable of executing the same set of basic functionality, as shown in Figure 7. Furthermore, the CGRA contained a single shared memory for all arrays and objects. As a result, all memory access operations must be synchronized by the scheduling algorithm. The scheduling itself has been calculated by a longest path list scheduling.

The following data-set was gained for every benchmark:

(i) its runtime, and therewith the gained speedup,

(ii) the number of states of the controlling state machine,

(iii) the number of different contexts regarding the CGRA.

The reference value for all measurements is the plain software execution of the benchmarks. Note: the mean execution time of a bytecode in our processor is ~4 clock cycles. This is in the same order as JIT-compiled code on IA32 machines.

6.1. Benchmark Applications. We used applications of four different domains to test our synthesis algorithm. Firstly, we benchmarked several cryptographic ciphers as the importance of security in embedded systems increases steadily. Additionally, we chose hash algorithms and message digests as a second group of appropriate applications. Thirdly, we evaluated the runtime behavior of image processing kernels.

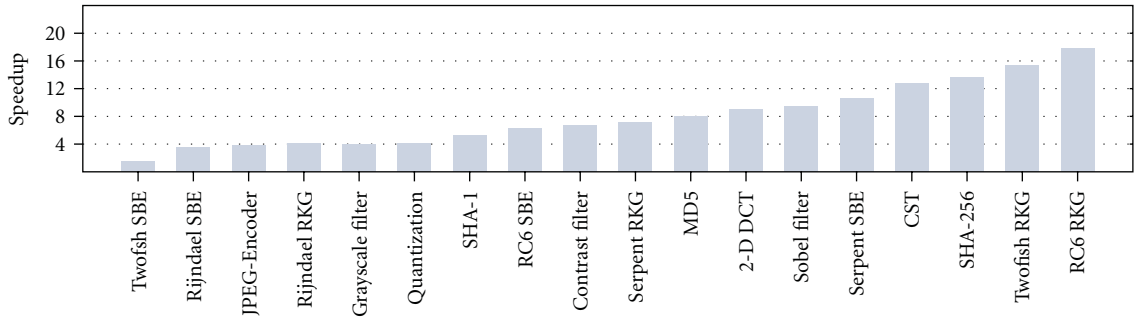


Figure 8: Diagram of runtime acceleration of benchmark applications.

All of these benchmark applications are pure computation kernels. Regularly, they are part of a surrounding application. Thus, we chose the encoding of a bitmap-image into a JPEG-image as our last benchmark application. This application contains several computation kernels, such as color space transformation, 2D forward DCT, and quantization. Nonetheless, it also contains a substantial amount of code that utilizes those kernels, in order to encode a whole image.

The group of cryptographic benchmarks contains four block ciphers. Rijndael, Twofish, Serpent, and RC6 all were part of the Advanced Encryption Standard (AES) evaluation process.

We evaluated the round key generation out of a 256-bit master key, as this is the largest common key length of those ciphers. Furthermore, we reviewed the encryption of a 16-byte data block, which is the standard block size for all of them. We did not examine the decryption of data, as it is basically an inverted implementation of the encryption. Thus, its runtime behavior is mostly identical.

Another typical group of algorithms used in the security domain are hash algorithms and message digests. We chose the Message Digest 5 (MD5) and two versions of the Secure Hash Algorithm (SHA-1 and SHA-256) as representatives. For instance, these digests are heavily utilized during TLS/SSL encrypted communication.

We measured the processing of sixteen 32 bit words, which is the standard input size for those three algorithms.

Thirdly, we rated the effects of our synthesis algorithm onto image processing kernels. Therefore, we chose a discrete differentiation that uses the Sobel convolution operator as one of those tests. This filter is used for edge detection in images. Furthermore, a grayscale filter and a contrast filter have been evaluated. As its name tells, the grayscale filter transforms a colored image into a grayscale image. The contrast filter changes the contrast of an image regarding given parameters for contrast and brightness.

These three filters operate on a dedicated pixel of an image, or on a pixel and its neighbours. Thus, we measured the appliance of every filter onto a single pixel.

Finally, as we mentioned before, we encoded a given bitmap image into a JPEG image. The computation kernels of this application are the color space transformation, 2D forward DCT and quantization. We did not downsample the chroma parts of the image. The input image we have chosen has a size of 160×48 pixels, which results in 20×6 basic blocks of 8×8 pixels. Thus, every one of the mentioned processing steps had been executed 120 times for each of the three color components, which results in a total of 360 processed input blocks.

6.2. Runtime Acceleration. First benchmarks with "out-of-the-box" code have shown no improvement in speed through the synthesis for all applications, except from the contrast and grayscale filter. Analysis of these runs have pointed to a failed synthesis due to unsupported bytecodes. The crucial parts of those applications contained either method invocations or access to multidimensional arrays. As we mentioned above, these instruction types are not supported by our synthesis algorithm yet. In order to show the potential of our algorithm, we inlined the affected methods and flattened the multidimensional arrays to one dimension.

The subsequent evaluations have shown sophisticated results. The gained speedup ranges from 1.52 to 17.84. The best results of the cryptographic cipher benchmarks were achieved by the round key generation of RC6 and Twofish, and furthermore the encryption of Serpent. All of them could gain speedups larger than 10. The smallest improvement of factor 1.5 was obtained by the Twofish encryption. This is clearly an outlier, which comes due to a very large communication overhead of the synthesized functional unit.

The hash algorithms we chose for our test achieved speedups from 5.25 to 13.63. The best result has been achieved by SHA256, while SHA1 has gained the weakest speedup. The MD5 could be sped up by a factor of eight.

The speedups of the image processing kernels range from 4.00 to 9.41. As the application of a grayscale filter to a pixel is a simple operation, it could only be sped up by the factor of four. The contrast filter could be accelerated by a factor of 6.76, while the Sobel convolution achieved the best result with a speedup of 9.41.

The JPEG encoding application as a whole has gained a speedup of 3.77. The computation kernels themselves achieved better results. The quantization has been sped up by a factor of 4.10, while the 2D forward DCT improved by a factor of 9.06. The largest speedup of 12.74 has been gained by the color space transformation.

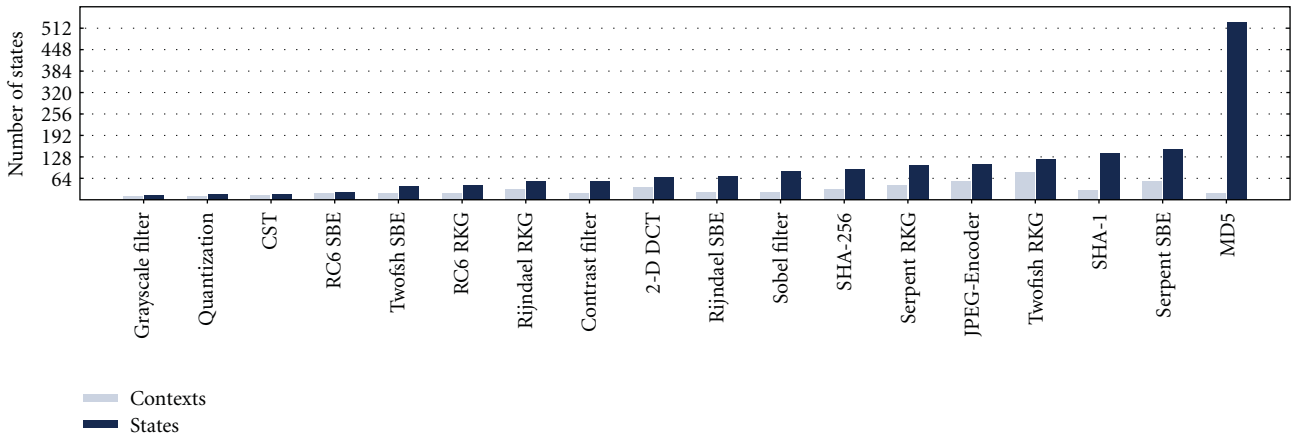


Figure 9: Diagram of complexity of the schedules of benchmark applications.

Table 1: Runtime acceleration of benchmark applications.

(a) Round key generation of cryptographic cipher benchmarks

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17760	—	525276	—	61723	—	44276	—
Synthesis enabled	4337	4.09	34112	15.40	3459	17.84	6230	7.11

(b) Single block encryption of cryptographic cipher benchmarks

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21389	—	12864	—	17371	—	34855	—
Synthesis enabled	6167	3.47	8452	1.52	2768	6.28	3273	10.65

(c) Hash algorithms and message digests

Configuration	SHA-1		SHA-256		MD5	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	23948	—	47471	—	11986	—
Synthesis enabled	4561	5.25	3484	13.63	1485	8.07

(d) Image processing kernels

Configuration	Sobel convolution		Grayscale filter		Contrast filter	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21124	—	236	—	608	—
Synthesis enabled	2246	9.41	59	4.00	90	6.76

(e) JPEG-encoding and its application kernels

Configuration	JPEG-encoder		Color space transformation		Forward DCT		Quantization	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17368663	—	3436078	—	23054	—	7454	—
Synthesis enabled	4612561	3.77	269702	12.74	2545	9.06	1816	4.10

Table 2: Complexity of the schedules of benchmark applications.

(a) Round key generation of encryption of cryptographic cipher benchmarks

Rijndael		Twofish		RC6		Serpent	
States	Contexts	States	Contexts	States	Contexts	States	Contexts
55	31	122	83	44	20	103	42

(b) Single-block encryption of encryption of cryptographic cipher benchmarks

Rijndael		Twofish		RC6		Serpent	
States	Contexts	States	Contexts	States	Contexts	States	Contexts
69	23	40	20	23	19	152	54

(c) Hash algorithms and message digests

SHA-1		SHA-256		MD5	
States	Contexts	States	Contexts	States	Contexts
138	29	92	30	531	20

(d) Image processing kernels

Sobel convolution		Grayscale filter		Contrast filter	
States	Contexts	States	Contexts	States	Contexts
86	23	13	9	56	18

(e) JPEG-encoding and its application kernels

JPEG-encoder		Color space transformation		Forward DCT		Quantization	
States	Contexts	States	Contexts	States	Contexts	States	Contexts
105	55	17	14	67	36	16	11

The runtime results for all of the benchmarks are displayed in Figure 8, while the corresponding measurements are shown in Table 1.

6.3. Schedule Complexity. In a next step, we evaluated the complexity of the controlling units that were created by the synthesis. Therefore, we measured the size of the finite state machines, that are controlling every synthesized functional unit. Every state is related to a specific configuration of the reconfigurable array. In the worst case, all of those contexts would be different. Thus, the size of a controlling state machine is the upper bound for the number of different contexts.

Afterwards, we created an execution profile for every context. This profile contains a key for every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of elements in this resulting set is a lower bound for the number of contexts that are necessary to drive the functional unit. The effective number of necessary configurations lies between those two bounds, as it depends on the place-and-route results of the affected operations.

The context information for the benchmarks is displayed in Figure 9, while the actual number of states and contexts is given in Table 2. It shows the size of the controlling finite state machine (States), and the number of actually different contexts (Contexts) for every of our benchmarks. It can be seen, that only three of eighteen state machines consist of more than 128 states. Furthermore, the bigger part of the state machines contains a significant number of identical states regarding the executed operations. Thus, the actual number of contexts is well below the number of states.

7. Conclusion

In this article, we have shown a simplistic online-synthesis algorithm for AMIDAR processors. It is capable of synthesizing functional units fully automated at runtime regarding given resource constraints. The target technology for our algorithm is a coarse-grained reconfigurable array. We assumed a reconfigurable fabric with homogeneously formed processing elements and one shared memory for all objects and arrays.

The displayed synthesis approach targets maximum simplicity and runtime efficiency of all used algorithms. Therefore, we used list scheduling as scheduling algorithm and passed on more complex methods like software pipelining. Furthermore, we have not run optimization algorithms on the finite state machines that were created by our synthesis.

In order to demonstrate the capabilities of our algorithm, we have chosen four groups of benchmark applications. Those applications were cryptographic ciphers, message digests, graphic filters, and the JPEG encoding process. All benchmarks could be accelerated by the synthesis, and the mean speedup that has been achieved is 7.95. Furthermore, we displayed the complexity of the gained synthesis results. This evaluation showed that most of our benchmarks are to be driven by less than 128 contexts.

8. Future Work

The full potential of online synthesis in AMIDAR processors has not been reached yet. Future work will concentrate on improving our existing synthesis algorithm in multiple ways. This includes the implementation of access to multidimensional arrays and inlining of invoked methods at synthesis time. Additionally, we will explore the effects of instruction chaining in synthesized functional units. Furthermore, we are planning to overlap the transfer of data to a synthesized functional unit and its execution. We are planning to introduce an abstract description layer to our synthesis. This will allow easier optimization of the algorithm itself and will open up the synthesis for a larger number of instruction sets. Currently, we are able to simulate AMIDAR processors based on different instruction sets, such as LLVM-Bitcode, .NET Common-Intermediate-Language, and Dalvik-Executables. In the future, we are planning to investigate the differences in execution of those instruction sets in AMIDAR-processors.

References

[1] S. Vassiliadis and D. Soudris, *Fine-and Coarse-Grain Reconfigurable Computing*, Springer, New York, NY, USA, 2007.

[2] C. Hochberger, R. Hoffmann, K.-P. Völkmann, and S. Waldschmidt, "The cellular processor architecture CEPRA-1X and its conguration by CDL," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '00)*, pp. 898–905, 2000.

[3] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 189–208, 2007.

[4] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, pp. 12–21, April 1997.

[5] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 507–512, June 2000.

[6] A. Koch and N. Kasprzyk, "High-level-language compilation for reconfigurable computers," in *Proceedings of the International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC '05)*, pp. 1–8, 2005.

[7] Y. Ha, R. Hipik, S. Vernalde et al., "Adding hardware support to the hotSpot virtual machine for domain specific applications," in *Proceedings of the International Conference on Field Programmable Logic (FPL '02)*, pp. 1135–1138, 2002.

[8] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD—reconfigurable pipelined datapath," in *Proceedings of the International Conference on Field Programmable Logic (FPL '96)*, pp. 126–135, 1996.

[9] Y. Chou, P. Pillai, H. Schmit, and H. P. Shen, "PipeRench implementation of the instruction path coprocessor," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '00)*, pp. 147–158, 2000.

[10] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable Kress Arrays," in *Proceedings of the International Conference on Field Programmable Logic (FPL '99)*, pp. 385–390, 1999.

[11] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the Design, Automation and Test in Europe (DATE '03)*, pp. 10296–10301, 2003.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the International Conference on Field Programmable Logic (FPL '03)*, pp. 61–70, 2003.

[14] W. Kehuai, A. Kanstein, J. Madsen, and M. Berekovic, "MT-ADRES: multithreading on coarse-grained reconfigurable architecture," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '07)*, pp. 26–38, 2007.

[15] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.

[16] G. Hempel, C. Hochberger, and A. Koch, "A comparison of hardware acceleration interfaces in a customizable soft core processor," in *Proceedings of the International Conference on Field Programmable Logic (FPL '10)*, pp. 469–474, 2010.

[17] C. Rupp, Multi-scale programmable array (US patent 6633181), October 2003.

[18] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: rotating instruction set processing platform," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 791–796, June 2007.

[19] R. L. Lysecky and F. Vahid, "Design and implementation of a microblaze-based WARP processor," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–22, 2009.

[20] A .C .S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility," in *Proeedings of the Design Automation Conference (DAC '05)*, pp. 732–737, September 2005.

[21] J. C. Dehnert, B. K. Grant, J. P. Banning et al., "The transmeta code morphing™ software: using speculation,recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, pp. 15–24, 2003.

[22] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, 2008.

[23] S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer, "A Two-dimensional Superscalar processor architecture," in *Proceedings of the Computation World: Future Computing, Service Computation, Adaptive, Content, Cognitive, Patterns (ComputationWorld '09)*, pp. 608–611, 2009.

[24] H. Corporaal, *Microprocessor Architectures: From VLIWto TTA*, John Wiley & Sons, New York, NY, USA, 1997.

[25] S. Gatzka and C. Hochberger, "A new general model for adaptive processors," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, pp. 52–60, June 2004.

[26] S. Gatzka and C. Hochberger, "The AMIDAR class of reconfigurable processors," *Journal of Supercomputing*, vol. 32, no. 2, pp. 163–181, 2005.

[27] S. Gatzka and C. Hochberger, "The organic features of the AMIDAR class of processors," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '05)*, pp. 154–166, 2005.

[28] S. Gatzka and C. Hochberger, "Hardware based online profiling in AMIDAR processors," in *Proceedings of the19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005.

[29] S. Döbrich and C. Hochberger, "Towards dynamic software/hardware transformation in AMIDAR processors," *Information Technology*, vol. 50, no. 5, pp. 311–316, 2008.

[30] S. Döbrich and C. Hochberger, "Effects of simplistic online synthesis for AMIDAR processors," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 433–438, December 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 603059, 12 pages
doi:10.1155/2010/603059

Research Article

3D Network-on-Chip Architectures Using Homogeneous Meshes and Heterogeneous Floorplans

Vitor de Paulo and Cristinel Ababei

Electrical and Computer Engineering Department, North Dakota State University, Fargo, ND 58108-6050, USA

Correspondence should be addressed to Cristinel Ababei, cristinel.ababei@ndsu.edu

Received 19 February 2010; Accepted 25 August 2010

Academic Editor: Lionel Torres

We propose new 3D 2-layer and 3-layer NoC architectures that utilize *homogeneous* regular mesh networks on a separate layer and one or two *heterogeneous* floorplanning layers. These architectures combine the benefits of compact heterogeneous floorplans and of regular mesh networks. To demonstrate these benefits, a design methodology that integrates floorplanning, routers assignment, and cycle-accurate NoC simulation is proposed. The implementation of the NoC on a separate layer offers an additional area that may be utilized to improve the network performance by increasing the number of virtual channels, buffers size, or mesh size. Experimental results show that increasing the number of virtual channels rather than the buffers size has a higher impact on network performance. Increasing the mesh size can significantly improve the network performance under the assumption that the clock frequency is given by the length of the physical links. In addition, the 3-layer architecture can offer significantly better network performance compared to the 2-layer architecture.

1. Introduction

3D integration is emerging as an attractive solution to the problem of increasing global interconnect delay of integrated systems [1–4]. The main advantage of 3D integration technologies is that the footprint area of the chip is smaller compared to the 2D case. Therefore, because the connections between device layers can be realized by short and reduced delay through silicon vias (TSVs), the average interconnect delay is significantly shorter. However, 3D integration technologies face challenges related to thermal issues.

Network-on-Chip (NoC) represents a new design paradigm for increasingly complex Systems-on-Chip (SoC), and since the idea of routing packets instead of wires was proposed [5–7], it has grown into a rich research topic [8–10]. The NoC concept replaces design-specific global on-chip wires with a generic on-chip interconnection network realized by specialized routers that connect generic processing elements (PE)—such as processors, ASICs, FPGAs, memories, and so forth—to the network and facilitate communications or links between them. The benefits of the NoC based SoC-design include scalability, predictability, and higher bandwidth with support for concurrent communications.

The scalability and predictability of NoCs enable designers to design increasingly complex systems, with large numbers of IP/cores and lower communication latencies for many applications. In such scenarios, where flexibility and predictability are primary concerns, homogeneous regular networks (see Figure 1(a)) are preferred. However, homogeneous NoC topologies have limitations in that communication locality is poorly supported, and the utilization of network resources is low. Moreover, designs with IP/cores with different sizes are not well suited to implementations based on regular mesh NoC topologies. Therefore, when area and performance are more important, application-specific heterogeneous irregular networks (see Figure 1(b)) are preferred. However, the design of these networks is more difficult and specialized routing algorithms are necessary to prevent deadlock.

Most of the previous works assumed equal area for all tiles. This assumption simplifies the design process due to the regularity of the mesh NoC topologies. However, assuming tiles with equal area in cases where IP/cores have different

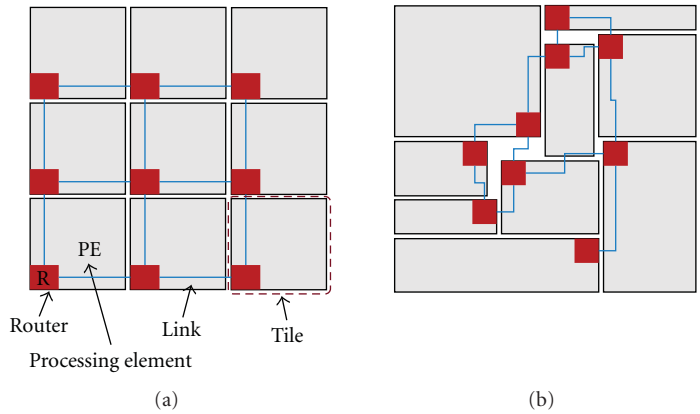


Figure 1: (a) 2D homogeneous NoC design. A tile is composed of a router (R) and a generic processing element (PE). A PE can implement any IP/core of a given application. Routers are interconnected via physical links. (b) Custom heterogeneous NoC design.

sizes is unrealistic. To address this problem for systems with heterogeneous floorplans and to exploit the benefits of 3D integration, in this paper, we propose new 3D NoC architectures. The proposed architectures utilize homogeneous networks on a separate layer and heterogeneous floorplans on different layers. Our objective is to combine the benefits of compact heterogeneous floorplans with those of regular homogeneous mesh networks.

2. Related Work

In this section, we discuss recent works on 3D NoCs and studies that utilize floorplanning information during the design and optimization of NoCs. The reader is referred to [11, 12] for recent surveys of NoCs. Nanophotonic and wireless NoCs have also been recently proposed as alternative solutions to 2D architectures. However, they pose several issues including scaling and integration of photonic devices and power dissipation of mm-wave transceivers [13].

Performance benefits of 3D NoC topologies were investigated analytically in [14] and experimentally in [15]. The transition from 2D to 3D NoC architectures is done by equally distributing tiles onto the device layers of the 3D architecture in [16, 17]. A different approach is to reduce the footprint of each tile by implementing the processing element and the router in a distributed fashion across layers [18]. By leveraging long wires to connect remote intralayer nodes, a low-diameter 3D network is studied in [19]. Efficient application-specific 3D NoC topology synthesis algorithms are studied in [20]. A novel layer-multiplexed 3D network architecture with vertical demultiplexing and multiplexing links is proposed in [21]. The scalability of 3D NoCs in terms of throughput, latency, and area overhead is studied in [22]. The per-flow worst-case communication performance in 2D and 3D regular mesh NoCs with four layers is investigated in [23]. The first demonstration of a fabricated 3D NoC is reported in [24]. Previous works focused on homogeneous regular NoCs and floorplans where all PEs have equal size. In this paper, the proposed architectures provide a design solution to applications where PEs are heterogeneous with different sizes. Hence, the proposed 3D NoC architectures represent an alternative solution to heterogeneous irregular NoC topologies.

Floorplanning information is used in the area-wirelength calculations from [25, 26] or during mapping [27]. A floorplanner is used to compute links power consumption and to detect timing violations in application-specific NoCs in [28]. The NoC synthesis approach from [28] was extended to designing custom 3D network topologies in [29]. Slicing floorplanning is used in the design methods for custom NoC topology synthesis studied in [30–32]. Frequently communicating modules are placed next to each other using a floorplanner in the network synthesis heuristic studied in [33]. A physical planner is used in [34] during topology design to reduce power consumption on wires. Previous works use floorplanning either for wirelength and power estimations or to find a *single* placement, which then remains fixed throughout the topology synthesis process. However, other floorplanning solutions may represent better starting placements for the synthesis process. To address this problem, the methodology proposed in this paper explores multiple floorplans to increase the chance of finding the optimal initial placement.

3. Contribution

In this paper, we propose novel 3D NoC architectures and implement an automated design space-exploration tool. Our main contribution can be summarized as follows.

(i) We propose and study two 3D NoC architectures (2-layer and 3-layer architectures) that utilize a homogeneous network on a separate layer and heterogeneous floorplans on different layers. In this way, the network regularity is maintained for flexibility and delay predictability while the IP/cores can have arbitrary sizes. This approach avoids design difficulties due to IP/core size irregularities that are typically addressed by specialized routing algorithms [35].

(ii) For the 2-layer architecture, we propose the use of a floorplanning and routers assignment-based design methodology for the placement of IP/cores on the

first layer and the minimization of their connections to the NoC routers located on the second layer. In the case of the 3-layer architecture, the design methodology also includes a partitioning step. The floorplanning of the two partitions on layers 1 and 3 is done using a newly modified floorplanner capable of handling vertical constraints. One advantage of the separation between IP/cores and network is that once the best floorplan is found, one can focus on improving the system performance by focusing on the network. The second layer has an additional available area that may be utilized to increase the number of routers or their complexity (e.g., increase the number of virtual channels and/or the buffers size). In addition, network interfaces (NIs), which are important components of NoC-based systems, also may be placed on the second layer, thereby reducing the footprint area of the floorplanning layers.

(iii) We implemented a versatile software framework to investigate the benefits of the proposed 3D architectures. It integrates an efficient B*Tree-based floorplanner with a cycle-accurate NoC simulator for maximum confidence in the experimental results.

Preliminary results on the 2-layer NoC architecture were reported in [36]. In this paper, we also propose the second 3-layer NoC architecture that aims at further reducing the footprint area of the chip and at improving the average flit latency. Due to the differences from the first proposed architecture, we also modify the design methodology by introducing an additional partitioning step and enhancing the floorplanning algorithm to handle vertical constraints.

4. 3D Architectures

4.1. 2-Layer Architecture. The *2-layer architecture* has two device layers. The first layer is used entirely for the *heterogeneous irregular* IP/cores, while the second layer is dedicated to the *homogeneous regular* NoC (see Figure 2(b)). This approach simplifies the design process in that it separates the floorplanning optimization from the network topology synthesis. The goal of the floorplanning step is to find the best floorplan with minimal white space. The second device layer accommodates the regular mesh network. In this way, the network regularity is maintained for flexibility and delay predictability, while the IP/cores can have arbitrary sizes. In addition, a simple packet routing algorithm can be used such as the deterministic XY routing.

4.2. 3-Layer Architecture. The *3-layer architecture* has three device layers. The second layer is again dedicated to implementing the NoC, while layers 1 and 3 are used for IP/cores placement (see Figure 2(c)). This architecture aims at reducing the footprint area of the chip, which in turn leads to shorter physical links, hence improving the network performance (overall average flit latency). In both proposed architectures, the vertical connections between IP/cores and their assigned routers are realized using through silicon vias (TSV). Routers connected to IP/cores have five ports, while the rest of the routers have only four ports.

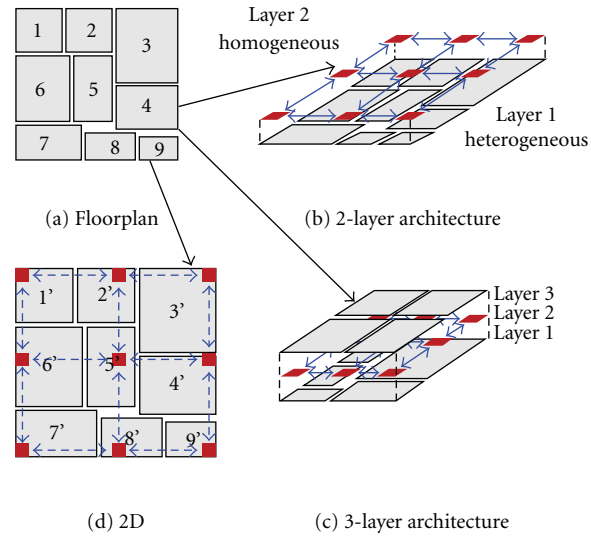


FIGURE 2: (a) Initial floorplan with no routers. (b) 2-layer architecture. (c) 3-layer architecture. (d) 2D implementation with each IP/core inflated to account for area required by network interfaces, routers and wires for physical links.

One advantage of the proposed 3D NoC architectures is that the 3D fabrication will be simpler compared to 3D architectures with more than three layers [17], as the misalignment is only between two or three layers. In addition, the thermal management of fewer layers also will be simpler. Moreover, the extra space available on the second layer may be used to increase the number of routers or their complexity. The additional area may be utilized to implement fault/error tolerance techniques such as error-correcting codes to address crosstalk issues or could be allocated to additional wires to increase the bandwidth of physical links and therefore improve the overall network performance. Alternatively, the extra area also may be utilized to implement thermal monitoring and management schemes [37], to implement buffers for pipelining the physical links, or to incorporate reconfigurability capabilities [38].

5. Proposed Design Methodology

The proposed design methodology is presented in Figure 3. The input to the design flow is the application represented as a communication task graph (CTG) whose tasks have been mapped to *floating* IP/cores using existing mapping algorithms [39]. By floating it is meant that the location of these tasks is yet to be determined during the floorplanning step. In addition, the user can specify several control parameters including the number of different floorplans to be explored N, the number of best floorplans M recorded in the *best M list* and evaluated later using the integrated cycle-accurate simulator and the mesh size $R \times R$. The main steps of the proposed design methodology are described in the following sections.

5.1. Partitioning of the CTG. This step is done only for the 3-layer architecture. Because in this case, the IP/cores are placed on two layers (1 and 3), we first partition the CTG of each application into two subgraphs (see Figure 4), which will be placed by the floorplanner in the next step. The bipartitioning is done such that the total area of the cores in each partition is balanced while the number of arcs (an arc represents a source-destination communication pair in the communication task graph) cut is minimized. The two partitions have to be balanced to minimize the footprint area of the 3D chip, which will be determined by the maximum area of the accumulated area of the blocks in each partition. The minimization of the cut size between the two partitions has as a result that highly connected cores are placed on the same layer. This, as observed in our experiments, helps these cores to be floorplanned closer to each other, which in turn leads to better overall latencies.

For this step, we use the well-known hMetis partitioner [40, 41]. hMetis is a multilevel move-based partitioner, which can achieve balanced and minimum cutsize partitions very efficiently.

5.2. Exploration of N and Recording of Best M Floorplans. The integrated floorplanner is based on the B*Tree representation from [42]. It employs a simulated annealing-based algorithm, with a cost function that combines area and wirelength (user can specify $\alpha \in [0, 1]$)

$$\text{Cost function} = \alpha \cdot \text{Area} + (1 - \alpha) \cdot \text{Wire Length}. \quad (1)$$

Connections between cores are weighted by the communication volume (available from the CTG) so that the resulting floorplanning solution minimizes first the connections with higher communication volume.

A number of N different floorplans are generated by running the floorplanner N times with the selected weights for area and wirelength and with different seeds for the internal random number generator. During this step, a number of best $M \leq N$ floorplans are recorded in the *best M list*. The selection is made according to the chosen criterion of smaller area or shorter total wirelength, which is related to the total communication volume inside the application. The default values of N and M are $N = 30$, $M = 10$. However, they also may be specified by the user. A typical result after floorplanning is shown in Figure 5.

For the 3-layer architecture, the floorplanning step is different. In this case, cores are placed on layers 1 and 3 as dictated by the result of the partitioning step. To do that, we have modified the floorplanning algorithm to be able to handle *vertical constraints*. As a result, this step is split into two substeps (i) In the first substep, the first partition is floorplanned on layer 1 using the original version of the floorplanning algorithm (this is similar to the 2-layer architecture). (ii) During the second sub-step, the second partition is floorplanned on layer 3 using the modified floorplanner. In this case, connections between cores of this partition and cores of the first partition (already placed and fixed on layer 1) act as vertical constraints for the floorplanning process on layer 3. Vertical constraints aim at minimizing the overall wirelength of the top-level application. Intuitively, cores on layer 3 connected in the top-level communication task graph to cores on layer 1 should be overlapping or vertically aligned to shorten the communication distance via the network on layer 2. A typical result after floorplanning for the 3-layer architecture is shown in Figure 6.

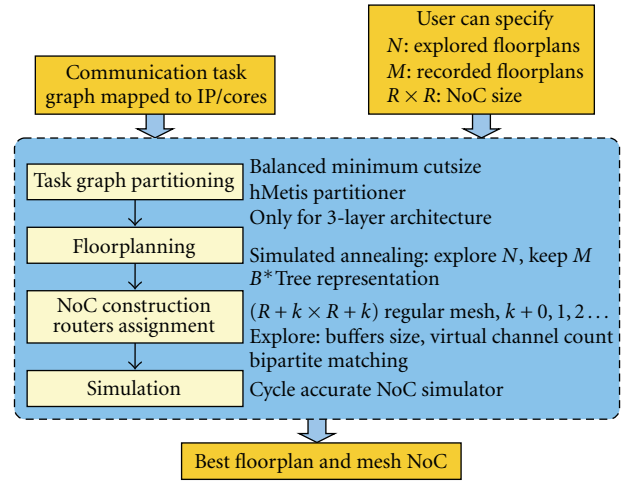
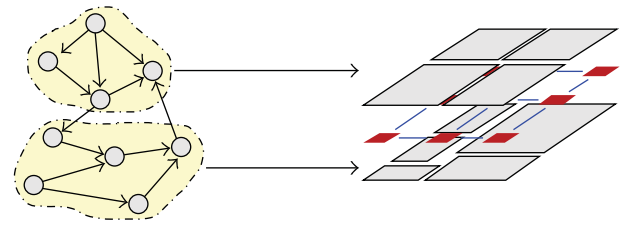


Figure 3: Proposed NoC design exploration methodology.



(a) bi-partitioning (b) 3-layer architecture

Figure 4: Bipartitioning of the application communication task graph and assignment of the two partitions to layers 1 and 3 of the 3-layer architecture.

5.3. Routers Assignment. In this step, each floorplan from the list of best M floorplans undergoes the routers assignment step. The regular $R \times R$ mesh NoC is constructed on layer 2. This square regular mesh network utilizes the minimum number of routers that can guarantee at least one router for each IP/core. This topology is referred to as the *direct topology*. However, the mesh can optionally be expanded to a larger number of routers in both x, y directions. Since we deal with heterogeneous floorplans, it is not possible to guarantee the presence of routers at the locations of IP/core corners (or even the IP/core layout). Therefore, some of the IP/cores will have to use *extralinks* to connect to the assigned routers. These extra-links introduce additional delays (included inside the cycle-accurate simulator) that affect the overall performance.

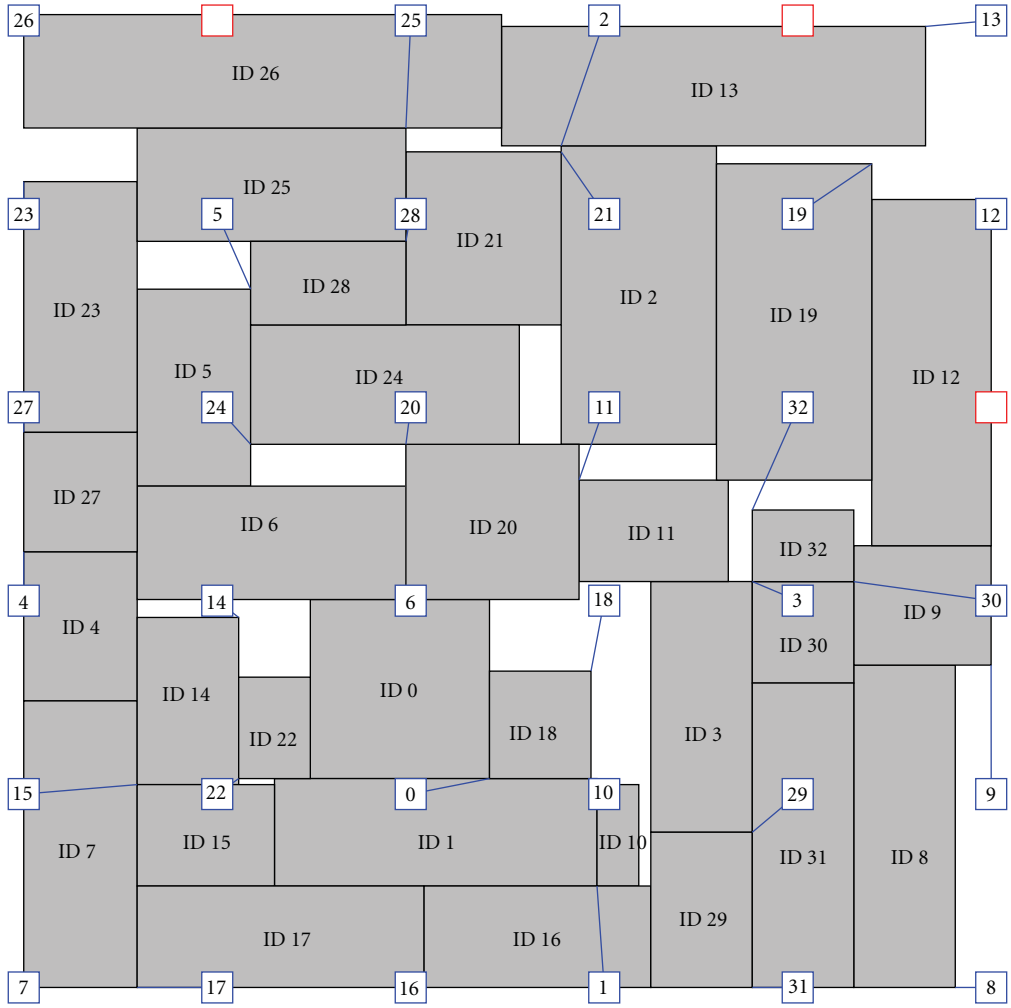
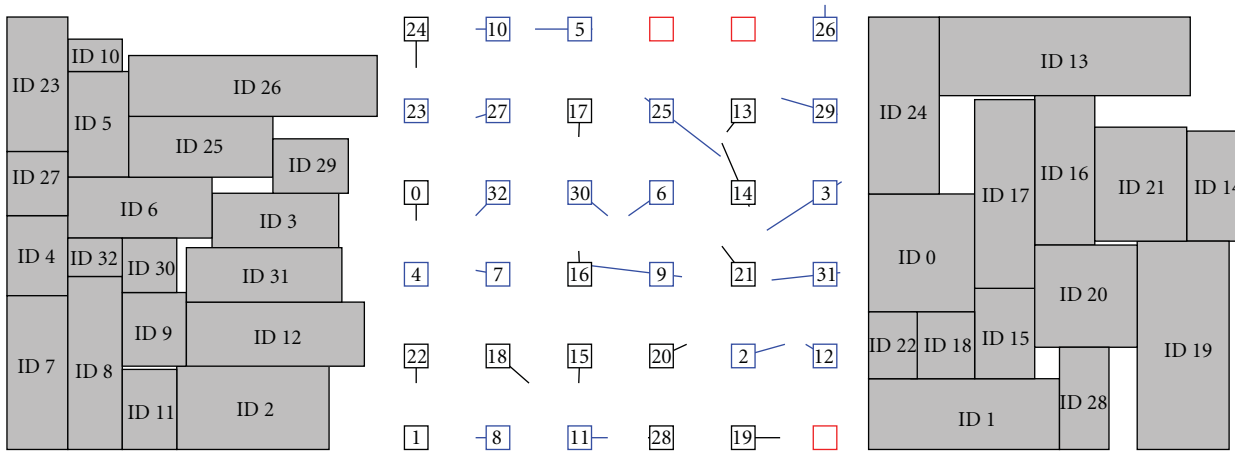


FIGURE 5: Screen capture with the result of floorplanning and routers assignment for *ami33*. The NoC is a direct topology of 6×6, which is the minimum to guarantee at least a router for each IP/core. The extralinks are shown as straight lines between cores and assigned routers.



(a) Layer 1 (b) Layer 2 (c) Layer 3

FIGURE 6: Floorplanning and routers assignment for *ami33* using the 3-layer architecture.

Table 1: Characteristics of the used testcases.

Testcase	Number of IP/cores	Core avg. W/H	Core std. dev. of W/H	Direct topology $R \times R$
apte	8	4324/2499	27/4	3×3
xerox	10	2114/2872	335/1290	4×4
hp	11	4533/924	2498/386	4×4
ami25	25	1770/1408	1201/896	5×5
ami33	33	1581/1573	830/865	6×6
ami49	49	1089/1123	768/651	7×7

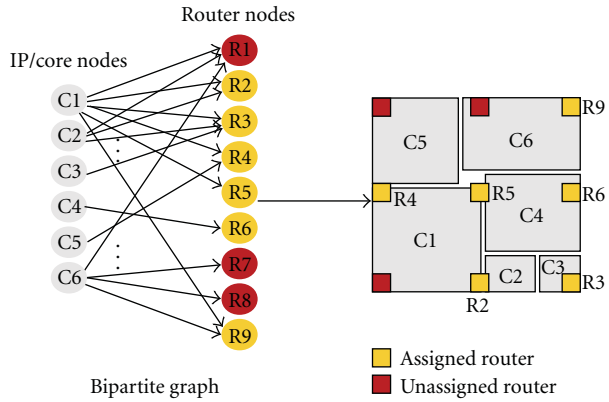


Figure 7: Illustration of the routers assignment step on a testcase with 6 IP/cores and a regular mesh network of 3×3 routers.

The goal of the routers assignment step is to associate each IP/core with a router from the regular mesh on layer 2 such that the total wirelength of the extra-links between each IP/core and its assigned router is minimized. This is a linear assignment problem solved by using the efficient Kuhn-Munkres algorithm [43]. The algorithm utilizes a bipartite graph (see Figure 7) with two sets of nodes: left-nodes representing the application IP/cores and right-nodes representing the routers of the regular mesh NoC. Edges connect each node from one set to all nodes in the other set. Edge weights are proportional to the Manhattan distance between the IP/core and routers. In this way, we treat the assignment of all IP/cores *simultaneously* and achieve an overall minimal total length of the extra-links. This step is the same for both 2-layer and 3-layer architectures. The examples from Figures 5 and 6 also show the result of the routers assignment step.

5.4. NoC Simulation. In the last step, each of the best M NoC topologies is verified using the integrated cycle-accurate simulator. The simulator is an adapted version of the one studied in [37]. We use the following default values for the NoC topology: packet size of 5 flits with each flit being 64 bits wide, input buffer size of 12 flits, and two virtual channels. We use XY routing and wormhole flow control, which is known to be very efficient and requiring small hardware overheads. The cycle-accurate simulator is always run until all injected flits reached their destination and the average latency is computed allowing first 1000 warm-up cycles. The router architecture is similar to the one presented in [44]. The final average flit latency, which is obtained during this step, is recorded for each of the floorplans from the best M list. The NoC topology with the best overall latency is selected as the final result.

Finally, we note that ideally, one would use the routers assignment and the cycle-accurate simulation inside the optimization loop of the simulated annealing based floorplanning algorithm (the concept of unifying different design flow steps to better explore the design solution space has been applied successfully for example to mapping and routing in [45].) However, this becomes computationally too expensive due to the long CPU runtimes required by the cycle-accurate simulator.

6. Experimental Results

6.1. Experimental Setup and Testcases. We implemented the proposed design methodology, which integrates the partitioner, the floorplanner, the routers assignment, the NoC cycle-accurate simulator, and the GUI, using C++. The tool can be downloaded from [46]. In our experiments, we used six testcases whose characteristics are shown in Table 1. In this table, we also present the size of the *direct topologies*. We constructed these testcases from the classic MCNC testcases, whose area was scaled to achieve an average size of about $1\,\text{cm} \times 1\,\text{cm}$, which is a typical area for NoCs reported in the literature [24, 47–49]. The initial connectivity between the modules was used to compute the communication volume in the communication task graph associated with each testcase floorplan.

For the simulated annealing-based floorplanning step, we used an *alpha* value of 0.25, which in our experiments proved to be a good balance between area and wirelength while the aspect ratio of the resulting floorplan was close to 1. In the NoC simulation step, each testcase was subject to uniform traffic with packets injected at each source router at a rate proportional to the communication volume of the corresponding source-destination communication pair.

Because in our methodology the length of the physical links between the network routers varies with the network size, we estimate the link delay by extrapolating the physical link delay from [37] using a simple Elmore delay formula [50]. The same delay estimation technique was applied to the extra-links between IP/cores and routers, which were

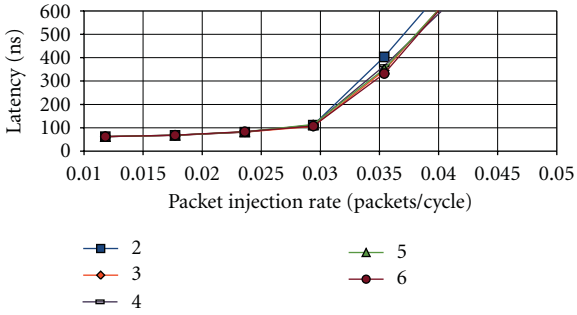


Figure 8: Latency as number of virtual channels is varied for *apte*.

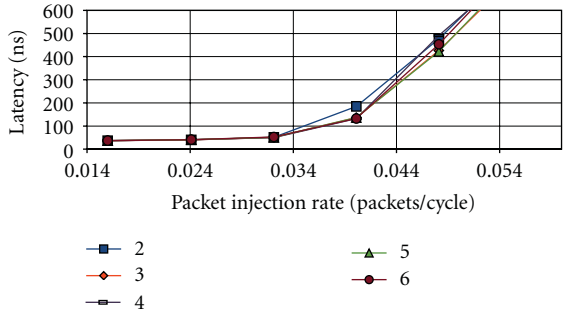


Figure 9: Latency as number of virtual channels is varied for *xerox*.

assumed to be L-shaped (with negligible via delay between metal layers). We do, however, consider the delay of the through silicon vias (TSVs) between two device layers of the 3D architectures. We estimated the TSV delay by technology projection [51] using the delay data from [17]. Based on the analyses in [17, 52], we assume that the area required by TSVs is negligible and that TSVs can be accommodated within the white space available in typical floorplans. The CPU runtime is approximately 30 minutes (Linux machine, 2.5 GHz, 2 GB memory) for the largest testcase *ami49*.

6.2. Exploration of the 2-Layer Architecture. In the first part of the experiments, several variations are applied to the default network specifications. The purpose of these experiments is to identify the optimal network that minimizes the average flit latency.

6.2.1. Varying the Virtual Channels Count. We start by investigating the impact of increasing the number of virtual channels. We can afford to do that because routers are expected to be smaller than the average core size (roughly 20% of the total cores area), which means that on layer 2 there is extra area that may be utilized to further improve the NoC performance. In this experiment, we varied the number of virtual channels between 2 and 6.

The results for the average flit latency (as reported by the cycle-accurate NoC simulator) are shown in Figures 8, 9, 10, 11, 12, and 13. We observe that the average flit latency generally improves with the increase of the number of virtual channels. We also plot (see Figure 14) the normalized latency

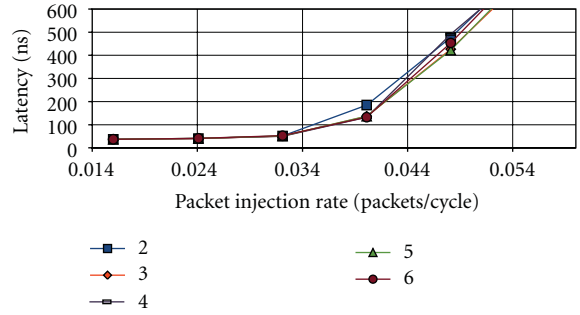


Figure 10: Latency as number of virtual channels is varied for *hp*.

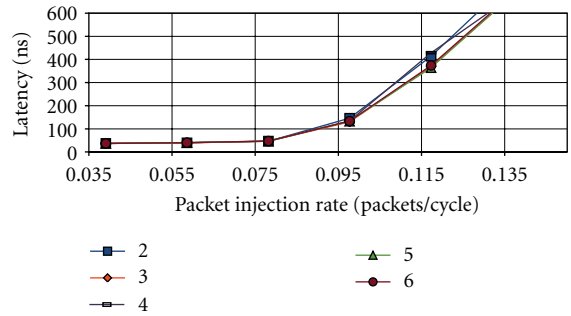


Figure 11: Latency as number of virtual channels is varied for *ami25*.

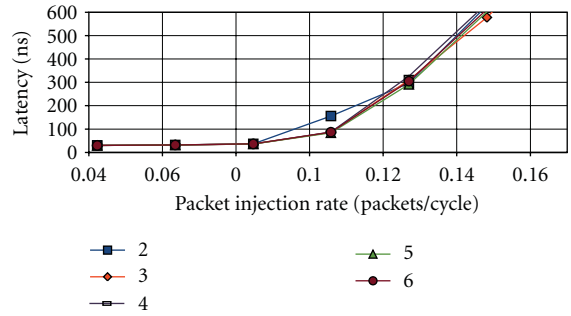


Figure 12: Latency as number of virtual channels is varied for *ami33*.

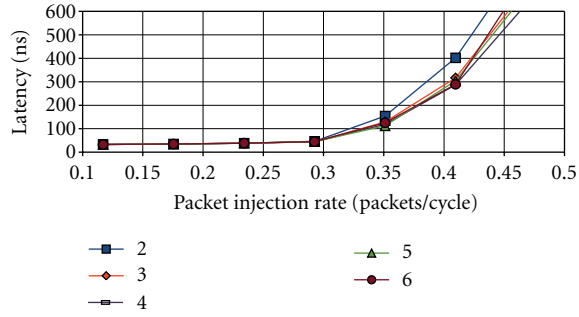


Figure 13: Latency as number of virtual channels is varied for *ami49*.

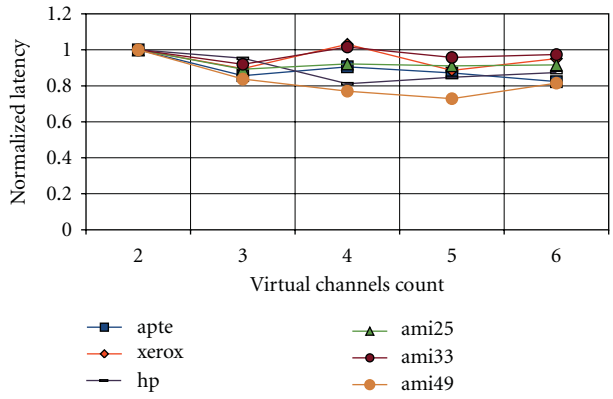


Figure 14: Normalized latency with variation of the number of virtual channels for the packet injection rate when the network saturation occurs.

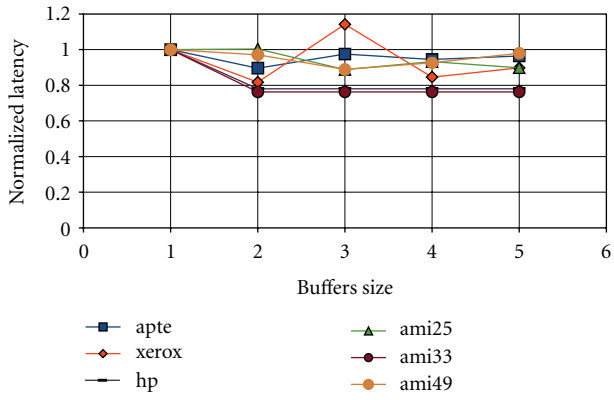


Figure 15: Normalized latency with variation of the buffers size for the packet injection rate when the network saturation occurs.

(with respect to the latency achieved using the default of 2 virtual channels) for the packet injection rate when the network saturation occurs. We find that the optimal number of virtual channels is different for different testcases.

These results are expected because the overall congestion in the network is intuitively reduced if the number of virtual channels multiplexed in the time-domain over a physical channel is increased. However, it is also evident that increasing the number of virtual channels more than necessary can have a negative impact on performance. This is explained as follows: as the network gets loaded with injected packets, the average amount of stalling due to arbitration inside routers (with increasingly more virtual channels) may increase and affect negatively the latency.

6.2.2. Varying the Buffers Size. In this experiment, instead of increasing the number of virtual channels, we increase the buffers size. Since we assume the area occupied by routers to be roughly 20% of the total cores area, we increase the area of each router to up to 5x by increasing both the input and output buffer sizes of each port (buffers occupy most of the area inside the router architecture).

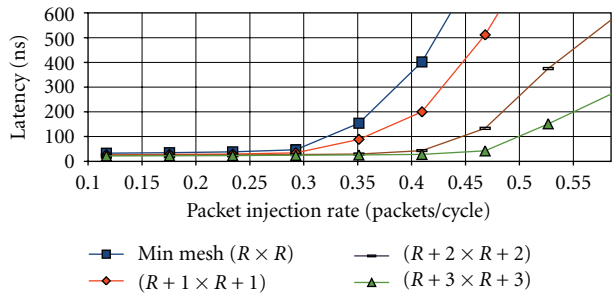


Figure 16: Latency as mesh size is varied for *ami49*.

Due to space limitations, we report (see Figure 15) only the normalized latency (with respect to the latency achieved using the minimum buffer size) for the packet injection rate when the network saturation occurs. We observe that the average flit latency improves with the increase of the buffer size. However, once a certain buffer size is reached (which is roughly the 3x data point except for *xerox*) further increasing the buffer size does not improve the latency. We note that in general, latency is improved more by increasing the number of virtual channels rather than increasing the buffers size.

6.2.3. Varying the Mesh Size. In this experiment, we investigate the impact of increasing the size of the regular mesh network ($R + k \times R + k$, $k = 1, 2, 3$) on the average flit latency. Increasing the number of routers in both dimensions x, y for the same testcase translates in shorter physical links between routers. As a result, the entire system can be clocked at higher frequencies, which significantly improves the saturation throughput. For example, the result of this experiment for testcase *ami49* is shown in Figure 16. The results of the rest of the testcases are similar; we do not include the rest of the plots here due to space limitations.

It has to be noted that this result is achieved under the assumption that the delay of the router pipeline is a clock period, which is given by the delay of the physical link. This assumption is made in [37], from where we adopted the NoC simulator, and can be validated by using speculation and lookahead as discussed in [8]. If, however, this assumption is removed, the router pipeline should incur a delay equal to two, three, or four clock cycles (to account for typical operations including routing computation, virtual channel allocation, switch allocation, and switch traversal), depending on the type of flit (head, body or tail) and the degree of speculation and lookahead [8]. On the other hand, if the router pipeline is assumed to incur a constant and fixed delay—irrespective of the physical link length—then the system clock frequency will be given by the maximum between the delay of the physical link and the delay of the router pipeline. In this case, the impact of increasing the mesh size will be less significant, and it could actually lead to an increase of the flit latency as studied in [36].

6.3. Comparison of the 2-Layer and 2D Architectures. In order to compare the proposed 2-layer architecture against

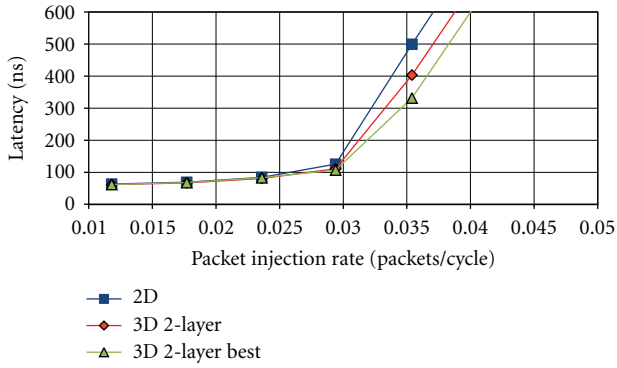


Figure 17: Latency comparison for 2D and 2-layer implementations of *apte*.

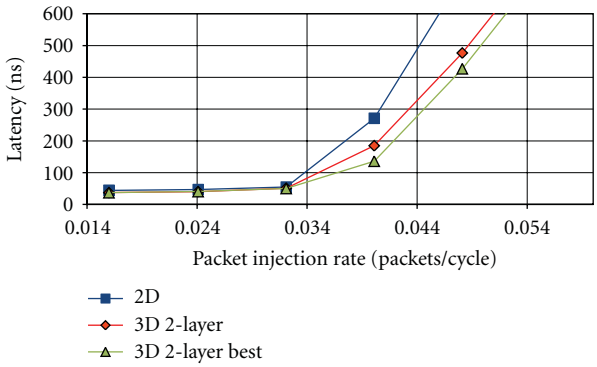


Figure 18: Latency comparison for 2D and 2-layer implementations of *xerox*.

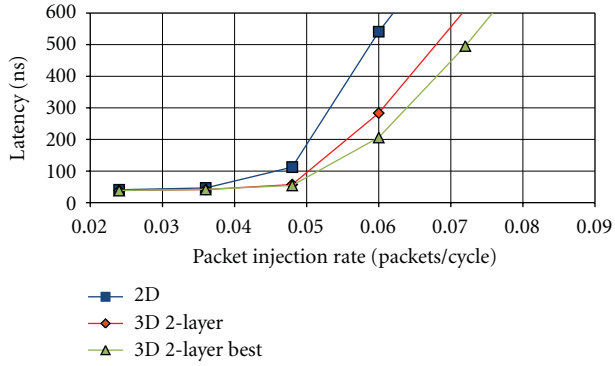


Figure 19: Latency comparison for 2D and 2-layer implementations of *hp*.

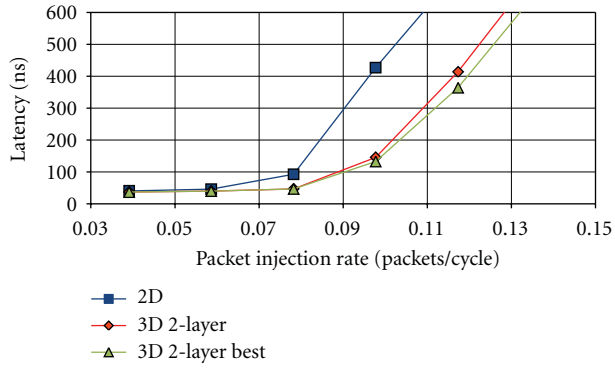


Figure 20: Latency comparison for 2D and 2-layer implementations of *ami25*.

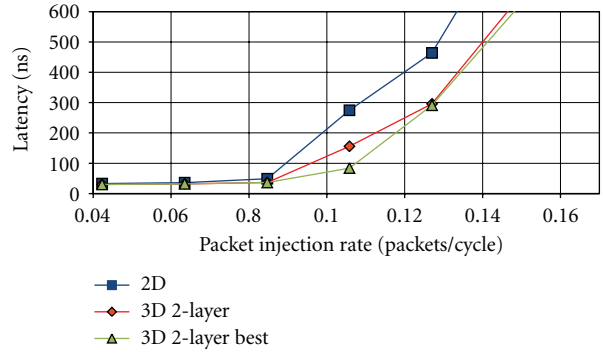


Figure 21: Latency comparison for 2D and 2-layer implementations of *ami33*.

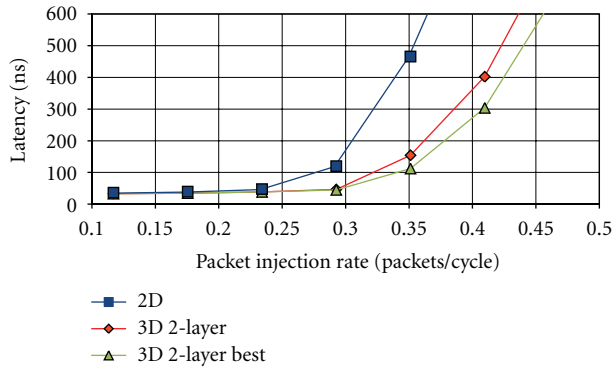


Figure 22: Latency comparison for 2D and 2-layer implementations of *ami49*.

a traditional approach we construct the 2D architecture by artificially expanding the IP/cores with 20% [12, 16] to account for the space occupied by routers and network interface (implemented within the cores boundaries on the same layer).

The simulation results are shown in Figures 17, 18, 19, 20, 21, and 22. We observe that the performance of the 2-layer architecture is better for each testcase. For the 2-layer architecture, the additional delays incurred due to the TSVs negatively impact the flit latency. However, the footprint area is smaller (as cores are smaller), and therefore the physical links are shorter, which leads to significantly smaller flit latencies. In addition, if we exploit the extra area on the top layer of the 2-layer architecture as described in the previous sections, then the latency can be further improved as illustrated by the *3D 2-layer best* data points from Figures 17–22.

We also note that the performance of the 2D architecture may be improved by designing custom NoCs similar to those studied in [28, 30]. However, the main goal of this paper is not to show that *regular* homogeneous 3D NoCs are better than *custom* 2D NoCs, which is unlikely, but to propose 3D

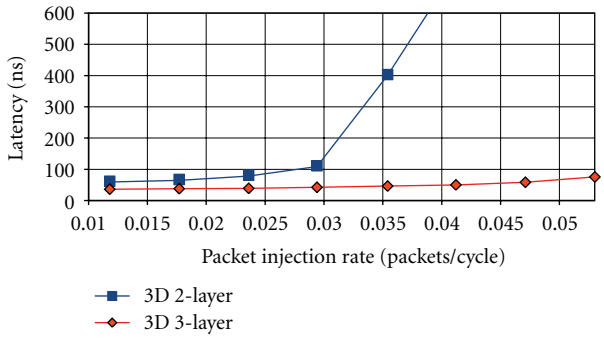


Figure 23: Latency for 2-layer and 3-layer implementations of *apte*.

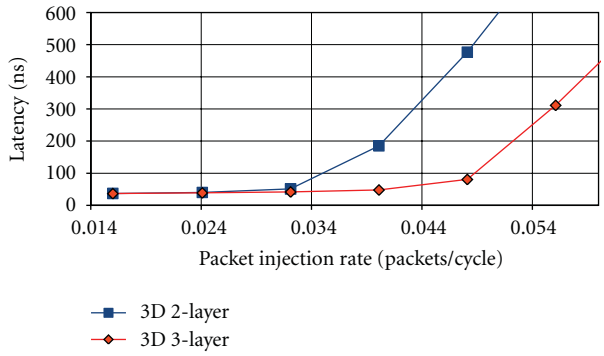


Figure 24: Latency for 2-layer and 3-layer implementations of *xerox*.

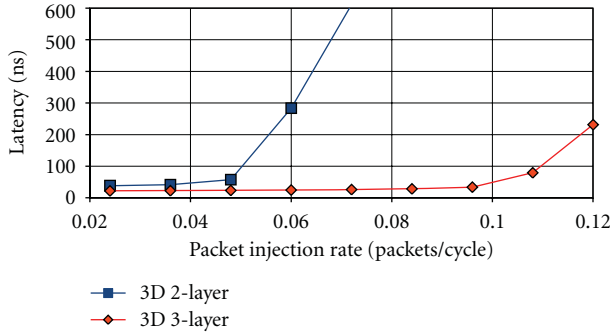


Figure 25: Latency for 2-layer and 3-layer implementations of *hp*.

architectures as alternatives to *regular* 2D NoCs and explore their performance when the number of virtual channels, buffers size, and mesh size is varied to use the extra space available on layer 2.

6.4. Comparison of the 2-Layer and 3-Layer Architectures. In the last part of our experiments, we compare the average flit latencies of the 2-layer and 3-layer architectures. The simulation results, using the default values for mesh size, number of virtual channels and buffers size are shown in Figures 23, 24, 25, 26, 27, and 28. As expected, because in the case of the 3-layer architecture the physical links are shorter, the clock frequency at which the system can work is higher. Hence, the average flit latency is improved significantly. Therefore, we conclude that the 3-layer architecture is better than the

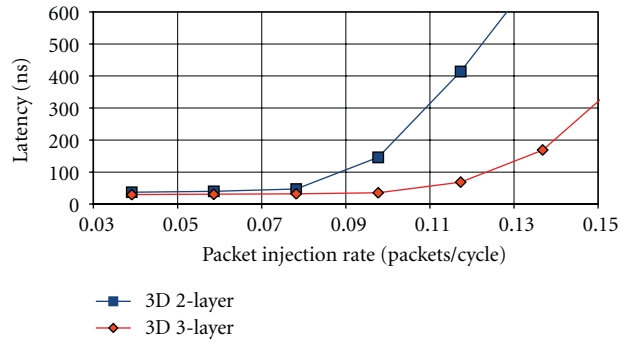


Figure 26: Latency for 2-layer and 3-layer implementations of *ami25*.

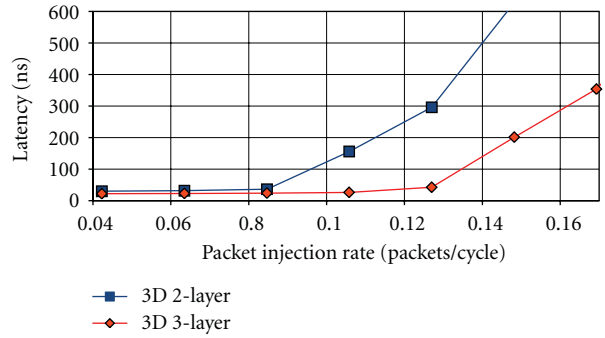


Figure 27: Latency for 2-layer and 3-layer implementations of *ami33*.

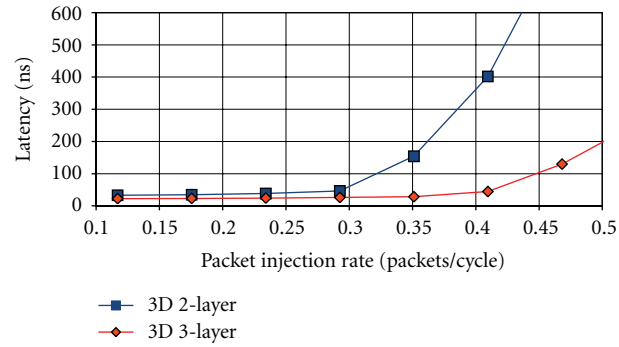


Figure 28: Latency for 2-layer and 3-layer implementations of *ami49*.

2-layer architecture. However, the design methodology for the 3-layer architecture requires additionally the partitioning step and the modification of the floorplanning algorithm. It may potentially require more complex thermal management too due to the increased integration density and the 3D fabrication technology will be more complex due to the alignment of three layers.

7. Conclusion and Future Work

In this paper, we proposed 3D 2-layer and 3-layer NoC architectures that utilize homogeneous networks on a separate layer and heterogeneous floorplans on different layers. A

design methodology that consists of floorplanning, routers assignment, and cycle-accurate NoC simulation was implemented and utilized to investigate the new architectures. Experimental results showed that increasing the number of virtual channels rather than the buffers size is more effective in improving the NoC performance. In addition, increasing the mesh size can significantly improve the NoC performance under the assumption that the clock frequency is given by the length of the physical links. Moreover, the 3-layer architecture can offer significantly better NoC performance compared to the 2-layer architecture.

As future work, we plan to address the problems of energy consumption and thermal profile optimization [53, 54] possibly in a unified fashion inside the floorplanning algorithm. The floorplanning step will be modified to consider the allocation of white space and TSVs planning under area constraints.

Acknowledgment

This paper was supported by the Electrical and Computer Engineering Department at North Dakota State University (NDSU).

References

[1] L. Xue, C. C. Liu, H.-S. Kim, S. K. Kim, and S. Tiwari, "Three-dimensional integration: technology, use, and issues for mixed-signal applications," *IEEE Transactions on Electron Devices*, vol. 50, no. 3, pp. 601–609, 2003.

[2] W. R. Davis, J. Wilson, S. Mick et al., "Demystifying 3D ICs: the pros and cons of going vertical," *IEEE Design and Test of Computers*, vol. 22, no. 6, pp. 498–510, 2005.

[3] P. Morrow, B. Black, M. J. Kobrinsky et al., "Design and fabrication of 3D microprocessors," in *Proceedings of Materials Research Society Symposium*, 2006.

[4] S. J. Koester, A. M. Young, R. R. Yu et al., "Wafer-level 3D integration technology," *IBM Journal of Research and Development*, vol. 52, no. 6, pp. 583–597, 2008.

[5] P. Guerrier and A. Grenier, "A generic architecture for on-chip packetswitched interconnections," in *Proceedings of ACM/IEEE Design Automation and Test in Europe Conference (DATE '00)*, pp. 250–256, 2000.

[6] A. Hemani, A. Jantsch, S. Kumar et al., "Network on chip: an architecture for billion transistor era," in *Proceedings of IEEE NorChip Conference*, November 2000.

[7] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 684–689, June 2001.

[8] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.

[9] G. de Micheli and L. Benini, *Networks on Chip*, Morgan Kaufmann, 2006.

[10] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, 2009.

[11] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, pp. 71–121, 2006.

[12] E. Salminen, A. Kulmala, and T. D. Hamalainen, "Survey of Network-on-Chip proposals," White Paper OCP-IP, 2008.

[13] L. P. Carloni, P. Pande, and Y. Xie, "Networks-on-chip in emerging interconnect paradigms: advantages and challenges," in *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NoCS '09)*, pp. 93–102, May 2009.

[14] V. F. Pavlidis and E. G. Friedman, "3-D topologies for networks-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 10, pp. 1081–1090, 2007.

[15] B. S. Feero and P. P. Pande, "Networks-on-chip in a three-dimensional environment: a performance evaluation," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 32–45, 2009.

[16] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and management of 3D chip multiprocessors using network-in-memory," in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA '06)*, pp. 130–141, June 2006.

[17] J. Kim, C. Nicopoulos, D. Park et al., "A novel dimensionally-decomposed router for on-chip communication in 3D architectures," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, pp. 138–149, June 2007.

[18] D. Park, S. Eachempati, R. Das et al., "MIRA: a multi-layered on-chip interconnect router architecture," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA '08)*, pp. 251–261, June 2008.

[19] Y. Xu, Y. Du, B. Zhao, X. Zhou, Y. Zhang, and J. Yang, "A low-radix and low-diameter 3D interconnection network design," in *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA '09)*, pp. 30–42, Raleigh, NC, USA, February 2009.

[20] S. Yan and B. Lin, "Design of application-specific 3D networks-on-chip architectures," in *Proceedings of the 26th IEEE International Conference on Computer Design (ICCD '08)*, pp. 142–149, October 2008.

[21] R. S. Ramanujam and B. Lin, "A layer-multiplexed 3D on-chip network architecture," *IEEE Embedded Systems Letters*, vol. 1, no. 2, pp. 50–55, 2009.

[22] A. Y. Weldezion, M. Grange, D. Pamunuwa et al., "Scalability of network-on-chip communication architecture for 3-D meshes," in *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NoCS '09)*, pp. 114–123, May 2009.

[23] Y. Qian, Z. Lu, and W. Dou, "From 2D to 3D NoCs: a case study on worst-case communication performance," in *Proceedings of ACM/IEEE International Conference on Computer Aided Design (ICCAD '09)*, pp. 555–562, November 2009.

[24] C. Mineo, R. Jenkal, S. Melamed, and W. Rhett Davis, "Inter-die signaling in three dimensional integrated circuits," in *Proceedings of IEEE Custom Integrated Circuits Conference (CICC '08)*, pp. 655–658, September 2008.

[25] S. Murali and G. De Micheli, "SUNMAP: a tool for automatic topology selection and generation for NoCs," in *Proceedings of the 41st Design Automation Conference (DAC '04)*, pp. 914–919, June 2004.

[26] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar, "A design methodology for application-specific networks-on-chip," *Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 263–280, 2006.

[27] S. Murali, L. Benini, and G. de Micheli, "Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees," in *Proceedings of ACM/IEEE Asia and South Pacific Design Automation Conference (ASPDAC '05)*, pp. 27–32, 2005.

[28] S. Murali, P. Meloni, F. Angiolini et al., "Designing application-specific networks on chips with floorplan information," in *Proceedings of International Conference on Computer-Aided Design (ICCAD '06)*, pp. 355–362, November 2006.

[29] S. Murali, C. Seiculescu, L. Benini, and G. De Micheli, "Synthesis of networks on chips for 3D systems on chips," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, pp. 242–247, Yokohama, Japan, January 2009.

[30] K. Srinivasan and K. S. Chatha, "A low complexity heuristic for design of custom network-on-chip architectures," in *Proceedings of Design, Automation and Test in Europe (DATE '06)*, pp. 130–135, March 2006.

[31] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear-programming-based techniques for synthesis of network-on-chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 4, pp. 407–420, 2006.

[32] K. S. Chatha, K. Srinivasan, and G. Konjevod, "Automated techniques for synthesis of application-specific network-on-chip architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1425–1438, 2008.

[33] C. Neeb and N. Wehn, "Designing efficient irregular networks for heterogeneous systems-on-chip," *Journal of Systems Architecture*, vol. 54, no. 3-4, pp. 384–396, 2008.

[34] T. Ahonen, D. A. Sigüenza-Tortosa, H. Bin, and J. Nurmi, "Topology optimization for application-specific networks-on-chip," in *Proceedings of International Workshop on System Level Interconnect Prediction (SLIP '04)*, pp. 53–60, Paris, France, February 2004.

[35] S.-Y. Lin, C.-H. Huang, C.-H. Chao, K.-H. Huang, and A.-Y. Wu, "Traffic-balanced routing algorithm for irregular mesh-based on-chip networks," *IEEE Transactions on Computers*, vol. 57, no. 9, pp. 1156–1168, 2008.

[36] V. de Paulo and C. Ababei, "A framework for 2.5D NoC exploration using homogeneous networks over heterogeneous floorplans," in *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 267–272, Cancun, Mexico, December 2009.

[37] L. Shang, L.-S. Peh, and N. K. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," in *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA '03)*, 2003.

[38] F. A. Samman, T. Hollstein, and M. Glesner, "Networks-on-chip based on dynamic wormhole packet identity mapping management," *VLSI Design*, vol. 2009, Article ID 941701, 2009.

[39] P. Zipf, G. Sassatelli, N. Utlu, N. Saint-Jean, P. Benoit, and M. Glesner, "A decentralised task mapping approach for homogeneous multiprocessor Network-On-Chips," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 453970, 14 pages, 2009.

[40] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[41] G. Karypis, hMetis, 2009, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download.

[42] Y. Chang, Y. Chang, G. Wu, and S. Wu, "B*-trees: a new representation for non-slicing floorplans," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 458–463, June 2000.

[43] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society of Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[44] L. Peh and W. J. Dally, "Delay model and speculative architecture for pipelined routers," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*, pp. 255–266, October 2001.

[45] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic," *VLSI Design*, vol. 2007, Article ID 68432, 2007.

[46] C. Ababei, VNOC3, 2009, http://venus.ece.ndsu.nodak.edu/~cris/software.html.

[47] S. R. Vangal, J. Howard, G. Ruhl et al., "An 80-Tile Sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, 2008.

[48] S. Bell, B. Edwards, J. Amann et al., "TILE64™ processor: a 64-core SoC with mesh interconnect," in *Proceedings of IEEE International Solid State Circuits Conference (ISSCC '08)*, pp. 88–81, February 2008.

[49] K. Kim, S. Lee, J.-Y. Kim et al., "A 125GOPS 583mW network-on-chip based parallel processor with bio-inspired visual-attention engine," in *Proceedings of IEEE International Solid State Circuits Conference (ISSCC '08)*, pp. 305–615, February 2008.

[50] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

[51] ITRS, 2007 Edition, Interconnects, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Interconnect.pdf.

[52] I. Loi, S. Mitra, T. H. Lee, S. Fujita, and L. Benini, "A low-overhead fault tolerance scheme for TSV-based 3D network on chip links," in *Proceedings of International Conference on Computer-Aided Design (ICCAD '08)*, pp. 599–602, November 2008.

[53] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, pp. 551–562, 2005.

[54] W. Hung, C. Addo-Ouaye, T. Theocharides, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, "Thermal-aware IP visualization and placement for networks-on-chip architecture," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '04)*, pp. 430–437, October 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 375245, 12 pages
doi:10.1155/2010/375245

Research Article

Exploiting Dual-Output Programmable Blocks to Balance Secure Dual-Rail Logics

Laurent Sauvage, Maxime Nassar, Sylvain Guilley, Florent Flament, Jean-Luc Danger, and Yves Mathieu

Département COMELEC, Institut TELECOM/TELECOM ParisTech, CNRS LTCI (UMR 5141), 46 Rue Barrault, 75 634 Paris Cedex 13, France

Correspondence should be addressed to Laurent Sauvage, laurent.sauvage@telecom-paristech.fr and Sylvain Guilley, sylvain.guilley@telecom-paristech.fr

Received 2 March 2010; Accepted 5 October 2010

Academic Editor: Lionel Torres

FPGA design of side-channel analysis countermeasures using unmasked dual-rail with precharge logic appears to be a great challenge. Indeed, the robustness of such a solution relies on careful differential placement and routing whereas both FPGA layout and FPGA EDA tools are not developed for such purposes. However, assessing the security level which can be achieved with them is an important issue, as it is directly related to the suitability to use commercial FPGA instead of proprietary custom FPGA for this kind of protection. In this article, we experimentally gave evidence that differential placement and routing of an FPGA implementation can be done with a granularity fine enough to improve the security gain. However, so far, this gain turned out to be lower for FPGAs than for ASICs. The solutions demonstrated in this article exploit the dual-output of modern FPGAs to achieve a better balance of dual-rail interconnections. However, we expect that an in-depth analysis of routing resources power consumption could still help reduce the interconnect differential leakage.

1. Introduction

During the last decade, a considerable number of countermeasures have been proposed to protect cryptographic devices against Side Channel Analysis (SCA). They customarily fall in two categories. The first one, *masking* [1, 2], is very popular in the smart card community as it can be implemented at the algorithm level. Intermediate data processed by a cryptoprocessor are concealed by a value called mask, randomly chosen, which in turn makes power consumption random. In the second category, *hiding*, intermediate values remain the same as for unprotected implementations, but power consumption is made as constant as possible. According to the state-of-the-art attacks, masked implementation on a Field Programmable Gates Array (FPGA) could be broken with Higher-Order Differential Power Analysis (HODPA) using 12,000 power consumption traces [3] whereas 1,500,000 measurements are not sufficient to disclose the entire secret key of an Application Specific Integrated Circuit (ASIC) cryptoprocessor protected by Wave Dynamic Differential Logic (WDDL) [4], the most popular hiding countermeasure developed by Tiri and Verbauwhede. Proposals for merging both techniques with a view to compensate for weakness of masking against HODPA and for backend operations hardness of hiding have been reported in [5, 6], but this approach unfortunately remains vulnerable when the masking relies on one single bit of entropy [7].

Unmasked dual-rail with Precharge Logics (DPLs) in general seem thus to be a sound solution. As WDDL is based on a standard cell flow, it is the most suited for FPGA implementation. Guidelines for synthesis can be found in [8, 9]. We notice incidentally that those articles target $4 \rightarrow 1$ LuT-based FPGA technologies and thus do not take full advantage of the advanced features of modern FPGAs, such as ALM (Adaptative Logic Modules) configurable blocks and

dual-output logic blocks. DPL robustness against SCA relies on perfectly matching differential routing, which appears to be incredibly hard to achieve for large FPGA designs because of the following.

(i) The only way to evaluate the imbalance between two differential paths is to compute the difference of path delays. But as delays provided by development tools are maximum values, not typical, unbalance evaluation is coarse.

(ii) Routing resources are limited and have thus to be properly shared between each component of the design. This may affect placement, which could no longer simply consist in placing differential components side by side.

(iii) Commercial off-the-shelf (COTS) FPGA layout has been designed to provide flexibility, not differential capability. As a consequence, the performance of DPL is expected to be less efficient than when embedded in ASIC, for which differential components can be placed as close as possible, decreasing the impact of intradie process variability.

To overcome these problems, Yu and Schaumont suggest the Double WDDL (DWDDL) [10] design strategy: from a first direct WDDL module, a complementary clone with identical routing is obtained by duplication, relocation, and logic modification. This way, leakages due to imbalances of each module are expected to compensate one with each other. Unfortunately, some registers of DWDDL never go to precharge value and introduce leakages in the Hamming Distance (HD) model. McEvoy et al. discovered this flaw and propose Isolated WDDL (IWDDL) [11] to solve it. To decrease the fourfold area increase of DWDDL and IWDDL, Baddam and Zwolinski published methods suitable for both ASIC and FPGA. Path Switching [12] balances true and false paths with long high capacitive lines by random swaps. Attractive from a theoretical standpoint, in practice, Path Switching is realized by active elements, which consume power, and thus might diminish robustness against SCA. Divided Backend Duplication [13] consists in splitting dual networks by replacing inverters by exclusive-or (XOR) gates: in precharge phase, the XOR gates are configured as an identity function ($a \mapsto 0 \oplus a = a$) thus propagating the precharge wave, whereas in evaluation phase, they become functional ($a \mapsto 1 \oplus a = \overline{a}$). Unfortunately, this logic has never proved to be glitch-free, nor of constant activity (because of unwanted spurious glitches). Some logic-level upgrades of WDDL have also been proposed, amongst which iMDPL [14], DRSL [6], STTL [15, 16], SecLib [17], WDDL w/o early evaluation [18], and BCDL [19]. All those styles can be mapped onto an FPGA fabric and are thus concerned with dual-rail balancing.

In this article, we do not search to add logic to balance dual networks but rather finely direct the place-and-route (PAR) tool to avoid area increase. The research is led with a view to assess the security level which can be achieved using commercial FPGAs and conclude on their suitability for balanced DPL. Impact of constrained placement of SBoxes has already been described in [20]. Here, we go several steps further: routing is also constrained and we study and experimentally evaluate a complete cryptoprocessor. This latter is presented is Section 2, along with a thorough characterization of the PAR at design-level. Then, we explain how to best take advantage of dual-output programmable logic blocks. The experimental results we obtain against SCA are provided in Section 3. Finally, conclusions and perspectives are discussed in Section 4.

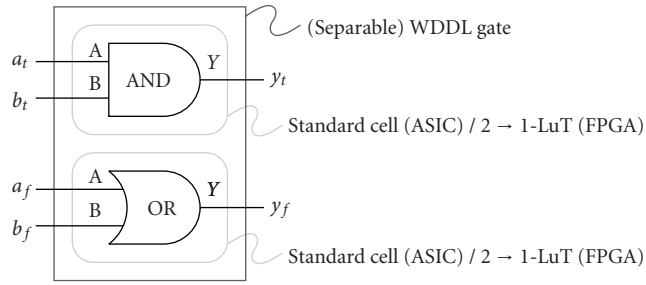


Figure 1: WDDL AND gate, suitable for both ASICs and FPGAs.

2. Fitting WDDL 3DES in Stratix II

2.1. WDDL Roots. In DPL, each bit is represented by a couple of signals, and calculations alternate between precharge phase (PRE) and evaluation phase (EVA). Figure 1 is an example of a WDDL AND gate. In PRE, all of the inputs on the left are forced to "0", which in turn forces the true output s_t and the false output s_f to "0". Then, in EVA, after inputs toggle, only s_t or s_f will be set to "1". This way, whatever the temporal transition, EVA to PRE or PRE to EVA, and whatever the processed data, only one of the differential outputs commutes, yielding a constant transition count. This protocol is illustrated in Figure 2. But this does not suffice to ensure a constant power consumption: the capacitive load of the True and False networks should be the same. Special care should thus be taken for back-end operations, which is the main purpose of this article.

To avoid glitches, a security flaw of WDDL as they are data dependant, synthesis should use only positive functions. For example, logical exclusive OR (XOR) operator cannot be directly used and will be replaced by $a \oplus b = a_t \cdot b_f + a_f \cdot b_t$. This induces cross connections between True and False paths, which increases PAR difficulty.

2.2. WDDL 3DES Cryptoprocessor. The implementation evaluated in this article conforms to the simple and triple Data Encryption Standard (3DES) [21], with all of the specified modes of operations. Although DES has been replaced by the Advanced Encryption Standard (AES) since year 2001, the American National Institute of Standards and Technology (NIST) considers 3DES to be appropriate through year 2030, and the electronic payments industry still uses it for its compactness when implemented in hardware, a quarter of AESs.

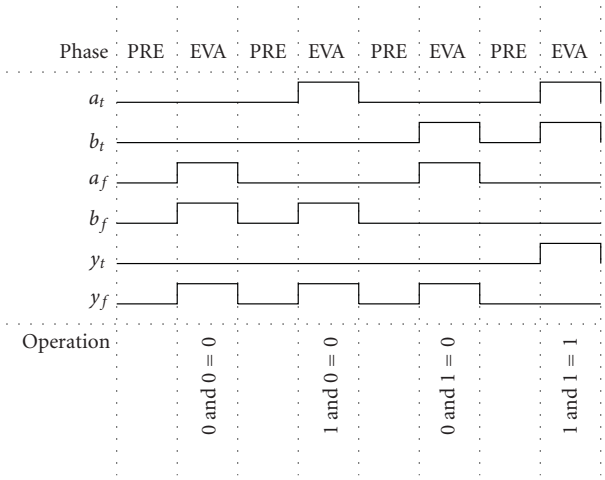


Figure 2: Secure return-to-null protocol for a constant activity of the WDDL AND gate depicted in Figure 1.

The architecture of Figure 3 corresponds to the part of our WDDL 3DES cryptoprocessor which is in charge of processing the 32-bit block R, as defined in page 10 and depicted by page 9 of [21]. This implementation has a straightforward parallel execution, scheduled at one round per clock cycle. The upper and lower paths correspond to the True and False dual networks, respectively. At the beginning of an encryption, the right half of the value resulting of an initial permutation (IP) of the message is loaded into the R master register (R_M), while the R slave register (R_S) still holds the zero precharge value. The latter is applied to the combinatorial logic of the DES datapath, comprised of the following:

(i) the *expansion permutation* (E) function,

(ii) a bit-by-bit addition modulo 2 between the output of E and the round key K_n, referred in the following as XOR_K,

(iii) *Substitution Boxes* (SBoxes) nonlinear functions,

(iv) the *permutation* (P) function,

(v) a bit-by-bit addition modulo 2 between the output of P and the left part of the LR register (L), referred in the following as XOR_L,

which in turn outputs zero. Due to the feedback, at the next rising edge of the clock R_M will load zero while R_S will sample the right half of IP, and the combinatorial logic computes the new intermediate value. This alternates until the end of the encryption.

2.3. Place-and-Route Strategies. Some previous articles [8, 9] already described the mapping of dual-rail logics into FPGA. However, all of them target 4 → 1 look-up-table FPGAs. This choice is interesting from the synthesis point of view, since the mapping is very close to that of an ASIC; as discussed in [4], a simple DPL-compliant design flow is based on the netlist duplication. However, in the FPGA

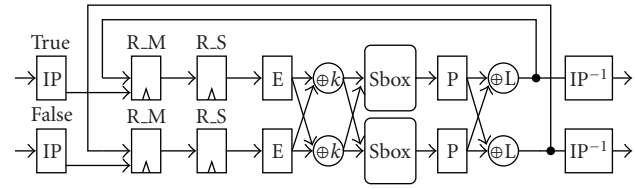


Figure 3: Part of our WDDL 3DES cryptoprocessor processing the 32-bit block R.

context, keeping two dual instances side-by-side is not trivial; for instance, [22] investigates on the various constraints (for Xilinx & Altera EDA tools) that can be defined at the user-level to force a pairwise placement. A typical mono-output LuT FPGA is the Stratix; Figure 4 illustrates its schematic. Experiments on this target reveal how hard it is to guarantee equal routing for differential signals. For instance, in Figure 5, a complete DES in WDDL is shown. The figure underlines the fact that one net with an extremely high fanout (one bit of the "LR" state addressing the S-Boxes) is doomed to have a different routing, even if the S-Boxes are implemented according to a balanced technique [23]. The idea is that the different location of the "true" and "false" instances will induce a difference in at least one routing path.

Therefore, a native packing of two dual instances into the same reconfigurable resource would be welcome. We choose to assess the robustness of our WDDL 3DES cryptoprocessor when programmed in an Altera Stratix II 90-nm FPGA. Its architecture relies on Adaptive Logic Modules (ALMs), basic building blocks of logic, whose high-level block diagram is shown in Figure 6. On the left, eight inputs drive a combinatorial logic block programmable as either one 6-bit Look-up-Table (6-LuT) or two Adaptive LuTs (ALuTs), both having their own register, named, respectively, *reg0* and *reg1*, on the right.

We have synthesized our implementation using 4-LuT, with the aim to test two PAR strategies. The first one, called "Vertical" strategy, places True and False networks as close as possible by assembling each dual component (R_M, R_S, XOR_K, etc.) in the same ALM (see Figure 7(a)). With the second one, the "Horizontal" strategy, routing resources are identical for the True and False networks: R_M, R_S and XOR_K are packed into the same ALM, and the dual part is placed in the adjacent ALM (see Figure 7(b)).

The floorplan of the Vertical strategy is shown in the layout of Figure 8. The four ALMs on the top left correspond to four bits of R_M. Two dual wires output from each of them, and they all go to R_S, in the middle, then to XOR_K, on the right, and finally reach the SBox and XOR_L (both outside of the figure). The wires on the middle top realize the expansion (E) function towards the SBox number $i+1$, while those on the middle bottom come from the SBox number $i-1$. As explained in Section 2.1, synthesis with positive functions may induce cross connections. Those of XOR_K are clearly visible on the top right of the figure. This schematic is reproduced for each SBox, thus eight times. Post PAR timing annotations give a first idea of the balance between

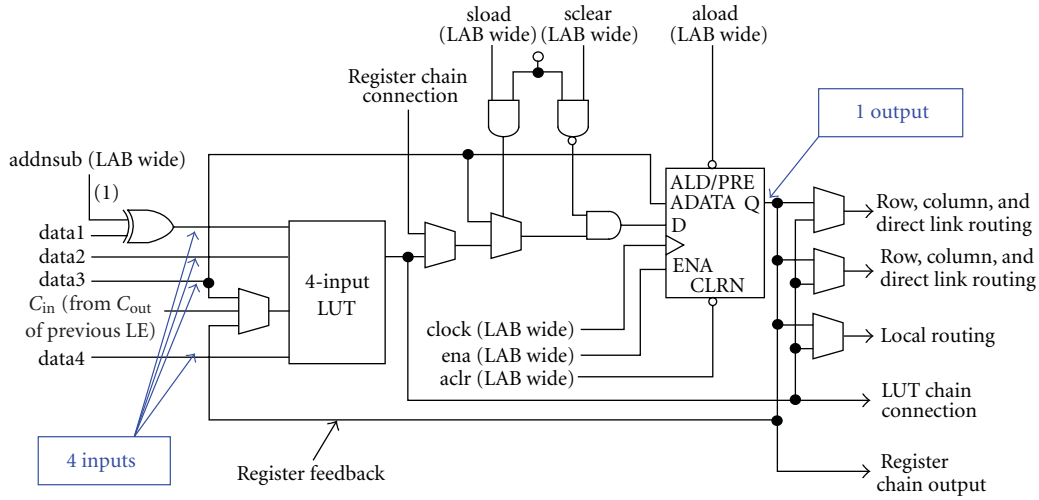


Figure 4: High-Level block diagram of a LuT, extracted from "Stratix Device Family Data Sheet, volume 1" [24].

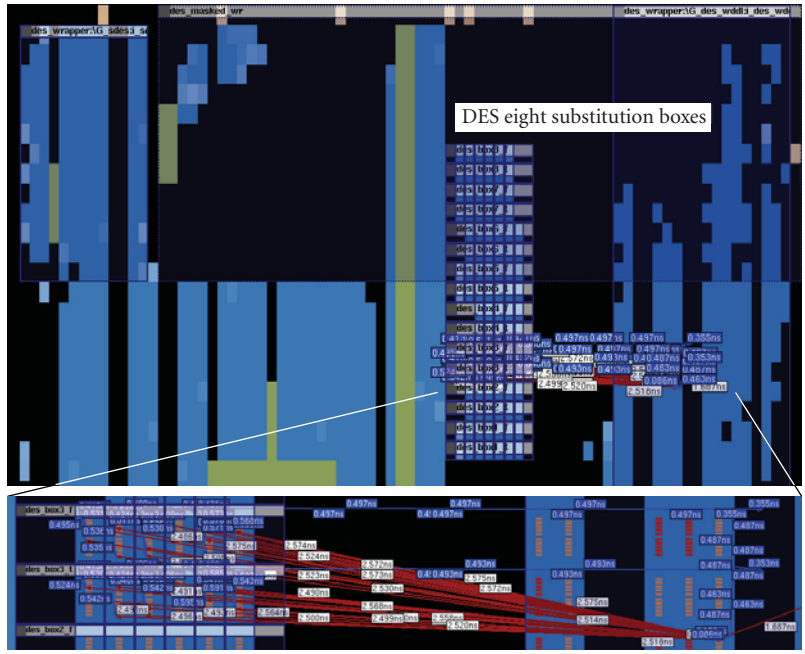


Figure 5: Floorplan of a full DES in WDDL (top) and slightly unbalanced fanout of one register of the "LR" state (bottom).

dual wires: for example, the bit of R_M on the top has an imbalance of $289 - 290 = -1$ ps.

Figure 9 illustrates the floorplan for the Horizontal strategy. The upper and lower ALMs contain the true and false networks, respectively. The *reg1* register from Figure 6 is assigned to the master register while the slave register is implemented in *reg0* and XOR_K by the combinatorial logic block, on the left. The gain in balance is optimum as timing annotations have exactly the same value.

2.4. Altera Place-and-Route Constraints. Under Altera's Quartus II design Software, placement is constrained via the LogicLock (LL) regions feature. An LL is a rectangular-shaped region defined by the following:

(1) a *state*: locked at an origin location or floating;

(2) a *size*: fixed user-defined or automatically determined by Quartus;

(3) a *reserved* property: defining whether entities not assigned to this LL region can use its remaining resources.

These properties can be set in a TCL script, according to the syntax shown in Figure 10.

Constraining the routing follows a different procedure. After fitting, a back-annotation of the routing proposed by

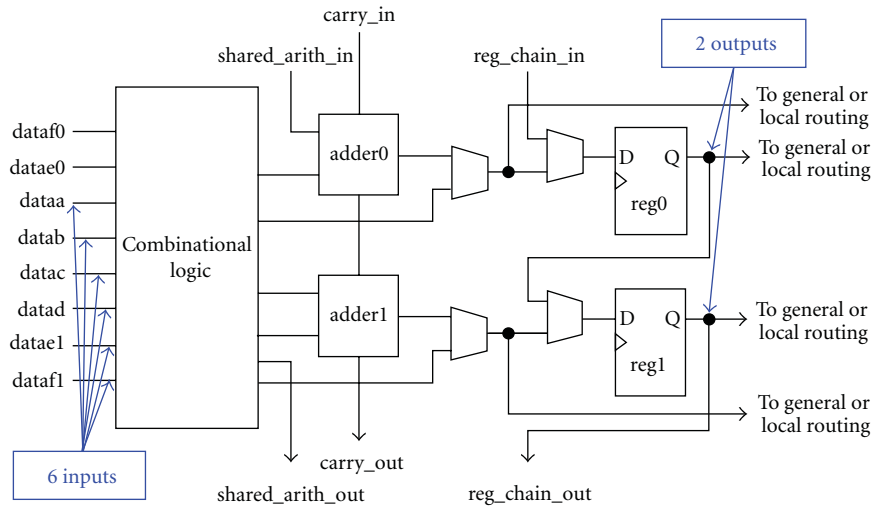


Figure 6: High-Level block diagram of an ALM, extracted from "Stratix II Device Family Data Sheet, volume 1" [24].

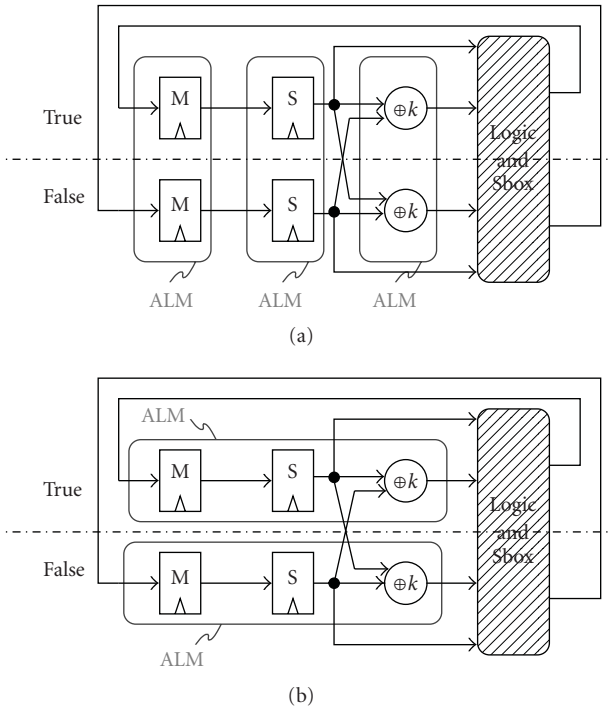


Figure 7: Vertical (a) and Horizontal (b) PAR strategies.

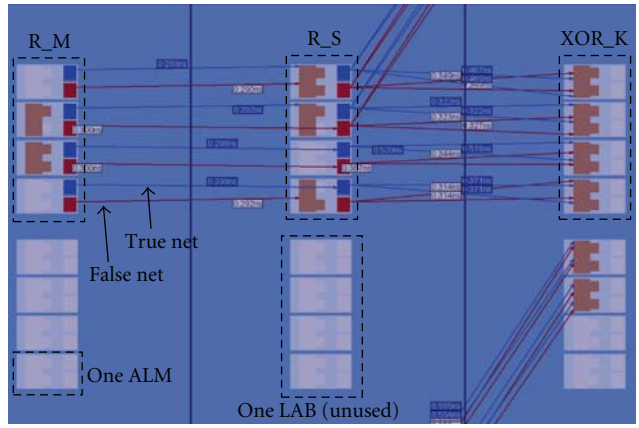


Figure 8: Vertical strategy.

Quartus should be requested. This generates a Routing Constraints File (RCF), whose format is given in Figure 11.

The signal *Input1* of this example comes from the output of the *LE_BUFFER* resource (corresponding to the cluster located at $X = 1$, $Y = 1$, S(ub-location) = 1, I(ndex) = 0) (device-dependant coordinates, described in the "QSF Assignment Descriptions Document"). It then goes through a vertical wire of length 4 (C4) and the local interconnect of the cluster located just above at $X = 1$ and $Y = 2$. The final destination is *DATAC*, an input port on the *InputReg1* block.

To customize the routing, the RCF has to be modified by the user. Afterwards the fitter has to be rerun (with back-annotation enabled to indicate to the Quartus Software that an RCF file exists and should be sourced). Finally, a new back-annotation has to be written out again, in order to verify that the router has met the user-constrained routing.

2.5. Experimental Characterization. Differential traces obtained by DPA can be helpful for a designer to pinpoint countermeasure weaknesses. Indeed, for the right key and monobit analyses, the amplitude of the correlation peak directly gives an experimental evaluation of the power consumption imbalance between true and false networks of a single bit. Many analyses can then be performed. In the following, we present two of them, focusing on the SBox1.

The trace in Figure 11, at the top, corresponds to the power consumption of the FPGA during the beginning of an encryption. At Round 0 evaluation phase, the IP of the message is loaded into LR_S (see Table 1 and Figure 3) after the rising edge of the master clock.

Table 1: Sequence at the beginning of the encryption.

DES Round	Initial State	0 (IP)		1	
WDDL phase	Precharge (PRE)	Precharge (PRE)	Evaluation (EVA)	Precharge (PRE)	Evaluation (EVA)
LR Master	0	IP	0	LR1	0
LR Slave	0	0	IP	0	LR1

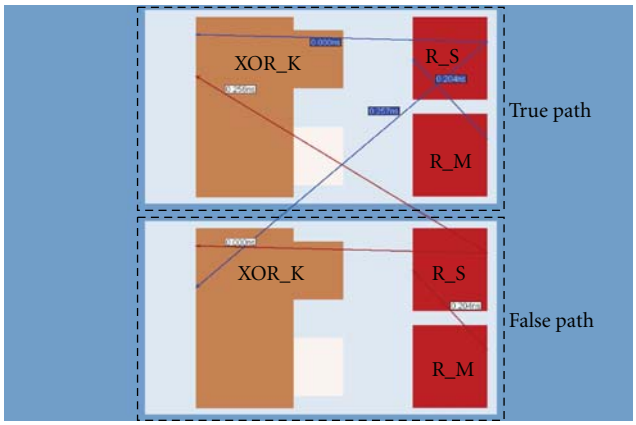


Figure 9: Horizontal strategy.

A few nanoseconds later, the result of the computation is available at the SBoxes output. The differential trace in the middle, obtained by targeting the bit 1 of the SBox1, indeed confirms this. Hence, one correlation peak clearly appears, slightly shifted on the right according to the rising edge of the consumption trace at $t = 418$ ns. Its amplitude, about -35 nV, is negative, suggesting that the false network has a power consumption higher than that of the true network.

Finally, on the rising edge Round 1 precharge phase, the value at the output of XOR_K is sampled by LR_M. Once again, a correlation peak emerges in the differential trace in the bottom, generated when targeting bit 16 of LR_M (after the P permutation, the bit 1 of SBox1 becomes bit 16 of LR). This time, the amplitude of the peaks is close to -60 nV. As DPA looks for the maximum amplitude, attacks on the LR register are the most powerful in our case.

This phenomenon can either be observed with single rail logic: indeed, registers consume more power than combinatorial logic.

Here, the explanation is that in combinatorial logic, commutation dates are data-dependant. Thus, the power consumption of each encryption vanishes in the differential trace because of a noncoherent averaging due to the intrinsic DPA processing whereas the power consumption of the register is well synchronized.

To gather it, we have to try to use a 4th-order integration [25], unsuccessfully.

To close these analyses, we have reported in Figure 10 the amplitude of the SBox1 output bits correlation peaks when targeting LR. Bit 2 has the highest amplitude, which justifies that it delivers the best performance for the attack (see Table 3). The imbalance is positive for the first and last bits, negative for the others. This experimentally confirms that attacking four bits at once as with unprotected module is less efficient, because bit imbalances counterbalance each other. An improved attack against DPL may be summing up the absolute value of each correlation peak.

Table 2: Static evaluation of timing imbalance, in ps.

PAR Strategy	None		Vertical		Horizontal	
Element	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
LR Master Register	251	322	24	23	3	3
LR Slave Register	566	327	137	88	25	28
XOR_K function	501	298	272	203	290	227
SBox1	202	174	157	119	157	119
SBox2	169	150	169	119	169	119
SBox3	165	155	169	112	169	112
SBox4	146	120	175	123	175	123
SBox5	131	114	167	128	167	128
SBox6	149	154	172	131	172	131
SBox7	170	152	160	118	160	118
SBox8	156	123	173	125	173	125

2.6. Static Evaluation of the Place-and-Route. In terms of timing differences, the study of the differential PAR quality can be achieved with an analysis of the Standard Delay Format (SDF) file generated by the Quartus II Compiler. Such a file gives interconnection and propagation delays of each instance in the FPGA. Analysis of registers and XOR function timing delays is trivial as less than three couples of dual wires per bit have to be considered. Determining the imbalance in the SBoxes is more difficult; indeed, all delay differences along the datapath have to be summed up.

Mean and standard deviation statistics on timing differences between the true and false network are given in Table 2 in picoseconds. The first column corresponds to the imbalance without specific PAR constraints. It serves as a reference to show the improvement generated by the two PAR strategies. On average, timing imbalances of SBoxes are not much affected by PAR strategies while those of the XOR_K are halved. The Horizontal strategy has a great impact on register imbalance, diminishing them to 3 ps. If the robustness is strongly correlated to the timing differences, this strategy should be the most robust. The next section on experimental results of the robustness evaluation shows that the opposite holds true.

```
// Definition of a LogicLock region assigned to <entity>,
// with STATE = locked at (1;1), SIZE = 10 LAB × 20 LAB,
// and free resources available for others entities.
set_global_assignment -name LL_MEMBER_OF <entity>
set_global_assignment -name LL_STATE LOCKED
set_global_assignment -name LL_ORIGIN LAB_X1_Y1
set_global_assignment -name LL_AUTO_SIZE OFF
set_global_assignment -name LL_WIDTH 10
set_global_assignment -name LL_HEIGHT 20
set_global_assignment -name LL_RESERVED OFF
```

Figure 10: Example of TCL script setting up a LogicLock region.

```
signal_name = Input1  {
LE_BUFFER:X1Y1S1I0;
C4:X1Y1S0I25;
LOCAL_INTERCONNECT:X1Y2S0I15;
dest = ( InputReg1, DATAC );
}
```

Figure 11: Example of an RCF file reporting the routing resources usage.

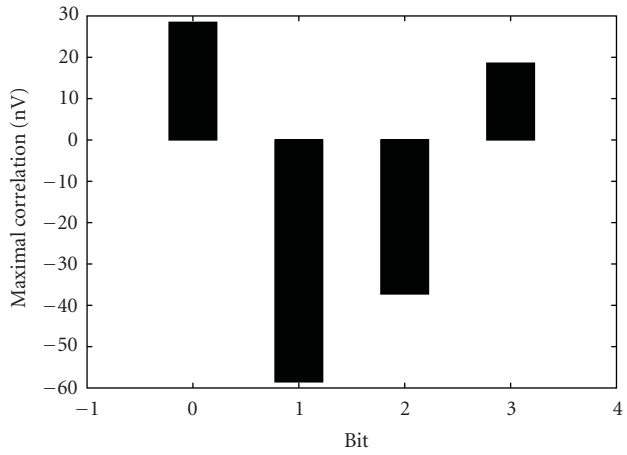


Figure 12: Histogram of SBox1 output bits power imbalance.

3. Security Evaluation

3.1. Background Material. Security evaluation is extremely influenced by various parameters: setups, acquisition conditions, target algorithms, target board, attack models, and so forth. Thus, it is hard to compare SCA countermeasures and attacks of different laboratories on a fair ground. To contribute to the elaboration of common platforms, guaranteeing experiments reproducibility for research institutions, the Tohoku University, and the Japanese Research Center for Information Security (RCIS) has developed in 2007 "Side-channel Attack Standard Evaluation Boards" (SASEBOs) [26]. Dedicated to the evaluation of SCA countermeasures, these electronic boards are distributed free of charge to academic laboratories leading innovative researches in the field of embedded security. Thereby, everyone can reproduce, analyse, and criticize results of their peers.

There are four versions, depending on the target chip. All experiments in this paper have been realized using the SASEBO-B, which incorporates two Altera [24] Stratix II FPGA: one *EP2S30F672C5N* supposed to embed all control modules, and one *EP2S15F484C5N* for the cryptographic modules. Having two FPGAs enhances the accuracy of the measurements as it uncouples the power consumption of the cryptographic parts from the power consumption of the others. Another measurement improvement feature is the possibility of using separate power supplies for the core and the input and output pads of the FPGA. Measurements have been done using the original 1 Ω spying shunt resistor in the positive rail of the cryptographic FPGA core power supply. The acquisition board is depicted in Figure 12.

To improve experiment reproducibility by peers, we have used "Eve (Eavesdropper) SoC", a System on Chip (SoC) providing a flexible way to easily and rapidly design real-life cryptographic applications: when a stand-alone hardware module protected by an SCA countermeasure is fully functional, it can simply be bound to EveSoc as a plug-and-play custom coprocessor, so as to constitute a complete cryptographic device. VHDL code files of EveSoc along with its documentation and miscellaneous tools including scripts for SASEBOs are freely downloadable from its *SourceForge* development website [27].

Power consumption measurements, referred in the following as *traces*, have been collected using a *1132A* differential probe and a *54855 Infiniium* oscilloscope from Agilent Technologies. The final setup has a 6 GHz bandwidth and a 40 GSa/s maximal sample rate.

3.2. Security Gain. As explained in the above section, various parameters affect the security evaluation, and it is always difficult to determine whether the hardness of an attack is caused by a good countermeasure or by a weak adversary. In this context, a framework has been proposed in [28] which suggests to start with an information theoretic analysis to assess the maximal amount of information which could be extracted. This corresponds in practice to the worst case attack, which is difficult to devise in the case of hardware implementations. Then, various distinguishers have to be used to see how a given adversary can take advantage of the

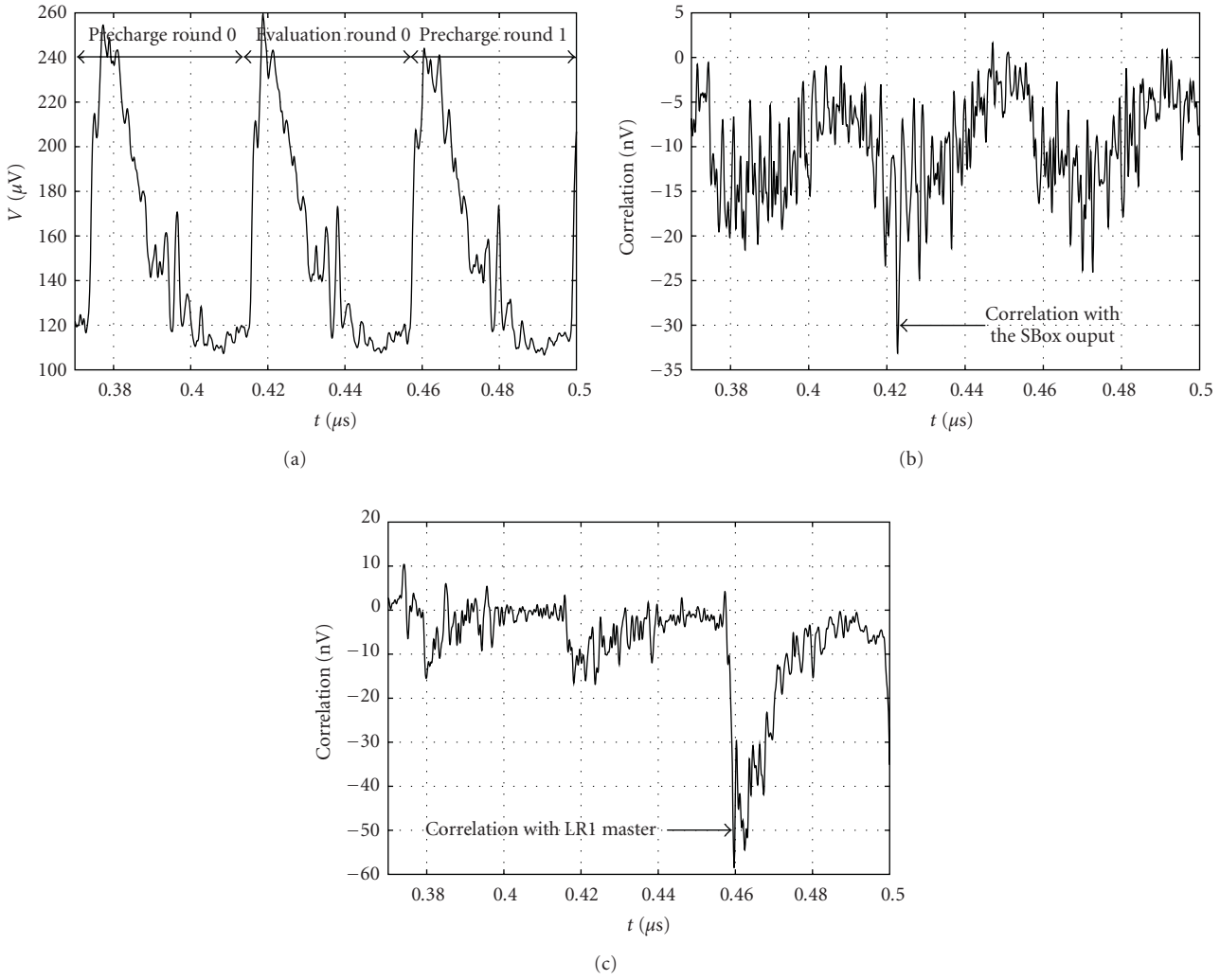


Figure 13: From (a)–(c) single trace and differential traces when targeting the SBox1 and the LR registers.

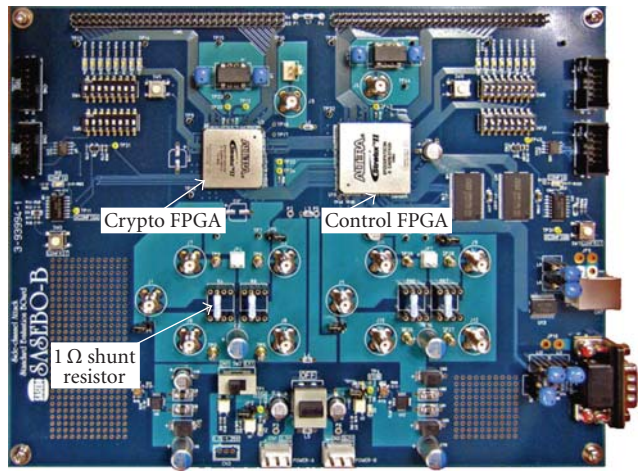


Figure 14: SASEBO-B configuration for the DPA attacks.

Table 3: Statistics for Bits of R_S Register.

(a) WDDL 3DES Module without PAR Constraints

SBox	1				2				3				4			
Bit of R	9	17	23	31	13	28	2	18	24	16	30	6	26	20	10	1
Timing imbalance, in ps	266	917	796	308	331	885	265	633	560	1,073	286	840	599	611	851	**253**
Security Gain	3	1	5	1	1	6	3	1	2	11	10	6	2	4	2	1
Minimal Security Gain	1				1				2				1			
SBox	5				6				7				8			
Bit of R	8	14	25	3	4	29	11	19	32	12	22	7	5	27	15	21
Timing imbalance, in ps	290	686	944	1,014	1,574	261	700	666	306	340	865	262	322	1,032	348	621
Security Gain	1	2	5	7	4	2	8	1	3	**22**	2	4	1	1	3	2
Minimal Security Gain	1				1				2				1			

(b) WDDL 3DES Module with Vertical Strategy

SBox	1				2				3				4			
Bit of R	9	17	23	31	13	28	2	18	24	16	30	6	26	20	10	1
Timing imbalance, in ps	38	64	23	23	53	30	5	1	29	32	**0**	1	1	59	1	17
Security Gain	50	10	9	5	1	8	10	4	12	20	9	31	6	21	23	21
Minimal Security Gain	5				1				9				6			
SBox	5				6				7				8			
Bit of R	8	14	25	3	4	29	11	19	32	12	22	7	5	27	15	21
Timing imbalance, in ps	25	1	29	1	123	30	1	1	51	9	1	**0**	42	1	59	42
Security Gain	2	19	13	—	214	5	15	—	19	352	11	31	36	1	8	31
Minimal Security Gain	2				5				11				1			

(c) WDDL 3DES Module with Horizontal Strategy

SBox	1				2				3				4			
Bit of R	9	17	23	31	13	28	2	18	24	16	30	6	26	20	10	1
Timing imbalance, in ps	325	125	125	152	182	176	**14**	29	156	146	65	41	38	221	42	355
Security Gain	9	8	7	2	3	10	4	1	18	15	25	11	2	5	20	5
Minimal Security Gain	2				1				15				2			
SBox	5				6				7				8			
Bit of R	8	14	25	3	4	29	11	19	32	12	22	7	5	27	15	21
Timing imbalance, in ps	26	39	204	136	168	139	147	247	208	166	14	150	261	50	83	182
Security Gain	2	4	2	15	110	8	11	15	23	**364**	18	23	32	4	9	14
Minimal Security Gain	2				8				18				4			

information leakage. However, with hardware implementation such as the one studied in this article, the construction of templates and the information theoretic analysis is very intensive and may not lead to better results than directly performing an attack. Thus, for a first study, we prefer to limit the security analysis and focus on few meaningful attacks.

The strength of these attacks, and thus the robustness against SCA of an implementation which is attacked, was first quantified by evaluating how many Measurements to Disclose (MTD) the secret key are needed. But this number has appeared to be dependent on the values, on the order of the messages, and on the secret key. Therefore, a sound approach is to perform a lot of attacks with different keys, for instance, one hundred, and then to consider the MTD for a given Global Success Rate (GSR), saying 90% or 95%. However, this absolute MTD remains not significant for a firm conclusion. Indeed, what could we think of a countermeasure which needs 1,500,000 traces to be broken, whereas its unprotected version resists analyses up to 1,000,000 measurements? Thus, in the following, we will quantify the security robustness by computing the *security gain* (SG), that is, the ratio between the MTD of a protected module and the MTD of an unprotected module:

$$\mathrm{SG} = \frac{\text{MTD-GSR90\%}_{\text{protected}}}{\text{MTD-GSR90\%}_{\text{unprotected}}}. \quad (1)$$

3.3. Experimental Results. With a view to cover a large SCA threat range, we have acquired 6,400,000 traces and performed numerous analyses: Differential Power Analysis (DPA) [29], but also Correlation Power Analysis (CPA) [30], with both Hamming Weight (HW) and Hamming Distance (HD) models, by guessing one to four bits of the main

internal states of the algorithm, that is, the output values of R (register), XOR_K, SBoxes, but not XOR_L as it is tantamount to guessing R. We discard other analyses, such as the template attacks [31], as they have never proved to be practical on parallel implementations of block ciphers. Besides, Mutual Information Analysis (MIA) [32] outperforms CPA [33] only when the side-channel signals are noisy, and leakages multiple, which is not our case. Moreover, we restrict the number of hypotheses to those on the first round key, seen as eight classes of 6-bit each. Hence, an attack on the unprotected module using the HD can only concern R.

Once again, experiments confirm that CPA is more powerful than the DPA in presence of noise (traces have not been averaged) [34]. Best results on unprotected implementations are obtained with the HD model, as it matches the physical phenomenon responsible for power consumption (commutations), and by targeting four bits at the same time, increasing the signal-to-noise (SNR) ratio of the correct peak with respect to the peaks present in incorrect guesses. For WDDL, because of the zero value precharge, best analyses are done by guessing the HW as it equals the number of commutations: $\mathrm{HD}(0, x) = \mathrm{HW}(x)$. Targeting a single bit is more powerful than four bits: indeed, the leakage in WDDL is caused by the imbalance between the True and the False networks, and this unbalance could be the opposite for targeted bits and therefore counterbalance each other. The same observation has been basically done for quasidelay insensitive asynchronous circuits [35].

We observe that no PAR strategy resists SCA attacks. As the weak element of the implementation is R_S, we focus in the following on the robustness of each bit of this register. Table 3 summarizes the results for all of the three modules, which are truly comparable as traces have been acquired using the same experimental setup, and the same pseudo-random messages, generated from the same seed. Table 3(a) concerns an unconstrained WDDL cryptoprocessor, serving as reference to estimate the security gain provided by WDDL, and to study the impact of the differential PAR. Tables 3(b) and 3(c) deal, respectively, with the results of Vertical and Horizontal PAR strategies. For Tables 3(a), 3(b), and 3(c), bits coming from the same SBox have been grouped together, and their position in the R register after (P) permutation of 3DES is recalled by the second and sixth lines. For example, bit 1 of SBox 1 becomes bit 9 of R. The third and seventh lines correspond to the timing imbalance in picoseconds. They detail values of Table 2. "Zero" means under the resolution of the timing analysis tool (one picosecond). Fourth and eighth lines provide the SG defined in Section 3.2. The HW model monobit attack has been one hundred times reiterated to get a representative success rate. The SG is then computed with the minimal number of traces needed to recover the full secret key (100% success rate, worst case).

Boldface cells correspond to best cases, that is, smallest value for the timing imbalance and largest value for the SG. The presence of two boldface cells in a same column means that static and experimental evaluations are correlated. An overall look at Table 3 shows that it is never the case. This tends to confirm that evaluating the imbalance with delays is not accurate enough, as they are maximum values, and not typical. One solution to improve analysis may be to rather consider the type of the line, short or long, and to take into account the fanout.

Best cases show that SG is increased by at most a factor 22 when using WDDL without specific efforts of PAR, 364 when using the Horizontal PAR strategy. The Vertical PAR strategy seems to be the more robust, as bit 31 and bit 18, marked with black shaded cells, do not disclose the secret key using 6,400,000 messages. Unfortunately, a reliable security assessment considers the worst case. With 3DES, SG of one SBox equals the minimal SG amongst the SG of its four output bits. Thus, a more accurate conclusion is that the WDDL implementation without specific efforts of PAR increases the robustness a little; the Horizontal PAR strategy multiplies it by 18 to break the SBox 7, but by 6.5 on average; the SG of the Vertical PAR strategy equals 11 for the SBox 7, 5 on average. Referring to Table 1 of [4], MTDs of the unprotected and WDDL-protected modules of Kris Tiri equal 320 and 21,185, respectively, discarding the 5 key bytes not disclosed. The SG is thus close to 66, overriding by one order of magnitude that of our FPGA implementation. However, some bits of the Vertical Strategy present a high robustness: two of them do not result in a disclosure, and two others have an SG of 214 and 352. These results are very promising: we think that we could in the future increase the SG of all of the bits to this level. A reliable approach to reach this goal seems to be in-depth analyses of the link between the routing resources used by a differential path and the SG of its corresponding bit.

4. Conclusion and Perspectives

We have presented in this article a WDDL 3DES implementation with a fully-fledged placement and routing reaching a timing balance lower than 290 ps according to static evaluation. The unconstrained DPL design is shown to be definitely more secure than the reference design. In addition, the PAR strategies we introduced increase further the secure level on top of the unconstrained design. However, despite these encouraging results, the implementations are still attackable with SCA, albeit with more measurements, thus making the attack more difficult.

A detailed analysis has shown that weaknesses originate from register imbalance, although, up to now, efforts were devoted to the design of SBoxes. Security evaluation cannot rely on experiments on a single cryptographic primitive and should concern a complete and real-life application to avoid side effects, by means of solutions such as EveSoc and SASEBO.

Efficient differential place-and-route on FPGA appears to be a great challenge. Perspectives for future works are to validate the influence of technological characteristic variability over the same chip by renewing experiments at different locations, but on another FPGA families as well. We plan also to test other FPGA design solutions, such as Xilinx's ISE, which may propose more accurate timing analysis tools.

References

[1] L. Goubin and J. Patarin, "DES and differential power analysis," in *Workshop on Cryptographic Hardware and Embedded Systems*, LNCS, pp. 158–172, Springer, worcester, Mass,USA, Aug 1999.

[2] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *Proceedings of the Selected Areas in Cryptography (SAC '04)*, vol. 3357 of *LNCS*, pp. 69–83, Springer, Waterloo, Canada, August 2004.

[3] É Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater, "Improved higher-order side-channel attacks with FPGA experiments," in *7th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2005*, pp. 309–323, gbr, September 2005.

[4] K. Tiri and I. Verbauwhede, "A digital design flow for secure integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1197–1208, 2006.

[5] T. Popp and S. Mangard, "Masked dual-rail pre-charge logic: DPA-resistance without routing constraints," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 3659 of *LNCS*, pp. 172–186, Springer, Edinburgh, UK, August 2005.

[6] C. Zhimin and Z. Yujie, "Dual-rail random switching logic: a countermeasure to reduce side channel leakage," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 4249 of *LNCS*, pp. 242–254, Springer, Yokohama, Japan, 2006.

[7] P. Schaumont and K. Tiri, "Masking and dual-rail logic don't add up," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 4727 of *LNCS*, pp. 95–106, Springer, Vienna, Austria, September 2007.

[8] K. Tiri and I. Verbauwhede, "Synthesis of secure FPGA implementations," in *Proceedings of the International Workshop on Logic and Synthesis (IWLS '04)*, pp. 224–231, June 2004.

[9] S. Guilley, L. Sauvage, J.-L. Danger, T. Graba, and Y. Mathieu, "Evaluation of power-constant dual-rail logic as a protection of cryptographic applications in FPGAs," in *Proceedinsg of the Conference on Secure Software Integration and Reliability Improvement (SSIRI '08)*, pp. 16–23, IEEE Computer Society, Yokohama, Japan, July 2008.

[10] P. Yu and P. Schaumont, "Secure FPGA circuits using controlled placement and routing," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pp. 45–50, ACM, New York, NY, USA, 2007.

[11] R. P. McEvoy, C. C. Murphy, W. P. Marnane, and M. Tunstall, "Isolated WDDL: a hiding countermeasure for differential power analysis on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 1, pp. 1–23, 2009.

[12] K. Baddam and M. Zwolinski, "Path switching: a technique to tolerate dual rail routing imbalances," *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 207–220, 2008.

[13] K. Baddam and M. Zwolinski, "Divided backend duplication methodology for balanced dual rail routing," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 5154 of *LNCS*, pp. 396–410, Springer, Washington, DC, USA, August 2008.

[14] T. Popp, M. Kirschbaum, T. Zefferer, and S. Mangard, "Evaluation of the masked logic style MDPL on a prototype chip," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 4727 of *LNCS*, pp. 81–94, Springer, Vienna, Austria, September 2007.

[15] R. Soares, N. Calazans, V. Lomné, P. Maurine, L. Torres, and M. Robert, "Evaluating the robustness of secure triple track logic through prototyping," in *Proceedings of the 21st Annual Symposium on Integrated Circuits and Systems Design (SBCCI '08)*, pp. 193–198, ACM, New York, NY, USA, September 2008.

[16] V. Lomné, P. Maurine, L. Torres, M. Robert, R. Soares, and N. Calazans, "Evaluation on FPGA of triple rail logic robustness against DPA and DEMA," in *Proceedings of the Design, Automation and Test in Europe (DATE '09)*, pp. 634–639, IEEE, Nice, France, April 2009, track A4 (Secure embeddedimplementations).

[17] S. Guilley, F. Flament, R. Pacalet, P. Hoogvorst, and Y. Mathieu, "Security evaluation of a balanced quasi-delay insensitive library," in *Proceedings of the Design of Circuits and Integrated Systems (DCIS '08)*, p. 6, IEEE, Grenoble, France, November 2008, Session 5D—Reliable and Secure Architectures.

[18] S. Bhasin, J.-L. Danger, F. Flament et al., "Combined SCA and DFA countermeasures integrable in a FPGA design flow," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 213–218, IEEE Computer Society, Quintana Roo, México, December 2009.

[19] M. Nassar, S. Bhasin, J.-L. Danger, G. Duc, and S. Guilley, "BCDL: a high speed balanced DPL for FPGA with global precharge and no early evaluation," in *Proceedings of the Design, Automation and Test in Europe (DATE '10)*, pp. 849–854, IEEE Computer Society, Dresden, Germany, March 2010.

[20] S. Guilley, S. Chaudhuri, L. Sauvage et al., "Place-and-route impact on the security of DPL designs in FPGAs," in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST '08)*, pp. 26–32, IEEE, Anaheim, Calif, USA, June 2008.

[21] DES, 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[22] S. Guilley, S. Chaudhuri, L. Sauvage et al., "Place-and-route impact on the securityof DPL designs in FPGAs," in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST '08)*, pp. 26–32, IEEE, Anaheim, Calif, USA, June 2008.

[23] T. Akishita, M. Katagi, Y. Miyato, A. Mizuno, and K. Shibutani, "A practical DPA countermeasure with BDD architecture," in *Proceedings of the International Conference on Smart Card Research and Advanced Application (CARDIS '08)*, vol. 5189 of *Lecture Notes in Computer Science*, pp. 206–217, Springer, London, UK, September 2008.

[24] "Altera FPGA designer," http://www.altera.com.

[25] T.-H. Le, J. Clédière, C. Servière, and J.-L. Lacoume, "Noise reduction in side channel attack using fourth-order cumulant," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 4, pp. 710–720, 2007.

[26] SASEBO, http://www.rcis.aist.go.jp/special/SASEBO/index-en.html.

[27] SourceForge, https://sourceforge.net/projects/evesoc/.

[28] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '09)*, vol. 5479 of *LNCS*, pp. 443–461, Springer, Cologne,Germany, April 2009.

[29] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the International Cryptology Conference (CRYPTO '99)*, vol. 1666 of *LNCS*, pp. 388–397, Springer, Santa Barbara, Calif, USA, Augudt 1999.

[30] É Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 3156 of *LNCS*, pp. 16–29, Springer, Cambridge, Mass, USA, August 2004.

[31] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 2523, pp. 13–28, Springer, Redwood City, Calif, USA, August 2002.

[32] B. Gierlichs, L. Batina, and P. Tuyls, "Mutual information analysis—a universal differential side-channel attack," report 2007/198, 2007, http://eprint.iacr.org.

[33] E. Prouff and M. Rivain, "Theoretical and practical aspects of mutual information based side channel analysis," in *Proceedings of the conference on Applied Cryptography and Network Security (ACNS '09)*, vol. 5536 of *LNCS*, pp. 499–518, Springer, Paris-Rocquencourt, France, June 2009.

[34] S. Guilley, L. Sauvage, P. Hoogvorst, R. Pacalet, G. M. Bertoni, and S. Chaudhuri, "Security evaluation of WDDL and SecLib countermeasures against power attacks," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1482–1497, 2008.

[35] G. Bouesse, M. Renaudin, B. Robisson et al., "DPA on quasi delay insensitive asynchronous circuits: concrete results," in *Proceedings of the 19th Conference on Design of Circuits and Integrated Systems (DCIS '04)*, pp. 24–26, Bordeaux, France, November 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 879281, 13 pages
doi:10.1155/2010/879281

Research Article

True-Randomness and Pseudo-Randomness in Ring Oscillator-Based True Random Number Generators

Nathalie Bochard, Florent Bernard, Viktor Fischer, and Boyan Valtchanov

CNRS, UMR5516, Laboratoire Hubert Curien, Université de Lyon, 42000 Saint-Etienne, France

Correspondence should be addressed to Nathalie Bochard, nathalie.bochard@univ-st-etienne.fr

Received 26 February 2010; Revised 22 September 2010; Accepted 12 December 2010

Academic Editor: Lionel Torres

The paper deals with true random number generators employing oscillator rings, namely, with the one proposed by Sunar et al. in 2007 and enhanced by Wold and Tan in 2009. Our mathematical analysis shows that both architectures behave identically when composed of the same number of rings and ideal logic components. However, the reduction of the number of rings, as proposed by Wold and Tan, would inevitably cause the loss of entropy. Unfortunately, this entropy insufficiency is masked by the pseudo-randomness caused by XOR-ing clock signals having different frequencies. Our simulation model shows that the generator, using more than 18 ideal jitter-free rings having slightly different frequencies and producing only pseudo-randomness, will let the statistical tests pass. We conclude that a smaller number of rings reduce the security if the entropy reduction is not taken into account in post-processing. Moreover, the designer cannot avoid that some of rings will have the same frequency, which will cause another loss of entropy. In order to confirm this, we show how the attacker can reach a state where over 25% of the rings are locked and thus completely dependent. This effect can have disastrous consequences on the system security.

1. Introduction

True Random Number Generators (TRNGs) are used to generate confidential keys and other critical security parameters (CSPs) in cryptographic modules [1]. Generation of high rate and high quality random bit-stream inside logic devices is difficult because these devices are intended for implementing deterministic data processing algorithms whereas generating true-randomness needs some physical nondeterministic process.

The quality of the generated bit-streams is evaluated using dedicated statistical tests such as FIPS 140-2 [1], NIST 800-22 [2], and Diehard [3]. However, the statistical tests are not able to give a mathematical proof that the generator generates true random numbers and not only pseudo-random numbers that can be employed in attacks [4]. For this reason, Killmann and Schindler [5] propose to characterize the source of randomness from the raw binary signal in order to estimate the entropy in the generator output bit-stream.

The TRNGs implemented in reconfigurable devices usually use metastability [6–8] or the clock jitter [9–12] as the source of randomness. Many of them employ ring oscillators (ROs) as a source of a jittery clock [13–15]. One of principles employed the most frequently in reconfigurable devices is that proposed by Sunar et al. in [15]. This principle was later exploited and modified in [16] and enhanced in [17, 18]. Sources of randomness and randomness extraction in RO-based TRNG were analyzed in [19–21].

In order to increase the entropy of the generated binary raw signal and to make the generator "provably secure", Sunar et al. employ a huge number of ROs [15]. The outputs of 114 supposedly independent ROs are XOR-ed and sampled using a reference clock with a fixed frequency in order to obtain a raw binary signal. This binary signal is then postprocessed using a resilient function depending on the

size of the jitter and the number of ROs employed. The main advantages of the generator of Sunar are

(i) the claimed security level based on a security proof,

(ii) easy (almost "push button") implementation in FPGAs.

Without the security proof, the generator of Sunar et al. can be considered as just one of many existing TRNGs that passes the statistical tests. This security approach is essential for TRNG evaluation according to AIS31 [5] that is accepted as a de facto standard in the field. Unfortunately, the Sunar's security proof is based on at least two assumptions that are impossible or difficult to achieve and/or validate in practice [11]:

(i) the XOR gate is supposed to be infinitely fast in order to maintain the entropy generated in rings;

(ii) the rings are supposed to be independent.

Wold and Tan show in [18] that by adding a flip-flop to the output of each oscillator (before the XOR gate), the generated raw bit-stream will have better statistical properties: the NIST [2] and Diehard [3] tests will pass without postprocessing and with a significantly reduced number of ring oscillators.

It is commonly accepted that contrary to the original design of Sunar et al., the modified architecture proposed by Wold and Tan maintains the entropy of the raw binary signal after the XOR gate if the number of rings is unchanged. However, we believe that several other questions are worthy of investigation. The aim of our paper is to find answers to the following questions and to discuss related problems:

(i) Is the security proof of Sunar valid also for the generator of Wold and Tan?

(ii) What is the entropy of the generated bitstream after the reduction of number of rings?

(iii) How does security enhancement proposed by Fischer et al. in [17] modify the quality of the generated binary raw signal?

(iv) How should the relationship between the rings be taken into account in entropy estimation?

The paper is organized as follows: Section 2 analyzes the composition of the timing jitter of the clock signal generated in ring oscillators. Section 3 deals with the simulation and experimental background of our research. Section 4 compares the behavior of the two generators in simulations and in hardware. Section 5 discusses the impact of the size and type of the jitter on the quality of the raw bit-stream. Section 6 evaluates the dependence between the rings inside the device and its impact on generation of random bit-stream. Section 7 discusses the obtained results and replies to the questions given in the previous paragraph. Section 8 concludes the paper.

2. Ring Oscillators and Timing Jitter

Ring oscillators are free-running oscillators using logic gates. They are easy to implement in logic devices and namely in Field Programmable Gate Arrays (FPGAs). The oscillator consists of a set of delay elements that are chained into a ring. The set of delay elements can be composed of inverting and noninverting elements, while the number of inverting elements has to be an odd number. The period of the signal generated in the RO using ideal components is given by the form

$$T = 2\sum_{i=1}^{k} d_i, \tag{1}$$

where $k \in \{3, 4, 5, \ldots, n\}$ is the number of delay elements and d_i is the delay of the i-th delay element. This expression is simplified in two ways:

(i) the delay d_i is supposed to be constant in time;

(ii) the delays of interconnections are ignored.

In physical devices, the delay d_i varies between two half-periods (i.e., between two instances i and $i+k$) and expression (1) gets the form

$$T = \sum_{i=1}^{2k} d_{(i-1 \bmod k)+1}. \tag{2}$$

In the case of ring oscillators, the variation of the clock period is observed as the clock timing jitter, which can be seen as a composition of the jitter caused by local sources and the jitter coming from global sources, usually from power supply and/or global device environment [19]. When observing rings implemented in real devices and namely, in FPGAs, the delays of interconnections cannot be ignored anymore. For simplicity, we propose to merge them with the gate delays. This approach was validated in [21]. The delay d_i of gate i including interconnection delay between two consecutive gates in the ring oscillator can then be expressed as

$$d_i = D_i + \Delta d_{Li} + \Delta d_{\mathrm{G}i}, \tag{3}$$

where D_i is a constant delay of gate i plus interconnection between gates i and $i + 1 \pmod k$ corresponding to the nominal supply voltage level and nominal temperature of the logic device, Δd_{Li} is the delay variation introduced by local physical events, and $\Delta d_{\mathrm{G}i}$ is the variation of the delay caused by global physical sources such as substrate noise, power supply noise, power supply drifting, and temperature variation.

The delay d_{Li} is dynamically modified by some amount of a random signal $\Delta d_{\mathrm{LG}i}$ (LG-Local Gaussian jitter component) and by some local cross-talks from the neighboring circuitry $\Delta d_{\mathrm{LD}i}$ (LD-Local Deterministic jitter component). The jitter from local sources used in (3) can thus be expressed as

$$\Delta d_{Li} = \Delta d_{\mathrm{LG}i} + \Delta d_{\mathrm{LD}i}. \tag{4}$$

The local Gaussian jitter components $\Delta d_{\mathrm{LG}i}$ coming from individual gates and interconnections are characterized by normal probability distribution $N(\mu_i, \sigma_i^2)$ with the mean value $\mu_i = 0$ and the standard deviation σ_i. We can suppose that these sources are independent. On the other side, the

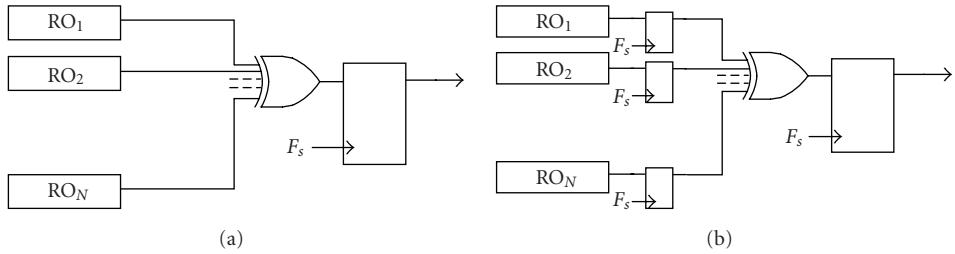


Figure 1: Original TRNG architecture of Sunar et al. (a) and modified architecture of Wold and Tan (b).

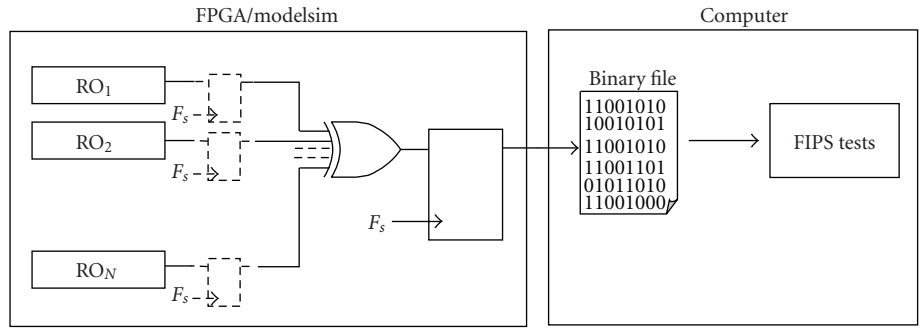


Figure 2: Simulation and experimental platforms.

local deterministic components can feature some mutual dependency, for example, from cross-talks.

Besides being influenced locally, the delays of all logic gates in the device are modified both slowly and dynamically by global jitter sources. The slow changes of the gate delay ΔD (the drift) can be caused by a slow variation of the power supply and/or temperature. The power source noise and some deterministic signal, which can be superposed on the supply voltage, can cause dynamic gate delay modification composed of a Gaussian global jitter component Δd_{GG} and a deterministic global jitter component Δd_{GD}. The overall global jitter from (3) can therefore be expressed as

$$\Delta d_{\mathrm{G}i} = K_i(\Delta D + \Delta d_{\mathrm{GG}} + \Delta d_{\mathrm{GD}}), \tag{5}$$

where $K_i \in [0; 1]$ corresponds to the proportion of the global jitter sources on the given gate delay. This comes from the fact that the amount of the global jitter included in delays of individual logic gates is not necessarily the same for all gates. It is important to note that K_i depends on the power supply voltage, but this dependence may differ for individual gates.

In real physical systems, the switching current of each gate modifies locally and/or globally the voltage level of the power supply, which in turn modifies (again locally and/or globally) the gate delay. This way, the delays of individual gates are not completely independent. We will discuss this phenomenon in Section 6.

3. Simulation and Experimental Setup

The aim of the first part of our work was to compare the behavior of two RO-based TRNGs: the original architecture depicted in Figure 1(a) that was proposed by Sunar et al. in [15] and its modified version presented in Figure 1(b) proposed by Wold and Tan in [18]. Contrary to the strategy adopted in [18], where the behavior of the two generators was compared only in hardware, we propose to compare it on simulation level, too. This approach has two advantages:

(i) the functional simulation results correspond to an ideal behavior of the generator, this way the two underlying mathematical models can be compared;

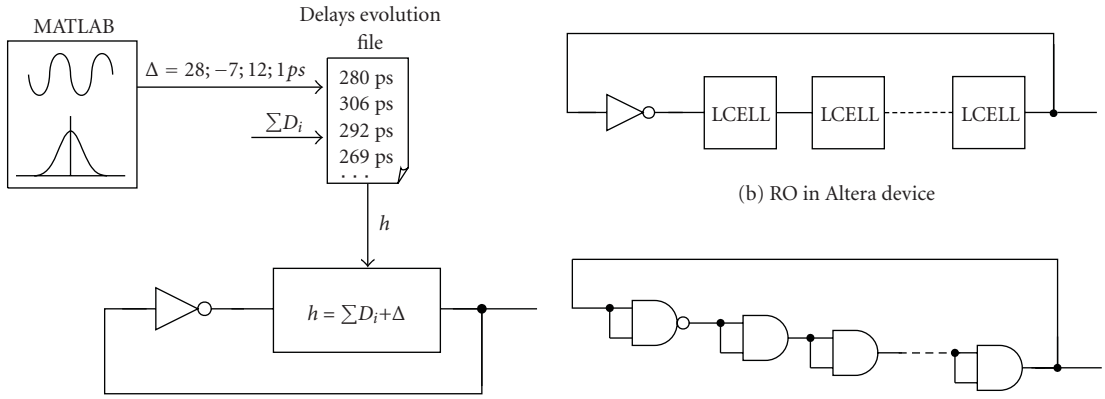
(ii) in contrast with the real hardware, thanks to simulation we can modify the parameters of injected jitter and evaluate the impact of each type of jitter on the quality of the generated bit-stream.

The principle of our simulation platform and experimental platform is depicted in Figure 2. For both platforms, the two generators were described in VHDL language and their architectures differed only in the use of flip-flops on the rings outputs (dashed blocks in Figure 2). The bitstreams obtained at the output of the final sampling flip-flop (before the post-processing) were tested and evaluated for different types and sizes of jitter in simulations and for different numbers of ring oscillators in both simulations and hardware experiments. The output of the TRNG was written into a binary file that was used as an input file in statistical tests.

We avoided the post-processing in the generator of Sunar et al. for two reasons:

(i) the post-processing function can hide imperfections in the generated signal;

(ii) using the same structures, we wanted to compare the two generators more fairly.



(a) RO in simulations

(b) RO in Altera device

(c) RO in Actel device

Figure 3: Implementation of ring oscillators in simulations and in hardware.

At this first level of investigation, we used the statistical tests FIPS 140-2 [1] in order to evaluate the quality of the generated raw bit-streams. We preferred the FIPS tests before the NIST test suite [2], because of the speed and the size of files needed for testing. While using significantly smaller files, the FIPS tests give a good estimation of the quality of the generated raw signal. If these tests do not pass, it is not necessary to go further. As our aim was to test a big set of TRNG configurations, the time and data size constraints were important. However, when the FIPS tests pass, we cannot conclude that the quality of the sequence produced by the TRNG is good. In this case, the generator should be thoroughly inspected.

3.1. Simulation Methodology. In order to compare the two generators on the functional simulation level (i.e., using ideal components), the behavior of ring oscillators was modeled in VHDL by delay elements with dynamically varying delays.

The jittered half-period, generated in MatLab Ver. R2008b, is based on (2) to (5). However, we take into account only local Gaussian jitter $\Delta d_{\mathrm{LG}i}$ (the source of true-randomness) and global deterministic jitter Δd_{GD} (the source of pseudo-randomness which can easily be manipulated) in our simulations. This approach was explained in [19]. Both sources of jitter depend a priori on the time t. Thus the simplified equation used in MatLab for generating jittery half-periods over the time t, denoted by $h(t)$, is expressed as

$$h(t) = \sum_{i=1}^{k} (D_i(t) + \Delta \mathrm{d}_{\mathrm{LG}i}(t) + K_i \times \Delta d_{\mathrm{GD}}(t)). \quad (6)$$

The fix part of the generated delay that determines the mean half-period of the ring oscillator is defined as a sum of k delay elements featuring constant delay $D_i(t) = D_i$ for each gate i. The variable delay that is added to the mean half-period is composed of a Gaussian component generated for each gate individually and of a deterministic component generated by the same generator for all gates and rings. The Gaussian delay component $\Delta d_{\mathrm{LG}i}(t)$ can be seen as a stationary process (i.e., mean and variance of $\Delta d_{\mathrm{LG}i}(t)$ do not change over the time t). Thus $\Delta d_{\mathrm{LG}i}(t)$ can be generated using the *normrnd* function in MatLab with mean 0 and standard deviation σ_i. The deterministic component $\Delta d_{\mathrm{GD}}(t)$ applied at time t is calculated in MatLab in the following way:

$$\Delta d_{\mathrm{GD}}(t) = A_{\mathrm{GD}} \times \sin(2\pi F_{\mathrm{GD}} \times t), \quad (7)$$

where A_{GD} and F_{GD} represent the amplitude and frequency of the deterministic signal. Note, that the *sin* function can be replaced with the *square*, *sawtooth*, or other deterministic function. Coefficients K_i are used to simulate the varying influence of $\Delta d_{\mathrm{GD}}(t)$ on each gate i.

Once the parameters k, D_i, K_i, σ_i, A_{GD}, and F_{GD} were set, a separated file containing a stream of half-period values was generated for each ring oscillator. These files were read during the VHDL behavioral simulations performed using the ModelSim SE 6.4 software, as presented in Figure 3(a).

The output signals were sampled using a D flip-flop at the sampling frequency $F_s = 32$ MHz, and the obtained 20,000 samples were written during the simulation to a binary file. Finally, generated sequences were tested using FIPS 140-2 tests.

3.2. Methodology of Testing in Hardware. Enhancements of the generator architecture brought by Wold and Tan were related to the behavior of the XOR gate. In order to compare generators' behavior in two different technologies, we employed one from Altera (the same that was used in [18]) based on Lookup tables (LUT) and one from Actel based on multiplexers. We developed two different modules dedicated to TRNG testing. Each module contained a selected FPGA device, a 16 MHz quartz oscillator and low-noise linear voltage regulators. The modules were plugged into a motherboard containing Cypress USB interface device CY7C6B013A-100AXC powered by an isolated power supply.

The Altera module contained the Cyclone III EP3C25F256C8N device. The noninverting delay elements and one inverter were mapped to LUT-based logic cells (LCELL) from Altera library (see Figure 3(b)). This way, either odd or even number of delay elements could be used in order to tune the frequency in smaller steps. We used

Quartus II software version 9.0 from Altera for mapping the rings into the device. All delay elements were preferably placed in the same logic array block (LAB) in order to minimize the dispersion of parameters.

The Actel Fusion module featured the M7AFS600FGG2-56 FPGA device. The non-inverting delay elements were implemented using AND2 gates from Actel library with two inputs short-connected and one inverter (again from Actel library) was added to close the loop (see Figure 3(c)).

On both hardware platforms, an internal PLL was used to generate the 32 MHz sampling clock F_s. The generated bitstreams were sent to the PC using the USB interface. A 16-bit interface communicating with the Cypress USB controller was implemented inside the FPGA. A Visual C++ application running on the PC read the USB peripheral and wrote data into a binary file that was used by the FIPS 140-2 tests software.

4. Comparison of the Generators' Behavior in Simulations and in Hardware

First, we compared the behavior of both generators in VHDL simulations. The generators used 1 to 20 ROs consisting of $k = 9$ inverters each. To take into account the differences related to the placement and routing of ROs, we supposed that the mean delay D_i of individual gates was slightly different from one ring to another (between 275 ps and 281 ps). The additional Gaussian jitter Δd_{LG_i} has mean 0 and standard deviation $\sigma_i = 30$ ps. No deterministic component was added in this experiment (i.e., $\Delta d_{\mathrm{GD}}(t) = 0$). It is important to note that the same random data files were used for both Sunar's and Wold's generators.

4.1. Simulation Results and Mathematical Analysis of the Generator Behavior. The simulation results for both evaluated generators are presented in Figure 4. Note that the "Monobit" and "Poker" tests succeed if the test result lies in the gray area. The "Run" test succeeds if no run fails. The "Long run" test is not presented in the graphs because it always succeeded.

It can be seen that in all configurations the two versions of the generator gave very similar (almost identical) results. Next, we will explain the reason for this behavior.

Let $b_j(t)$ be the bit sampled at time $t \geq 0$ at the output of the ring oscillator RO_j. This bit depends on the time t, the period $T_j > 0$ generated by RO_j, and the initial phase φ_j of RO_j at time $t = 0$.

Note that T_j, t, and φ_j are expressed in the time domain. This dependency is given by the following relation:

$$b_j(t) = 1 - \left\lfloor \frac{2\left(\varphi_j + t \bmod T_j\right)}{T_j} \right\rfloor. \tag{8}$$

Equation (8) ensures that $b_j(t)$ is a bit (i.e., $b_j(t) \in \{0, 1\}$).

Indeed, by definition of operation mod T_j,

$$0 \leq \left(\varphi_j + t\right) \bmod T_j < T_j, \tag{9}$$

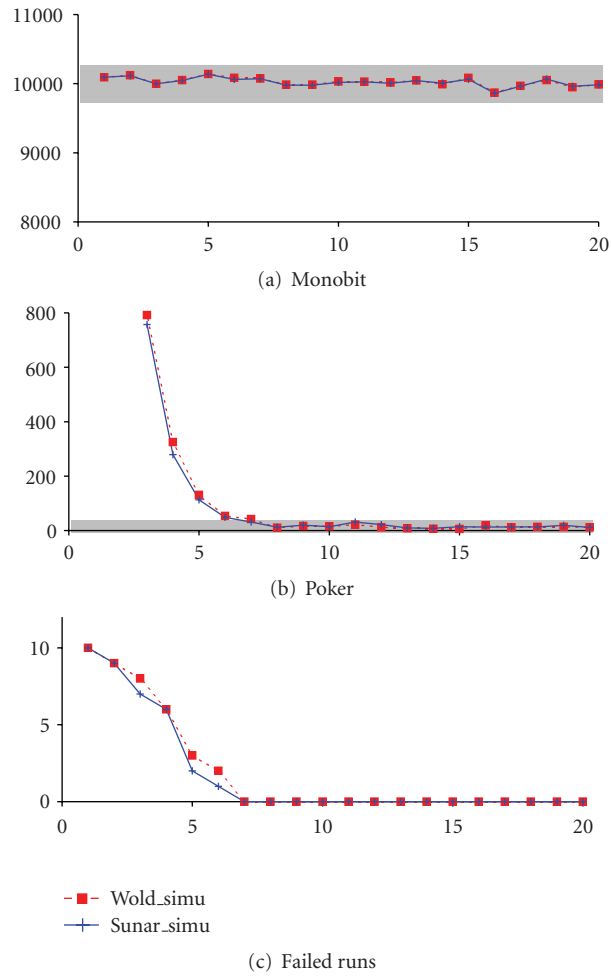


Figure 4: Results of the FIPS 140-2 tests in simulation with 30 ps of Gaussian jitter for Sunar's and Wold's architecture (excluding long runs that always passed), the tests pass if the results are in the gray region or are equal to zero for the "Runs" test.

then

$$0 \leq \frac{\left(\varphi_j + t\right) \bmod T_j}{T_j/2} < 2, \tag{10}$$

and by definition of the floor operation,

$$0 \leq \left\lfloor \frac{2\left(\varphi_j + t \bmod T_j\right)}{T_j} \right\rfloor \leq 1, \tag{11}$$

that means

$$\left\lfloor \frac{2\left(\varphi_j + t \bmod T_j\right)}{T_j} \right\rfloor \in \{0, 1\}, \tag{12}$$

thus

$$b_j(t) = 1 - \left\lfloor \frac{2\left(\varphi_j + t \bmod T_j\right)}{T_j} \right\rfloor \in \{0, 1\}. \tag{13}$$

Equation (8) holds if T_j is constant. Then if $(\varphi_j + t) \bmod T_j < T_j/2$, the sample value will be "1" otherwise it will be "0".

In both Sunar's and Wold's designs, outputs of N ring oscillators are XOR-ed to get one random bit as it is shown in Figure 1.

From the mathematical point of view, XOR-ing the outputs of N ring oscillators in Sunar's architecture is given by

$$S(t) = \bigoplus_{j=1}^{N} b_j(t) \tag{14}$$

or

$$\begin{aligned} S(t) &= \sum_{j=1}^{N} b_j(t) \bmod 2 \\ &= \sum_{j=1}^{N} \left(1 - \left\lfloor \frac{(\varphi_j + t) \bmod T_j}{T_j/2} \right\rfloor\right) \bmod 2. \end{aligned} \tag{15}$$

Thus

$$S(t) = \left(N + \sum_{j=1}^{N} \left\lfloor \frac{(\varphi_j + t) \bmod T_j}{T_j/2} \right\rfloor\right) \bmod 2. \tag{16}$$

This way, the nth bit sampled at time $n \times T_s$ in the D flip-flop is given by (16) for $t = n \times T_s$.

In Wold's architecture, each ring oscillator is sampled at time $n \times T_s$, which gives a set of bits $\{b_1(n \times T_s), b_2(n \times T_s), \ldots, b_N(n \times T_s)\}$. Then all these bits are XOR-ed and the result is sampled at time $(n+1) \times T_s$, giving the output of Wold's TRNG denoted by $W((n+1) \times T_s)$:

$$W((n+1) \times T_s) = \bigoplus_{j=1}^{N} b_j(n \times T_s). \tag{17}$$

Thus using equations (14) and (17), the relation between outputs of Wold's TRNG and Sunar's TRNG is given by

$$W((n+1) \times T_s) = S(n \times T_s). \tag{18}$$

We can conclude that from the mathematical point of view, assuming constant periods of ring oscillators and ideal components, Sunar's and Wold's generators can be described in the same way. The only difference is the right shift of the sequence from the Wold's generator. Note that this conclusion explains also the similar behavior of both generators in simulation as it is shown in Figure 4.

The claim that both ideal generators behave according to the same mathematical model is very important, because it means that the security proof of Sunar can be applied (at least theoretically) to both of them. However, as we will see in the next section, their behavior in hardware is very different.

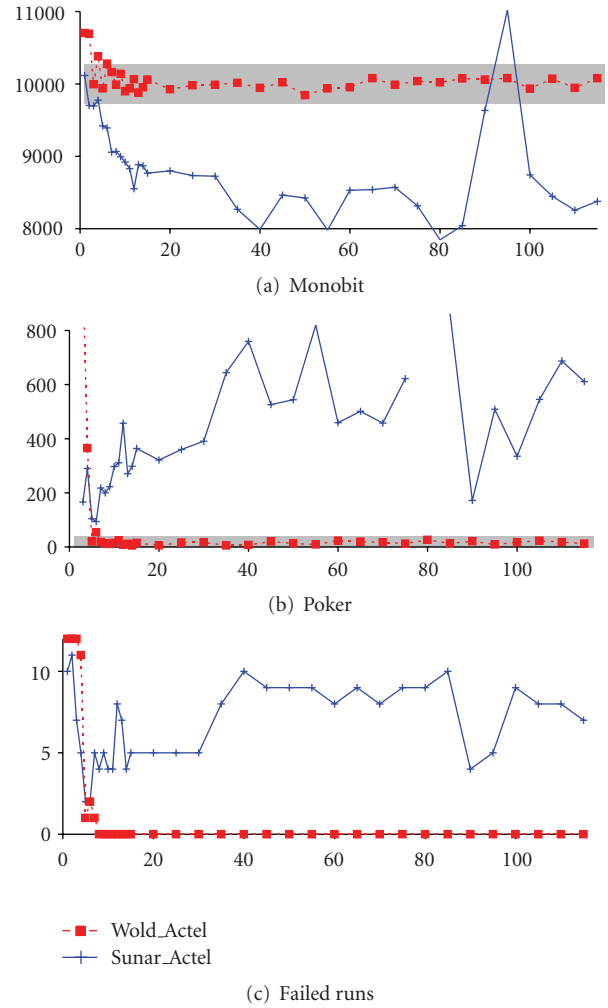


Figure 5: Results of the FIPS 140-2 tests for observed TRNG architectures with varying number of ring oscillators in Actel Fusion device.

4.2. Results Obtained in Hardware. We applied the FIPS 140-2 tests on the raw binary signals generated by the two generators, while incrementing the number of ROs. The results obtained for Actel FPGA are presented in Figure 5 and those obtained for Altera FPGA in Figure 6. The number of ROs varied from 1 to 20 by increments of 1 and from 20 to 115 by increments of 5.

It can be seen that for the Sunar's architecture the tests passed neither for Altera nor for Actel family. However, we can note that the Altera Cyclone III device gave slightly better results. This was probably due to the different behavior of the XOR gate in selected technologies. In the same time, concerning the architecture of Wold, the tests passed for both technologies if the number of ROs was higher than 8. Note that these results confirmed those obtained on standard evaluation boards from Altera and Actel published in [22].

The claims of Wold are thus confirmed in both technologies and various types of boards. However, it is not clear what kind of randomness lets the tests pass. Is it mostly a pseudo-randomness (coming from the sequential behavior

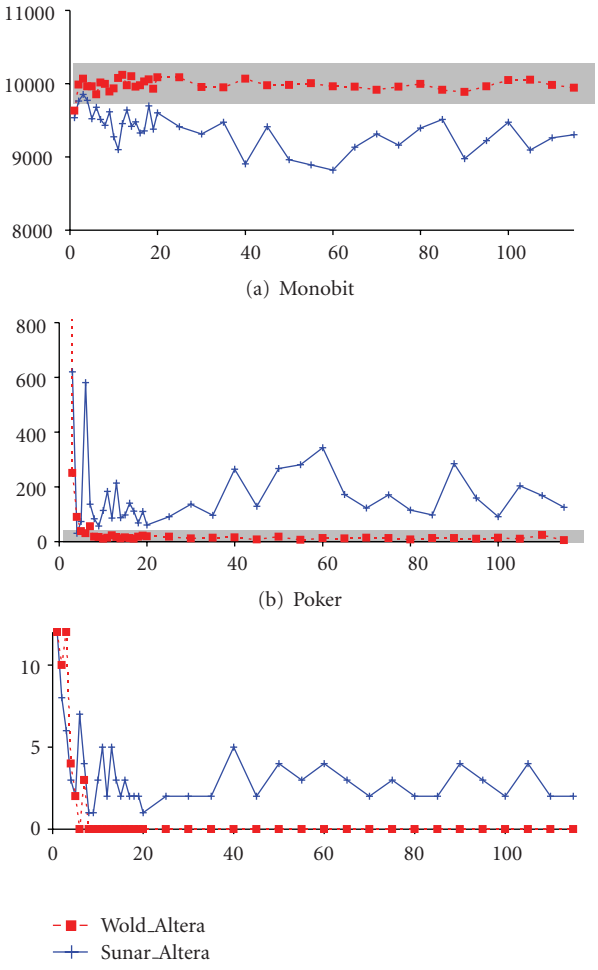


FIGURE 6: Results of the FIPS 140-2 tests for observed TRNG architectures with varying number of ring oscillators in Altera Cyclone III device.

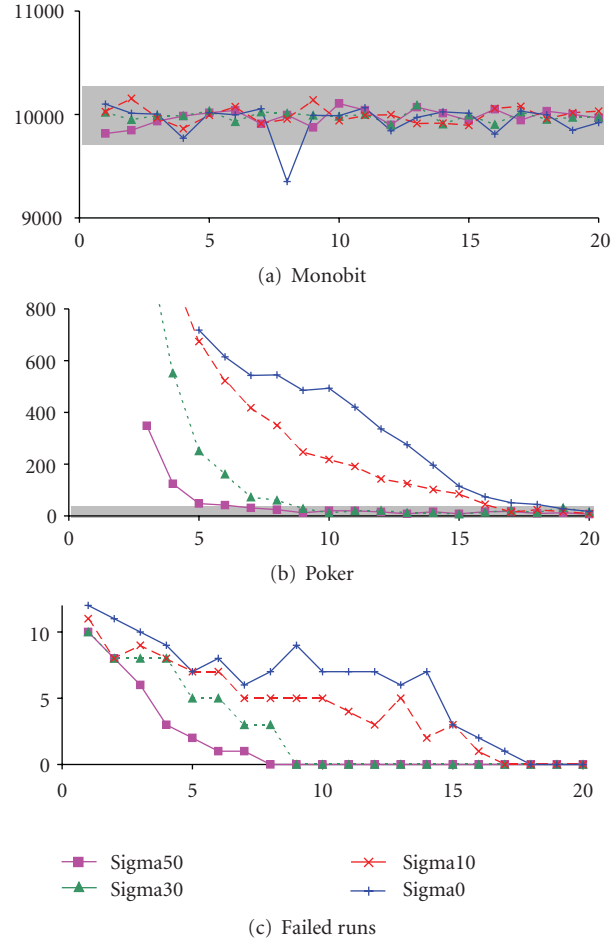


FIGURE 7: Results of the FIPS 140-2 tests in the Wold's TRNG architecture simulations with varying size of injected Gaussian jitter.

of the generator characterized by the internal state evolution) that can theoretically be attacked or a true-randomness that should be employed? The tests are clearly not able to distinguish between them. Again, the simulation can give answers to these questions.

5. Impact of the Size and Type of the Jitter on the Quality of the Raw Bit-Stream

As the architectures of Sunar and Wold have the same ideal behavior, we will analyze only the architecture of Wold and Tan, because its behavior in hardware is closer to the idealized mathematical model. Next, we will study the impact of both Gaussian and deterministic components of the jitter on the generated raw signal.

5.1. Impact of the Size of the Gaussian Jitter on the FIPS 140-2 Tests. Again, we have simulated the behavior of the Wold's architecture with 1 to 20 ROs composed of 9 elements. The half period of each RO was composed of a mean value (frequency of RO-generated clock signal varied between 197,5 MHz and 202 MHz in 250 kHz steps) and of an additional random value (normally distributed with mean 0 and σ_i = 0, 10, 30, and 50 ps for each gate). The random signals were generated in MatLab, independently for each RO. The results are presented in Figure 7.

Two facts can be observed in these figures:

(i) as expected, when the random jitter increases, the tests pass more easily (i.e., with a reduced number of ROs);

(ii) more surprisingly, the tests pass even if the random jitter is not injected at all ($\sigma = 0$), and this for 18 ROs or more.

Thanks to the mathematical model from (16), which generates the same sequences as the simulation tool, we could generate faster long bit-streams in order to perform NIST tests. The results were conclusive: the NIST tests passed with this full deterministic behavior (without any randomness) for only 18 ROs.

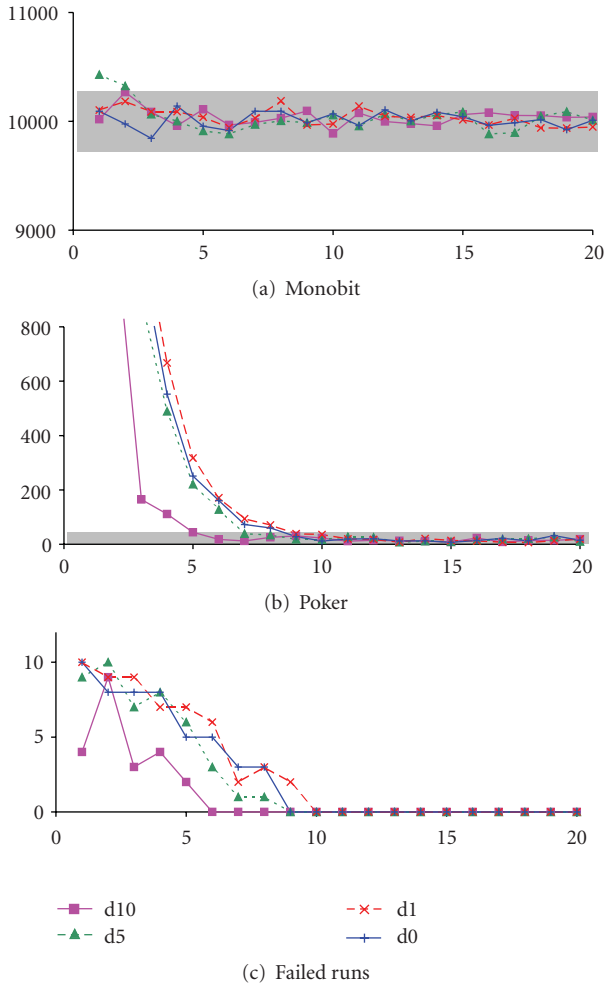


FIGURE 8: Results of the FIPS 140-2 tests in the Wold's TRNG architecture simulations while injecting a Gaussian jitter of 30 ps and a varying deterministic jitter.

The fact that the tests (FIPS and NIST) pass for a few ROs without Gaussian jitter means that both Sunar's and Wold's architectures produce a great amount of pseudo-randomness. We recall that pseudo-randomness in the generated sequence depends on the frequencies of ROs and can be manipulated from outside the chip (e.g., by modulating the power supply or by an electromagnetic interference as it was presented recently at the CHES conference [23]). Furthermore, using a mathematical model (e.g., that from (16)), some patterns can be predicted. The proportion of pseudo-random and true-random components in the generated sequence is thus very important. However, in the solution proposed by Wold, the number of ROs is significantly reduced, because the NIST tests passed. This can be considered as a security-critical attempt for cryptography applications and should certainly be avoided.

5.2. Injecting a Deterministic Jitter. In the next experiments, the Gaussian jitter remained constant ($\sigma_i = 30$ ps per gate) and we applied a sinusoidal deterministic jitter of 3 kHz and 0 to 10 ps in amplitude. The results are presented in Figure 8.

It can be seen that when the deterministic part increases, the tests pass more easily. But there are two problems concerning the deterministic component of the jitter:

(i) the results are strongly dependent on the frequency of the injected signal: depending on the frequency, the output of the TRNG can vary in time in a predictable way;

(ii) the deterministic jitter can be manipulated (for example, an attacker can superimpose a chosen signal that seems to improve randomness so that the tests would pass more easily, but in fact, he can predict some trends in subsequences).

For the above-mentioned reasons, the designer should reduce the pseudo-randomness coming from the deterministic jitter component as much as possible.

5.3. Reducing the Influence of the Deterministic Jitter Component. As we pointed out in [17], the impact of the deterministic jitter on the generated random numbers can significantly be reduced by the use of a reference clock signal featuring the same deterministic global jitter. This fact is not taken into account in any observed TRNG designs [15, 16, 18]. In the following experiments, we used a clock signal generated by another internal ring oscillator as a sampling clock. The generated signal frequency was divided by 8 and then used as a sampling clock (having thus the frequency of about 25 MHz). We implemented the Wold's TRNGs with 1 to 20 ROs. In the first experiment, we did not inject the deterministic jitter component and we let the Gaussian part vary between 0 and 50 ps. The results are presented in Figure 9. Next, we fixed the Gaussian jitter to $\sigma = 30$ ps and we applied a sinusoidal deterministic jitter of 3 kHz and 0 to 10 ps in amplitude. The results are presented in Figure 10.

As expected, increasing the standard deviation of the injected Gaussian jitter component from 0 to 50 ps, the tests passed more easily for both external and internal sampling clocks (see Figures 7 and 9, respectively). It is important to note that for a fixed Gaussian jitter size, tests passed more easily if the sampling was performed with an internal clock signal (i.e., from another ring oscillator), because independent Gaussian jitter components were included in both sampling and sampled signals and they increased the entropy of the generated bit-stream.

In the last experiment, the Gaussian jitter component was constant and the deterministic jitter component varied. The results for external and internal sampling clocks are presented in Figures 8 and 10, respectively. Contrary to the use of an external clock, the influence of the deterministic jitter was strongly reduced when an internal clock was used. As expected, whatever the size of the injected deterministic jitter component, the test results were similar.

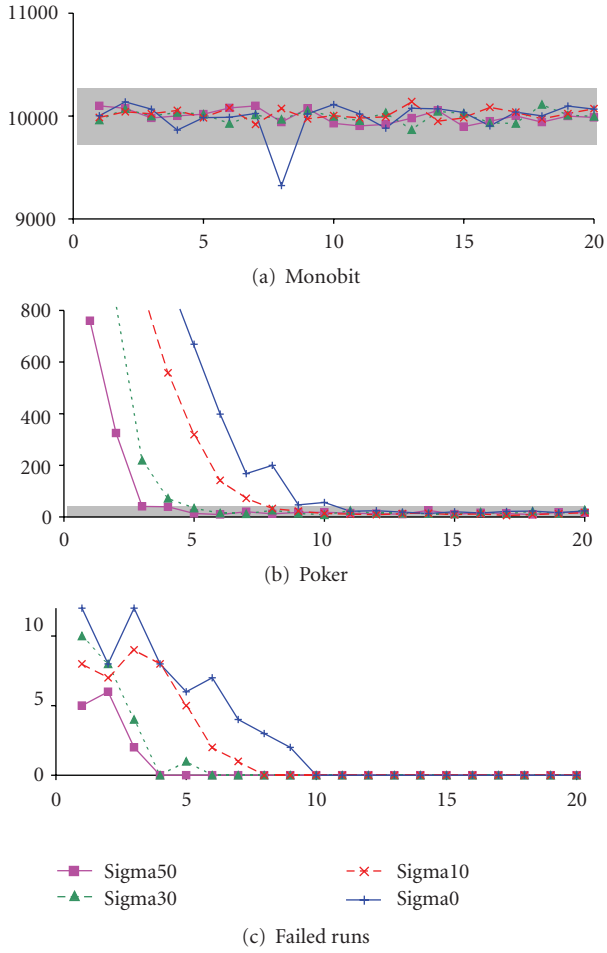


Figure 9: Results of the FIPS 140-2 tests of the Wold's TRNG architecture simulations with internal reference clock and varying size of injected Gaussian jitter.

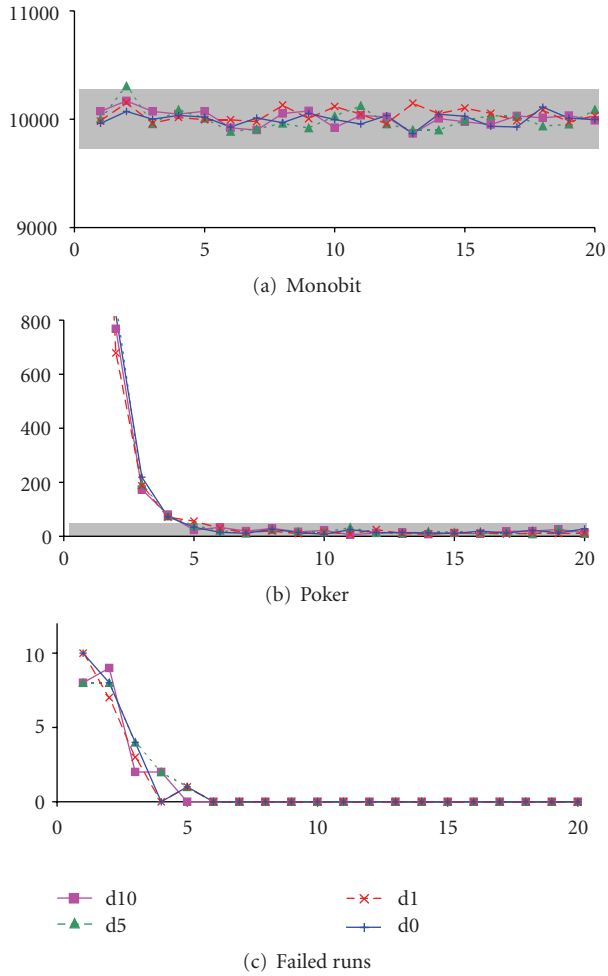


Figure 10: Results of the FIPS 140-2 test of the Wold's TRNG architecture simulations while injecting a Gaussian jitter with $\sigma = 30$ ps and a varying size of deterministic jitter, with internal reference clock.

6. Mutual Dependence of Rings and Its Impact on the Quality of the Raw Bit-Stream

The aim of the following experiment was to validate the mutual independence of ring oscillators implemented in FPGA. Note that this mutual independence of rings is a necessary condition for the validity of the security proof from [15].

First, we implemented pairs of 9-element ROs with similar topology (similar routing) in both tested FPGA technologies. The generated clock periods were measured using the high bandwidth (3.5 Ghz) digital oscilloscope LeCroy WavePro 735Zi. The 1.2 V power supply of the FPGA core was replaced with an external variable power supply. The output clock signal periods were measured for the power supply ranging from 0.9 V to 1.3 V.

We can observe in Figure 11 that for the Actel Fusion family the periods of the generated clock signals depend on the power supply in very similar, but not exactly the same way. We note that for a certain voltage interval the periods overlap, while outside this interval they tend to separate slightly. This corresponds to the use of coefficients K_i in Section 2 and to the claim that they depend on the power supply, but differently for each gate or ring.

The difference between the two clock periods as the function of the power supply can be observed in Figure 12. We can notice that it changes from negative to positive values following a monotonously rising curve; but suddenly, in the interval from 1.02 to 1.12 V, it drops almost to zero. The only explanation of this effect is that the rings become locked (otherwise the curve should drop to zero only in one point). The fact that the period difference is not zero is explained later. We can conclude that in the locking range the role of coefficients K_i on determination of the frequency (period) is overruled by some other phenomenon. This is coming probably from the dependence of the frequency of one oscillator on the current peaks caused by rising and falling edges of the second oscillator, as it is explained in the end of Section 2. This phenomenon was not observed in the literature up to now.

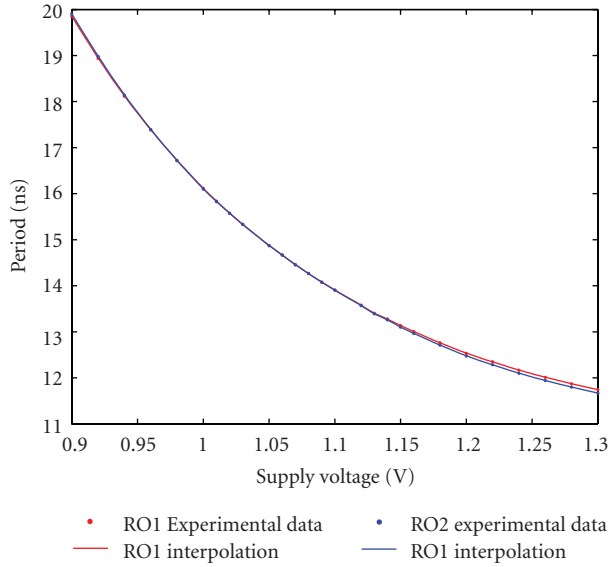


Figure 11: Dependence of clock periods of two rings in Actel Fusion device on the power supply.

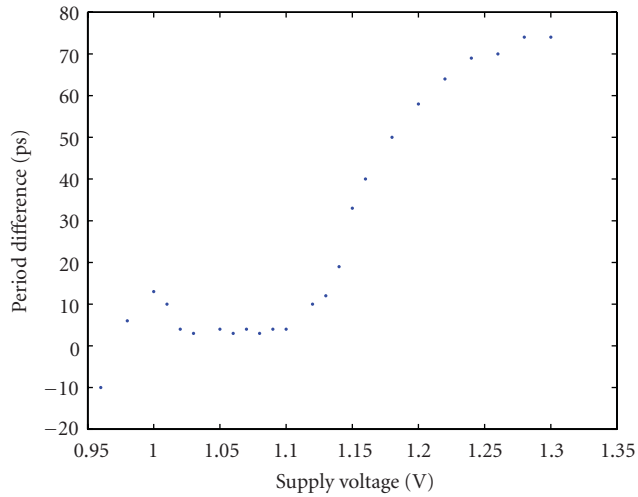


Figure 12: Difference between clock periods of two rings depending on the power supply.

The locking of two rings was also observable on the oscilloscope, as it is depicted in Figure 13 using the screen persistence. We could observe several phenomena during our experiments:

(i) when the rings' frequencies were sufficiently close, it was quite easy to lock the rings by modifying the power supply;

(ii) the locking could be observed for both technologies used;

(iii) although most of the time the phase of the two signals on the oscilloscope was perfectly stable, sometimes they became unlocked for a very short time—this explains why the period difference measured in long time intervals was not exactly zero in the locking zone;

(iv) we could quite easily obtain the state when about 25% of rings were locked—most of them were locked on a dominant frequency and other smaller groups of rings on other frequencies (this phenomenon was observed on all tested cards—five cards for each evaluated technology);

(v) the state when two or more rings were locked on the nominal voltage level (without manipulating the power supply) could also be obtained.

7. Discussion

As it was shown in Section 2, the generator of Sunar et al. and the generator of Wold and Tan are based on the same mathematical model (see (16)), when built using ideal components. This fact means that the proof of Sunar can be valid for both architectures. We recall that the proof of Sunar is based on the entropy estimation based on the jitter measurement before the XOR gate, while supposing that this gate does not reduce the entropy. However, this last assumption is not true for the architecture of Sunar when implemented in physical devices. The architecture of Wold and Tan solves this problem and should be preferred.

Nevertheless, even the new architecture does not eliminate serious doubts about the entropy contents in the raw signal. Unfortunately, this entropy cannot be measured. Applying the theory of Sunar et al., the entropy of the raw binary signal can be estimated knowing the sampling frequency, size of the jitter, and number of independent rings. Supposing that the rings are independent, this theory remains valid for the new generator architecture as we showed in Section 2. For this reason, we can conclude that while reducing number of rings, Wold and Tan reduced unconsciously the entropy of the generated signal. In order to maintain the security level, they should also modify the resilient function, in order to increase the compression ratio and to guarantee the output entropy per bit close to one. Instead, they propose to remove the post-processing, which is clearly a very dangerous action from the point of view of security.

Equation (16) implies that both generators contain some memory element (they are not memoryless in the sense of term used in [5]). This means that besides the true random behavior coming from the Gaussian jitter, they will feature a pseudo-random behavior. This behavior can be described for any time instant t depending on the previous generator state characterized by phases φ_j of N rings that generate clocks with periods T_j. Following the principle of the generator, the presence of this kind of pseudo-randomness in generated bitstream is unavoidable.

There is another source causing the pseudo-randomness in the raw binary signal. It comes from the global deterministic jitter sources. Both above-mentioned sources of pseudo-randomness are dangerous because they can mask the entropy insufficiency (the tests of randomness will pass)

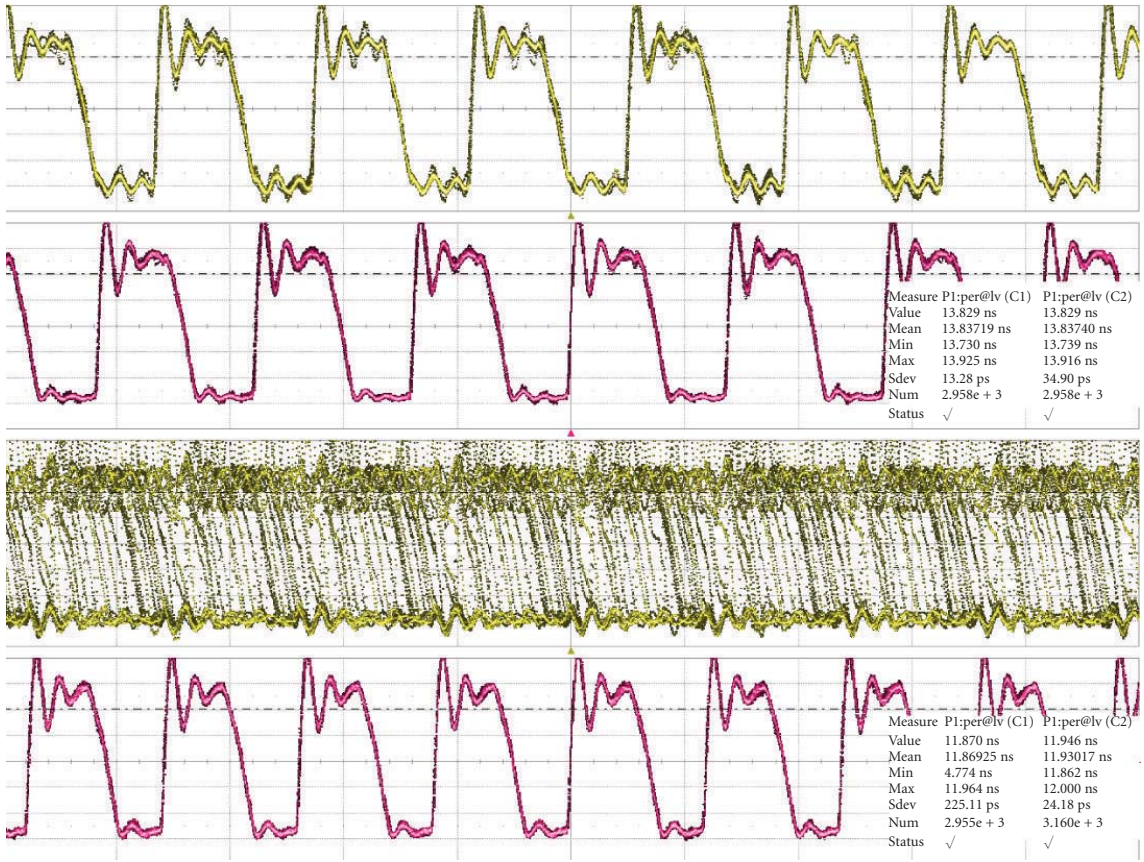


Figure 13: Waveforms of two locked (up) and unlocked (down) rings in Actel Fusion device.

and at the same time they can be manipulated. However, the pseudo-randomness coming from the evolution of the state of the generator described by (16) is more dangerous, because it can have two impacts: the attacker can manipulate the contents of the bit-stream and at the same time the entropy can be reduced.

For example, by modulating the power supply and thus changing the periods T_j, the attacker can control the pseudo-random behavior of the generator to some extent (mutual relations between clock periods) and the state can be reached where the rings are locked. This way, the effective number of usable (independent) rings is reduced. As in the case of the generator of Wold and Tan, the reduced number of rings will lower the entropy of the generated signal and at the same time the generator's pseudo-random behavior will be simpler and thus easier to guess.

8. Conclusions

As it was shown, the generator of Wold and Tan follows the same mathematical model as that of Sunar et al. The security proof of Sunar can thus be applied (theoretically) also in this case. Because the generator of Wold and Tan gives much better binary raw signal in hardware, it should be preferred. However, in order to assure that the proof of Sunar will hold, the number of rings should not be reduced as proposed in [18] only because the tests passed. As we showed, the generator using more than 18 ideal jitter-free rings having slightly different frequencies and producing only the manipulable pseudo-randomness will always let the tests pass.

In an ideal case (i.e., when the rings are independent) and following the proof of Sunar, the number of rings is defined by the size of the Gaussian component of the jitter and by the reference clock frequency, both in relationship with the post-processing resilient function. However, even if the number of rings remains high, some of them could be locked and the effective number of exploitable rings could be significantly lower. In this case, which is easy to obtain in real FPGAs and which can concern as much as 25% of rings or more, the entropy of the generated raw signal would be much lower than expected and the generator would be predictable or manipulable.

The locking of rings depends on their topology (placement and routing) and on the technology used. The probability of locking could perhaps be reduced by a careful placement and routing on a per-device basis or by an independent powering of all rings. Applying the first approach, the designer loses the main advantage of this class of TRNGs—device-independent design. The second approach is impossible to apply in FPGAs. Another strategy

can consist in detection of locking of rings in order to stop the generation of random numbers. However, the complexity of the detection circuitry would rise with the square of number of rings and it would thus limit the practical use of the generator, which is already penalized by the fact that the number of rings is considerable.

Although locking of rings can have disastrous consequences on the security of TRNGs based on ring oscillators, this phenomenon is not yet observed in the literature. For this reason, it should be studied extensively in the future.

Acknowledgments

This paper is an extended version of the conference paper [22] presented at ReConFig09. The research was partially supported by a Grant from the French National Research Agency-ANR-09-SEGI-013 and by the Rhone-Alpes Region and Saint-Etienne Metropole, France.

References

[1] Information Technology Laboratory, "FIPS 140-2: Security Requirements for Cryptographic Modules," Special Publication 140-2, May 2001.

[2] A. Rukhin, J. Soto, J. Nechvatal et al., "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST Special Publication 800-22, 2001, http://csrc.nist.gov/.

[3] G. Marsaglia, "DIEHARD: Battery of Tests of Randomness," 1996, http://stat.fsu.edu/pub/diehard/.

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, NY, USA, 1997, http://www.cacr.math.uwaterloo.ca/hac/.

[5] W. Killmann and W. Schindler, *AIS 31: Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators, Version 3.1*, Bundesamt fur Sicherheit in der Informationstechnik (BSI), Bonn, Germany, 2001.

[6] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy, "Fast digital TRNG based on metastable ring oscillator," in *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '08)*, E. Oswald and P. Rohatgi, Eds., vol. 5154 of *Lecture Notes in Computer Science*, pp. 164–180, Springer, Washington, DC, USA, August 2008.

[7] B. Fechner and A. Osterloh, "A meta-level true random number generator," *International Journal of Critical Computer-Based Systems*, vol. 1, no. 1–3, pp. 267–279, 2010.

[8] M. Varchola and M. Drutarovsky, "New high entropy element for FPGA based true random number generators," in *Proceedings of the 12th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '10)*, S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *Lecture Notes in Computer Science*, pp. 351–365, Springer, Santa Barbara, Calif, USA, August 2010.

[9] V. Fischer and M. Drutarovský, "True random number generator embedded in reconfigurable hardware," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES '02)*, B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, pp. 415–430, Springer, Redwood Shores, Calif, USA, 2003.

[10] K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "Compact FPGA-based true and pseudo random number generators," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 51–61, 2003.

[11] M. Dichtl and J. D. Golić, "High-speed true random number generation with logic gates only," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '07)*, P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, pp. 45–62, Springer, Vienna, Austria, September 2007.

[12] J.-L. Danger, S. Guilley, and P. Hoogvorst, "Fast true random generator in FPGAs," in *Proceedings of IEEE North-East Workshop on Circuits and Systems (NEWCAS '07)*, pp. 506–509, Montreal, Canada, August 2007.

[13] T. E. Tkacik, "A hardware random number generator," in *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES '02)*, B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, pp. 450–453, Springer, Redwood Shores, Calif, USA, 2003.

[14] P. Kohlbrenner and K. Gaj, "An embedded true random number generator for FPGAs," in *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '04)*, vol. 12, pp. 71–78, Monterey, Calif, USA, February 2004.

[15] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Transactions on Computers*, pp. 109–119, 2007.

[16] D. Schellekens, B. Preneel, and I. Verbauwhede, "FPGA vendor agnostic true random number generator," in *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 139–144, August 2006.

[17] V. Fischer, F. Bernard, N. Bochard, and M. Varchola, "Enhancing security of ring oscillator-based TRNG implemented in FPGA," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 245–250, September 2008.

[18] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in FPGA based on oscillator rings," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 501672, 8 pages, 2009.

[19] B. Valtchanov, A. Aubert, F. Bernard, and V. Fischer, "Modeling and observing the jitter in ring oscillators implemented in FPGAs," in *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS ')*, pp. 158–163, Bratislava, Slovakia, April 2008.

[20] V. Rožić and I. Verbauwhede, "Random numbers generation: investigation of narrowtransitions suppression on FPGA," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, M. Danek, J. Kadlec, and B. Nelson, Eds., pp. 699–702, Prague, Czech Republic, August-September 2009.

[21] B. Valtchanov, V. Fischer, A. Aubert, and F. Bernard, "Characterization of randomness sources in ring oscillator-based true random number generators in FPGAs," in *Proceedings of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '10)*, pp. 48–53, Vienna, Austria, April 2010.

[22] N. Bochard, F. Bernard, and V. Fischer, "Observing the randomness in RO-based TRNG," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 237–242, December 2009.

[23] A. T. Markettos and S. W. Moore, "The frequency injection attack on ring-oscillator-based true random number generators," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '09)*, C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, pp. 317–331, Springer, Lausanne, Switzerland, September 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 180242, 11 pages
doi:10.1155/2010/180242

Research Article

Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification

Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner

Department of Computer Science, Faculty for Electrical Engineering, Computer Science and Mathematics, Computer Engineering Group, Warburger Str. 100, 33098 Paderborn, Germany

Correspondence should be addressed to Stephanie Drzevitzky, stephanie.drzevitzky@upb.de

Received 8 March 2010; Revised 18 October 2010; Accepted 17 December 2010

Academic Editor: Lionel Torres

Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The capability to load hardware modules at runtime provides these systems with an unparalleled degree of adaptivity but at the same time poses new challenges for security and safety. In this paper, we elaborate on the presentation of proof carrying hardware (PCH) as a novel approach to reconfigurable system security. PCH takes a key concept from software security, known as proof-carrying code, into the reconfigurable hardware domain. We outline the PCH concept and discuss runtime combinational equivalence checking as a first online verification problem applying the concept. We present a prototype tool flow and experimental results demonstrating the feasibility and potential of the PCH approach.

1. Introduction

Dynamically reconfigurable hardware combines hardware performance with software-like flexibility and finds increasing use in networked systems. The dynamic reconfiguration capability provides networked systems with the flexibility to download new hardware functionality or hardware updates as needed. Since system downtime is often unacceptable, newly received hardware modules would have to be installed and run without previous extensive system testing and verification. However, due to its wide applicability reconfigurable hardware is often used in security and safety critical systems where an unintended system behavior could have severe consequences, such as heavy financial damage, loss of human life, and threat to national security, assuring the absence of unwanted and malicious behavior caused by outside attacks as well as internal construction flaws becomes essential in such scenarios.

The novel contribution of this paper is the presentation and elaboration of *proof-carrying hardware (PCH)* as an approach to reconfigurable system security, substantiated with an in-depth depiction of a prototype tool flow including experimental results. PCH takes a key concept from software security, known as proof-carrying code [1], into the reconfigurable hardware domain. We envision a scenario where an embedded reconfigurable target system, that is, the consumer, requests new functions with certain security guarantees at runtime. In response, reconfigurable hardware modules are being created by a design center, that is, the producer. The producer has to invest the substantial computational resources required to create a proof. The proof is then combined with the reconfigurable module into a binary and delivered as *proof-carrying bitstream*. The consumer can quickly verify the security property and, if successful, instantiate and run the hardware module. In essence, the consumer is enabled to only run verified hardware modules without having to trust the producer or rely on a secured transmission process or having to compute a formal proof of security features.

The PCH concept is rather general and can be applied to many types of security properties and corresponding proof systems and proofs. As a first application of PCH, we focus on runtime combinational equivalence checking (CEC) in this paper. Runtime CEC enables a consumer

to verify that a requested reconfigurable module actually adheres to its functional specification and, thus, eliminates a major security risk according to [2].

This paper extends our previous conference publication [3] by presenting an extended discussion of the PCH concept and a more elaborated CEC tool flow, using more test functions to demonstrate the feasibility of the PCH concept, and discussing more detailed and conclusive measurements including, for example, memory requirements.

The paper is organized as follows. In Section 2, we review related work. Section 3 presents the novel concept of proof-carrying hardware and elaborates on trust and threat models. Section 4 focuses on runtime CEC as a first PCH application. The implementation of the CEC tool flow, including experimental results and test functions, is depicted in Section 5. A conclusion and further work including possible other safety properties suited for PCH are presented in Section 6.

2. Related Work

In this section we review approaches to reconfigurable hardware security, an area that has gained interest only recently. An overview of security risks present in every step of the life cycle of reconfigurable hardware is presented by Kastner and Huffmire [2]. Drimer [4] also provides an exhaustive survey of possible attacks, involved parties, stages of life cycles, and possible defenses. For a detailed elaboration on FPGA design security, see [5].

A first step towards security of dynamically reconfigurable hardware is to establish trust in the bitstream transmission. Typically, FPGA bitstreams are minimally secured by checksums and some FPGA vendors even offer built-in hardware support for bitstream decryption and embedded keys. Chaves et al. [6] propose a more flexible approach based on hashing to secure correct bitstream delivery. The same authors also address another aspect of reconfigurable hardware security: they interpret the incoming bitstream to check whether the physical regions on the FPGA that are to be reconfigured match the intended reconfiguration area. Drimer and Kuhn [7, 8] distinguish between authentication and confidentiality and discuss a security protocol that combines both aspects to prevent system downgrades. Similar to this, Badrignans et al. propose a combination of special architecture and protocol to avoid the usage of old configurations in [9]. These techniques, however, assume that the module producer can be trusted and implemented the correct functionality and do not inspect functional properties of the reconfigurable modules at runtime.

Huffmire et al. [10] focus on multicore reconfigurable systems where several cores access the same memory. The authors define a set of high-level policies covering security scenarios such as Chinese Wall and Secure Hand-Off. The designer uses a formal regular language to describe valid memory accesses for a core and the interactions included in the policies. Starting from this formal specification, a synthesis tool generates a memory reference monitor. In [11], Huffmire et al. propose to secure IP cores on FPGAs with physical isolation primitives called moats and drawbridges. While moats prevent unwanted and unanticipated communication between cores, drawbridges allow for controlled and secure communication channels. Drawbridges are especially useful when also a reference monitor is invoked which enforces a memory policy specified for the intended access scenario. The combination of physical isolation primitives and reference monitors and the threats addressed by these techniques are also discussed in [12–14].

Many methods for functional verification known from the domains of software and (static) hardware verification rely on model checking (see, e.g., [15], for a survey). For example, Singh and Lillieroth [16] propose the Core Verification Flow which conducts a decomposition of a core and extracts a reference design for each entity to be verified from the core's behavioral specification. Eventually, they receive a logical formula against which the implementation is compared with NP-Tools or Prover Software. The Core Verification Flow runs the complete runtime and resource consuming analysis and security verification on the system that also executes the core.

The PCH approach presented in this paper shifts the computational burden for establishing security to the producer of a module which allows us to provide security also to embedded consumer platforms with limited computational power. Moreover, the PCH approach can be used for a multitude of nonfunctional and functional properties of reconfigurable hardware modules, including the above-mentioned ones.

3. The Transition from Proof-Carrying Code to Proof-Carrying Hardware

In this section we first introduce the principles of proof-carrying code and then transfer the concept to the reconfigurable hardware domain. Finally, we elaborate on trust and threat models that can be covered by proof-carrying hardware.

3.1. Principles of Proof-Carrying Code. In 1996, Necula and Lee proposed Proof-Carrying Code [1] as a means to validate code from an untrusted source before execution. The scenario includes a code consumer and a code producer. Both have to agree on a safety policy that includes formalisms to capture characteristics of the consumer's execution platform (e.g., machine model, memory accesses, type definitions) and to describe the desired safe code behavior. Depending on the actually chosen formalisms, producer and consumer must also decide on a corresponding proof system.

The producer then creates the program for the required functionality and, depending on the safety policy, may annotate the code with preconditions, invariants, and post-conditions. This step is crucial, since the annotated code together with the safety policy is transformed into the so-called safety predicate. The safety predicate is then proven to hold true in all states of the program. The proof is combined with the code into the proof-carrying code delivered to the

consumer. The consumer verifies the received proof and, thereby, checks the code's compliance with the safety policy.

The essence of proof-carrying code is to shift the burden of verification from the consumer to the producer, leaving the consumer with simply checking the delivered proof against the code, a task of insignificant size compared to the actual computation of the proof. Thus, proof-carrying code techniques are of special interest for target systems (consumers) with limited computational resources or systems that need to quickly extend their functionality by downloading mobile code.

The great universality of the proof-carrying code approach has been demonstrated in several case studies. For example, in [1] a subroutine scans incoming network packets and determines whether they should be accepted as being valid or rejected. The safety policy has to secure memory safety and guarantee the termination of the subroutine. Memory safety is ensured in this case by requiring that a designated argument register holds the start address of a package in memory and another register contains the package's size. A further case study computing the checksum of an IP packet extends the concept to loops dealing with memory arrays of varying sizes. Another application of proof-carrying code is demonstrated in [17], where the safety policy not only captures memory access but also handles resource usage, for example, execution times of agents.

The fundamental principle behind proof-carrying code states that for many computational problems validating a given solution is less complex and thus less costly in terms of computations than creating the solution in the first place. From this perspective, the proof-carrying code concept can be extended beyond its original, narrow definition and be applied to properties of interest other than safety features. Furthermore, a solution does not necessarily have to carry a proof in the literal meaning. For example, in [18] Klohs and Kastens apply the proof-carrying code concept to program analysis in compiler contexts: an optimizing compiler performs deep and complex analyses on the code to establish properties which are used as preconditions for optimizing transformations. Such analyses, for example, data flow analyses, are too time-consuming to be performed by a just-in-time compiler at program execution time. Hence, the producer generates intermediate code and annotates it with results of program analysis, thus proving certain properties of the code. At runtime, the virtual machine takes on the role of the code consumer and checks the annotations with respect to the code. If the check passes, the runtime system uses the annotations for optimizing transformations. In [19], Klohs extends the approach to interprocedural program analysis.

3.2. Proof-Carrying Hardware. We propose proof-carrying hardware as the equivalent of proof-carrying code for dynamically reconfigurable hardware systems. Reconfigurable hardware systems often have limited computational resources, rely on a fast instantiation of newly downloaded hardware modules, and are increasingly security and safety critical. All those characteristics match exactly the characteristics of target systems for which proof-carrying code has been established. There are, however, notable differences between software and hardware. Arguably the most important one is the complexity of the machine model. Software executes on instruction-set processors which allows us to abstract code as a sequence of instructions accessing a rather small set of microarchitectural components. In contrast, reconfigurable hardware modules utilize large numbers of spatially arranged (placed and routed) components. Furthermore, while downloaded code plugs into rather matured software ecosystems, there are no standardized or even commonly used execution environments for reconfigurable hardware modules. Lastly, while security and safety concepts for processor-based systems have been studied for quite some time, the field of reconfigurable hardware security is just emerging.

Figure 1 shows an abstract proof carrying hardware scenario. In this scenario, we denote embedded target platforms that execute functions in software and hardware on dynamically reconfigurable heterogeneous architectures (e.g., CPU/FPGA systems) as *consumers*. The functions are created in form of modules by design centers, denoted as *producers*. In this work we focus on the hardware functions that are requested by consumers, created as reconfigurable modules including security proofs by producers, downloaded over some network, and verified, configured, and executed by consumers—all at runtime. Depending on the use case, there are several reasons for the consumer to request a new hardware module, including bug fixes, using optimized functions and, most importantly, user-initiated extensions of the functionality of the embedded target system. The consumer and potential producers must have agreed on and use a common formalism to describe the functional and perhaps also nonfunctional specification of the requested modules, the characteristics of the target architecture, the safety requirements, and the used proof system.

Computing proofs for all tentative reconfigurable modules, security properties, and target architectures in advance is uneconomic. Along the same line we consider it infeasible for the consumer to store statically verified bitstreams for all tentative reconfigurable modules and security properties.

3.3. Trust and Threat Models. For the proof-carrying hardware scenario, we can establish a general trust and threat model similar to Myagmar et al. [20]: as with proof-carrying code, we neither need to trust the producer of the hardware module nor do we require specially secured transmission. The major difference between proof-carrying code and PCH is that the safety policy for the latter targets hardware instead of software modules. PCH considers hardware reconfigurations as entry points for possible security breaches and attacks. The reconfiguration bitstream could contain a module that provides unspecified, additional, or otherwise undesirable functionality. The cause for this might be an intentional attack or an unintentional alteration of the module's behavior during design or transmission.

The consumer sets up a safety policy that may capture not only the functionality of the module but also non-functional properties, for example, the minimum clock frequency and

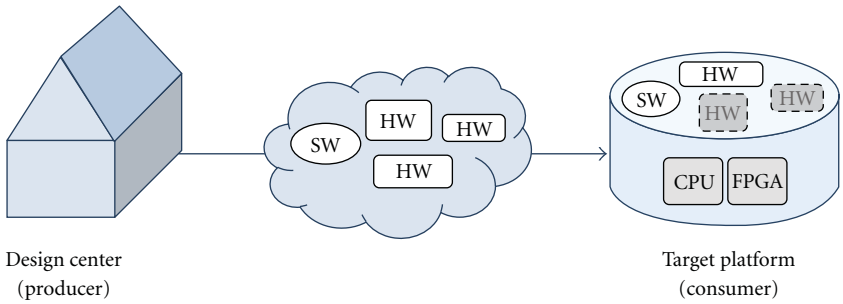


Figure 1: Proof carrying hardware scenario.

area of the module, and the specifics of the reconfigurable platform, for example, the device type and the size and position of the reconfigurable areas. Functional properties can include the complete behavior of the module checked by combinational or sequential equivalence to its specification, or some partial behavior such as ensuring that invariants always hold, accesses to external memory stay within given address bounds, or that bus protocols are adhered to. Any security threat that is described in the safety policy can be dealt with. It is the responsibility of the consumer to ensure that the safety policy is complete in the sense that all security threats to be eliminated for the specific module are covered. Naturally, the possible security threats and thereby results of a malicious hardware module will vary for each host system and use case and might include loss of confidentiality, data theft, data manipulation, or even a complete system failure.

Proof-carrying code and PCH as well do not rely on secure transmission. If the proof check on the consumer side succeeds, it is guaranteed that (i) the proof matches the code and the consumer's safety policy and (ii) the code has the proven property. The check will fail if the proof has been damaged, if it does not fit to the code, or if the code does not establish the required properties. Such a failure may be caused by an accidental or malicious modification of either the code section, or the proof section of the proof-carrying code, or both. In the unlikely case of matching changes to both the proof and the code, there is no damage done since the proof still verifies the code and thus guarantees its compliance with the safety policy. On the other hand, the check will fail if the producer and consumer operate, accidentally or maliciously, with different safety policies. In particular, the proof-carrying code approach is also robust against third parties performing man-in-the-middle attacks, even when circumstances require the safety policy to be public. Since the consumer always knows the original safety policy and uses it to verify the proof, the check would fail if a modified safety policy was used to create the proof. In other words, while it might be possible to pass off a different design, that is, code from a different source, with a matching proof, as soon as the proof check succeeds, no damage is done since any design that complies with the (consumer-stored) safety policy is per definition genuine. The only remaining requirement is a trustworthy procedure for checking the proof.

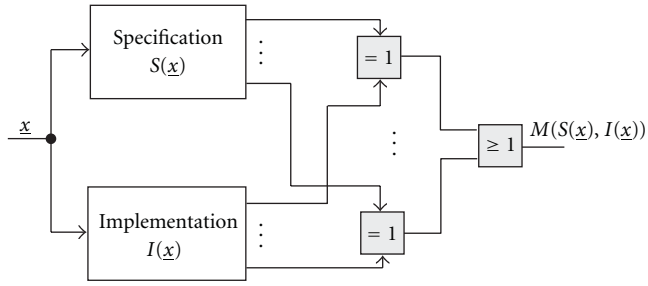


Figure 2: Construction of the miter $M(S(\underline{x}), I(\underline{x}))$.

4. Runtime Combinational Equivalence Checking

Combinational equivalence checking (CEC) is the most fundamental verification problem for hardware. The typical use of CEC is to verify whether a specification of a combinational function, $S(\underline{x})$, is equivalent to an implementation in a specific technology. To that end, the implemented circuit is analyzed and modeled with a logic function, $I(\underline{x})$. Apparently the number of inputs and outputs, respectively, of the specification and the implementation must match. Using $S(\underline{x})$ and $I(\underline{x})$, the miter is formed. The miter is a single-output function that provides both specification and implementation with the same inputs and compares their outputs pairwise with XOR gates. All XOR outputs are then OR-ed together to form the miter, $M(S(\underline{x}), I(\underline{x}))$. Figure 2 shows the construction of the miter graphically.

If under any input $\underline{x}$ the specification and the implementation generate different outputs, the miter will evaluate to 1. Consequently, demonstrating equivalence means to prove the unsatisfiability of the miter. Modern CEC tools represent the miter in conjunctive normal form (cnf) and rely on Boolean satisfiability (SAT) solvers to prove unsatisfiability.

Interestingly, during the last years SAT solvers have progressed into tools that can generate resolution proofs for unsatisfiability. The improvement of such techniques is still a focus of research such as [21–23]. The motivation for this development roots in the desire to build up trust in the results generated by a SAT solver. The direct way would be to prove the correctness of a SAT solver itself, which unfortunately seems to be out of reach given the complexity of modern SAT solvers. The next best approach is to verify

Table 1: Resolution proof trace.

(1)	$x_1 + \overline{x}_2 + \overline{x}_3$	
(2)	$x_1 + x_3$	
(3)	$\overline{x}_1$	
(4)	$x_2 + \overline{x}_3$	
(5)	x_3	Using (2), (3)
(6)	x_2	Using (4), (5)
(7)	$\varnothing$	Using (5), (1), (6), (3)

the unsatisfiability result for each single cnf, a step which requires access to a resolution proof. A resolution proof is a sequence of resolutions on the original cnf and intermediate clauses that eventually leads to an empty clause which models a contradiction. As the size of the generated proof has been a concern, proof traces have been proposed as a compact representation of a proof.

As an example, Table 1 gives a possible proof trace for the cnf $(x_1 + \overline{x}_2 + \overline{x}_3) \cdot (x_1 + x_3) \cdot (\overline{x}_1) \cdot (x_2 + \overline{x}_3)$: the first four lines list the clauses of the cnf. Lines five to seven present the resolution steps and refer to the clauses used to resolve the new terms. In each step, the literal that is used in its negated and in its nonnegated form can be resolved. The resulting clause is the empty clause.

We can make use of resolution proof traces to set up a proof-carrying hardware scenario for runtime (online) CEC. Figure 3 shows the scenario and details the steps producer and consumer perform for runtime CEC. The consumer requests a new reconfigurable hardware module with the combinational function $S(\underline{x})$ and sends the functionality description and the safety policy to a producer. Classically, the producer will run the specification through logic synthesis tools such as FPGA technology mapping and FPGA backend synthesis tools which include place and route and bitstream generation. In the runtime CEC scenario, the producer additionally forms a miter from the specification $S(\underline{x})$ and the implementation $I(\underline{x})$ (synthesized netlist). A CEC tool proves the equivalence and generates the resolution proof trace $P(M(S(\underline{x}), I(\underline{x})))$. Finally, the producer combines the bitstream and the proof trace into the proof-carrying bitstream and sends it to the consumer.

The consumer takes the received bitstream and extracts the implemented logic function $I(\underline{x})$. After forming the miter with this implementation and the original specification, the consumer checks the proof using the proof trace. Only in case the proof holds, the bitstream $B(\underline{x})$ is loaded into a reconfigurable area of the target device.

Any tampering with the bitstream or the proof sections of the proof-carrying bitstream will result in a failed proof check at the consumer side. If both are modified compatibly, the proof check will still fail if $I(\underline{x})$ does not correspond to the original specification $S(\underline{x})$ anymore.

In abstract terms, the safety policy includes the agreement on a specific bitstream format, on cnf to represent combinational functions, and on the use of propositional calculus with its resolution rules to derive and verify the proof. Furthermore, since the resolution proof indexes the clauses of the miter, consumer and producer must use the same way of constructing the miter.

5. Prototype Implementation and Results

To demonstrate the feasibility of runtime CEC (as an application of PCH) we set up the prototype tool flow shown in Figure 4. This proof of concept tool flow is a simplified version of the scenario shown in Figure 3 and serves to validate whether it is possible to shift the verification workload from the consumer to the producer. To this end, it is neither necessary nor the intention of this paper to verify all steps of the production, for example, FPGA backend synthesis tools, to check for correct transmission or to completely implement the consumer's functions. Specifically, the consumer specifies the combinational test function in behavioral Verilog for simplicity. In principle, any other and perhaps simpler specification formalism for capturing combinational functions can be used.

Our prototype tool flow bases on a number of freely available, noncommercial CAD tools which gives us full control over the different parameters involved and file formats used. After receiving the specification, the producer invokes the following tools.

(i) Odin [24, 25] performs front-end synthesis and translates the incoming circuit specification in Verilog into a logic description in blif (Berkeley Logic Interchange Format).

(ii) ABC [26] performs logic optimization and technology mapping to 4-LUTs, generating again a blif file. The optimization is based on And-Inverter Graphs and includes balancing, refactoring and rewriting; all applied multiple times to the circuit with the resync2 command.

(iii) T-VPack [27, 28] packs the LUTs into clustered logic blocks, generating a circuit description in form of a netlist (.net) of such logic blocks. The experiments described in this paper use nonclustered logic blocks and LUTs with four inputs, which are the default values. We run T-VPack to convert the LUT netlist into the format required by VPR and omit further logic specification.

(iv) VPR [27, 28] places and routes the packed logic block netlist and produces placement (.p) and routing (.r) information for inclusion in the final bitstream. The target architecture input file used by VPR has to agree with the specifications used for T-VPack. We use the k4-n1.xml architecture file provided by VPR. We operate VPR in the area minimization mode.

(v) To form the miter cnf, we use again ABC which translates the cnf from And-Inverter Graphs (AIGs).

(vi) The SAT solver PicoSAT [29] at the producer's side proves the unsatisfiability of the miter and generates an extended proof trace file.

(vii) ComPose (own code) is for composing the proof-carrying bitstream.

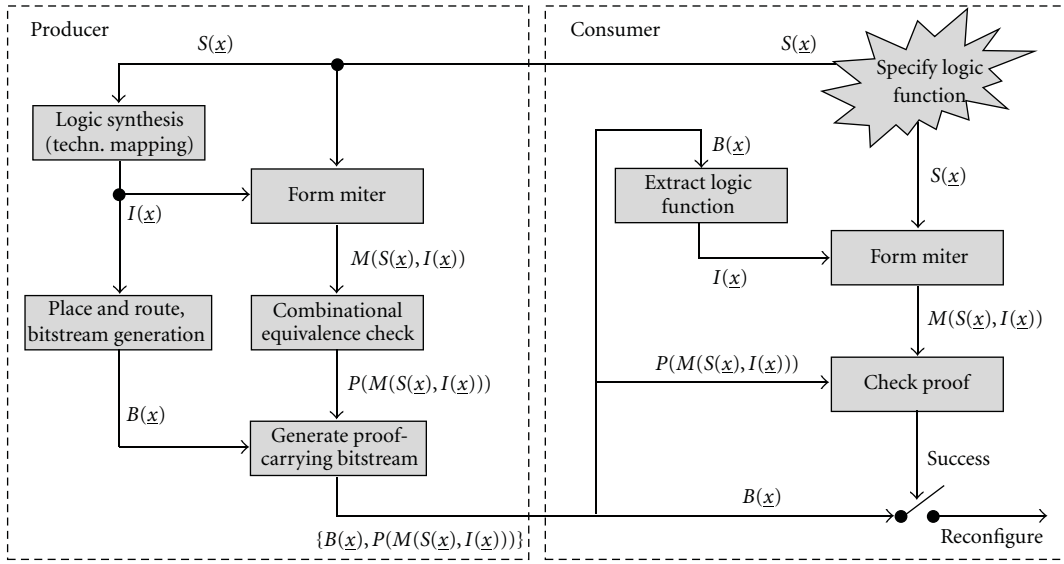


Figure 3: Runtime combinational equivalence checking as a proof of concept study for the proof-carrying hardware concept.

Table 2: Runtime measurements for producer and consumer in the online CEC scenario.

Test function	Producer [s]					Consumer [s]			
	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose	DeComPose	ABC′	TraceCheck
Converter	0.187	0.348	0.005	2.040	0.008	0.080	0.004	0.148	0.013
128-bit parity	0.194	0.350	0.100	10.892	0.036	0.111	0.007	0.156	0.027
8-bit add/sub	0.183	0.273	0.007	2.126	0.015	0.034	0.004	0.201	0.010
16-bit add/sub	0.122	0.279	0.011	5.520	0.048	0.074	0.004	0.206	0.035
32-bit add/sub	0.186	0.330	0.012	14.428	0.099	0.154	0.009	0.215	0.059
64-bit add/sub	0.195	0.432	0.220	40.674	0.244	0.327	0.015	0.234	0.126
6-bit multiplier	0.179	0.332	0.008	5.304	0.540	0.145	0.009	0.234	0.483
8-bit multiplier	0.184	0.450	0.010	13.205	15.337	1.148	0.123	0.213	11.179
10-bit multiplier	0.183	0.645	0.014	26.589	256.119	18.849	1.807	0.214	190.630
16-bit multiplier	0.205	1.473	0.040	133.814	3732.031*				
32-bit multiplier	0.229	6.236	0.163	2116.210	3895.797*				
64-bit multiplier	0.527	27.424	0.726	36447.768	6387.947*				

The proof-carrying bitstream is then sent to the consumer that executes the following steps:

(i) DeComPose (own code) for decomposing the proof-carrying bitstream,

(ii) ABC to form the miter,

(iii) TraceCheck [30] to validate the correctness of the proof trace and thereby give the final approval to load and execute the received bitstream; the output is a compact binary resolution trace.

Our simplified prototype tool flow shown in Figure 4 covers the main components and functionalities of the runtime CEC scenario but is incomplete since the consumer does not yet use the original specification to form the miter. Thus, the safety policy in our experiments is the demand for combinational equivalence of the function before and after the logic synthesis/optimization and technology mapping. This is sufficient for the experiments presented in this paper. We use ABC at the consumer side to form the miter from both the unprocessed and the technology mapped and optimized circuit which are embedded in the bitstream. The resulting cnf is then checked for consistency with the cnf in the resolution proof trace. If both cnf forms match and the proof turns out to be correct, the module can be loaded. Completing the tool flow by taking the original specification to form the miter and experimenting with more advanced safety policies is part of future work.

We test the prototype tool flow on an Intel Core 2 Duo 2 GHZ CPU with 4 GB RAM running Linux 2.6.31.5-0.1 and report on results using the following test functions: an EBCDIC to ASCII converter for the letter subset of the EBCDIC code, a 128-bit parity function, n-bit combined adders/subtractors with $n = 8, 16, 32$, and 64, respectively, and n-bit unsigned multipliers with $n = 6, 8, 10, 16, 32$, and 64. We have chosen these test functions and their

according input size variations to contrast functions which are presumably rather easy to verify with more demanding ones, that is, the multipliers. The resulting FPGA bitstreams implement all test functions with LUTs; special circuitry such as carry chains or heterogeneous blocks are not yet included in our FPGA model.

Using the prototype tool flow and the test functions we experimentally investigate the following questions.

(1) Does the runtime CEC tool flow work correctly?

(2) What are the runtimes for producer and consumer? The difference between the time required for proving the miter unsatisfiable and generating the proof trace versus the time required to check the proof is of special interest. This time difference constitutes the main savings for the consumer, in comparison to an approach where the consumer runs the complete verification tool chain.

(3) What are the memory usages for producer and consumer? Similar to the runtimes, the savings for the consumer are of special interest.

(4) How large is the overhead for the proof-carrying bitstream? The proof trace increases the size of the bitstream and leads to higher transmission costs.

As a first result, we are able to demonstrate the correct functionality of the runtime CEC prototype. The producer generates FPGA implementations and equivalence proofs for all test functions, which the consumer successfully verifies. Tampering with the files at different stages of the producer tool flow results in the expected effects: modifying the technology-mapped netlist of the miter cnf or the miter cnf itself results in a satisfiable miter, stating that the implemented circuit differs from the specification. Changing the proof trace at the consumer side or during transmission leads to a failed proof check at the consumer.

An important metric for runtime CEC is the required computation time, especially for the consumer. We measure the runtimes of all tools in our prototype. On the producer side, this includes Odin and ABC for creating the blif files as well as the miter cnf, T-VPack and VPR for packing, place and route, PicoSAT for equivalence checking and proof trace generation, and ComPose for compiling the proof-carrying bitstream (consisting of the two blif files, the packed netlist, the placement and routing information, and the resolution proof trace). On the consumer side, we measure the runtime of DeComPose, ABC only building the miter without performing logic optimization, and TraceCheck.

Table 2 presents the runtime measurements for our test functions. Columns two to seven give the computation time for the producer side of the tool flow while columns eight to ten depict the runtimes for the consumer. The table clearly shows that the multiplier test functions form a separate group. First, with growing number of inputs multipliers synthesized from LUTs become huge functions which is demonstrated by the high runtimes for VPR. Second, SAT solvers have difficulties in proving the unsatisfiability of multiplier miters which is reflected by the PicoSAT runtimes. In fact, in our experiments PicoSAT aborted the computation due to a lack of memory for multipliers with $n = 16$ and higher. Consequently, we could not conduct measurements for the consumer side of the tool flow for these functions. We mark these cases with an asterisk in Table 2.

Table 3: Runtimes comparison between consumer and producer.

Test function	Consumer	Producer	
	Total [s]	Total [s]	workload
Converter	0.165	2.668	93%
128-bit parity	0.190	11.683	98%
8-bit add/sub	0.215	2.638	92%
16-bit add/sub	0.245	6.054	96%
32-bit add/sub	0.283	15.209	98%
64-bit add/sub	0.375	42.092	99%
6-bit multiplier	0.726	6.508	89%
8-bit multiplier	11.515	30.334	72%
10-bit multiplier	192.651	302.339	61%

The main observation from Table 2 is the difference in time effort between producer and consumer. DeComPose is less costly than ComPose, so is ABC′ in comparison to ABC with the gap widening for the more complex test functions. Importantly, with one exception there is a notable difference between PicoSAT and TraceCheck runtimes, even if not as pronounced as expected. This might be due to the fact that on one hand unsatisfiability is easily proven for the miters built from our simple test functions, and on the other hand the generated netlists are quite large which means that all tools processing netlists spend substantial time in file I/O. For the more complex test functions the SAT solver dominates the producer's overall runtime, an effect that can be seen in going from the 8-bit to the 10-bit multiplier.

With Table 3, we give an overview of the total runtimes for both the consumer and producer side. The table also reports the producer's percentage of the total workload consisting of both the consumer's and producer's runtime. The data shows that we succeeded in our plan to shift the majority of the workload from the consumer platform to an external resource.

To further underline this key point, we conducted runtime measurements with cnf problems from the 2008 SAT-Race_TS_1 benchmark suite. Table 4 compares the runtimes of the SAT solver (PicoSAT) and the proof checker (TraceCheck) for a number of benchmark problems. The cnfs differ greatly in the number of variables and clauses but include instances with up to six orders of magnitude more variables and clauses (`narai_vpn-10 s`) than the miters for our test functions. The last column of Table 4 shows that the effort for checking a proof trace is between one and three (five in an extreme case) orders of magnitude lower than for proving equivalence and generating the proof trace.

Table 5 gives the results for the memory usage for each tool and test function. We measured the peak memory usage with Valgrind [31], that is, the massif tool, and included stacks as well as heaps in the measurements. The first set of columns displays the memory usage on producer side, and columns eight to ten show the measurement results on

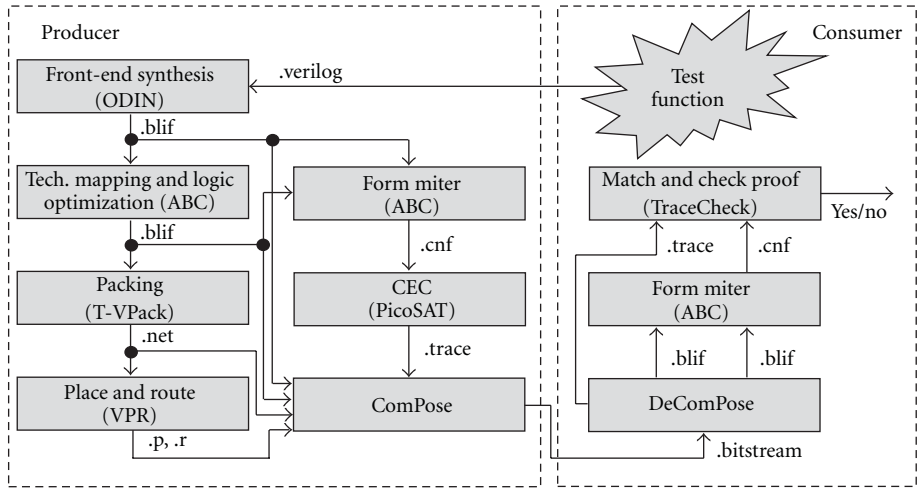


Figure 4: Runtime cec prototype tool flow.

Table 4: Runtime comparison between PicoSAT (SAT) and TraceCheck (Check) for benchmarks of the 2008 SAT-Race_TS_1.

cnf instance	Size of cnf [Vars × Clauses]	SAT [s]	Check [s]	Factor
een-tip-sr06-par1	163647 × 484827	6.412	0.024	267
een-tipb-sr06-tc6b	40196 × 115775	3.028	0.012	252
godlb-heqc-desmul	28902 × 179895	75.697	1.512	50
goldb-heqc-rotmul	5980 × 35229	31.458	2.272	13
hooons-vbmc-s04-05	8503 × 25097	11.065	1.536	7
hooons-vbmc-s04-07	25900 × 77627	158.294	8.557	18
manol-pipe-c10b	43517 × 129265	87.937	1.075	81
manol-pipe-c10ni_s	204664 × 609478	255.584	0.004	63896
manol-pipe-c6id	82022 × 242044	5.460	0.052	105
manol-pipe-c6n	37147 × 110077	49.547	0.820	60
manol-pipe-c6nid_s	148051 × 438562	5.528	0.020	276
manol-pipe-c7_i	13023 × 38509	17.685	0.292	60
manol-pipe-c7idw	112620 × 333058	131.444	1.412	93
manol-pipe-c8_i	14052 × 41596	74.433	2.028	36
manol-pipe-c8b_i	32057 × 95005	13.485	0.256	52
manol-pipe-c8n	53697 × 159595	111.351	1.988	56
manol-pipe-f6b	37002 × 109570	6.672	0.328	20
manol-pipe-f6n	37452 × 110920	7.500	0.236	31
manol-pipe-g10idw	174122 × 516784	141.241	1.964	71
manol-pipe-g6bid	40371 × 118192	5.272	0.112	47
manol-pipe-g7n	23936 × 70492	5.784	0.248	23
narai_vpn-10s	2270930 × 8901946	306.335	0.368	832
schup-l2s-s04-abp4	14809 × 48429	67.528	7.896	8
velev-npe-1.0-02	3295 × 35407	23.577	3.748	6
velev-sss-1.0	1453 × 12526	20.741	3.744	5

consumer side. As in Table 2, the numbers marked with an asterisk represent a minimum value, measured before the tool aborted the computation as it ran out of memory.

We note that PicoSAT and TraceCheck, followed by VPR and ABC, are the most memory consuming tools on the producer's and consumer's side, respectively. Formal verification is therefore among the most memory demanding tools in the scenario. The larger multiplier test functions with 16-bit inputs or more caused PicoSAT to run out of memory. Comparing TraceCheck and PicoSAT, the consumer has to invest slightly more in memory than the producer. As opposed to the runtimes where the producer is able to

Table 5: Memory usage measurements for producer and consumer in the online CEC scenario.

Test function	Producer					Consumer			
	Odin	ABC	T-VPack	VPR	PicoSAT	ComPose	DeComPose	ABC′	TraceCheck
Converter	0.997 MiB	13.79 MiB	239.6 KiB	1.207 MiB	69.42 KiB	62.17 KiB	62.12 KiB	10.06 MiB	81.00 KiB
128-bit parity	935.8 KiB	13.60 MiB	310.1 KiB	3.465 MiB	189.2 KiB	61.78 KiB	61.73 KiB	9.937 MiB	181.1 KiB
8-bit add/sub	711.4 KiB	13.90 MiB	243.5 KiB	1.218 MiB	87.71 KiB	61.62 KiB	61.58 KiB	10.54 MiB	101.1 KiB
16-bit add/sub	771.4 KiB	14.05 MiB	267.4 KiB	2.306 MiB	238.8 KiB	61.62 KiB	61.58 KiB	10.70 MiB	268.4 KiB
32-bit add/sub	899.6 KiB	14.27 MiB	315.4 KiB	4.480 MiB	488.5 KiB	61.62 KiB	61.58 KiB	10.98 MiB	485.9 KiB
64-bit add/sub	1.130 MiB	14.58 MiB	533.8 KiB	8.341 MiB	1.010 MiB	61.62 KiB	61.58 KiB	11.69 MiB	1.038 KiB
6-bit multiplier	774.6 KiB	13.93 MiB	263.6 KiB	2.298 MiB	1.380 MiB	62.70 KiB	62.66 KiB	10.38 MiB	2.167 MiB
8-bit multiplier	803.9 KiB	13.95 MiB	301.1 KiB	4.086 MiB	25.74 MiB	64.28 KiB	64.23 KiB	10.57 MiB	43.97 MiB
10-bit multiplier	887.3 KiB	14.10 MiB	391.9 KiB	6.425 MiB	374.8 MiB	71.20 KiB	71.15 KiB	10.87 MiB	652.8 MiB
16-bit multiplier	1.201 MiB	14.42 MiB	937.0 KiB	16.46 MiB	2.477 GiB*				
32-bit multiplier	2.750 MiB	19.83 MiB	3.590 MiB	67.33 MiB	2.742 GiB*				
64-bit multiplier	8.858 MiB	48.98 MiB	14.41 MiB	271.0 MiB	2.699 GiB*				

Table 6: Size measurements for the bitstream and proof trace.

Test function	Bitstream size [KiB]	Proof trace size [KiB]	Overhead [%]
Converter	120.2	23.2	19.35
128-bit parity	309.8	61.2	19.78
8-bit add/sub	99.4	32.8	32.98
16-bit add/sub	246.9	103.9	42.10
32-bit add/sub	490.6	194.3	39.60
64-bit add/sub	1077.9	464.9	43.13
6-bit multiplier	1657.6	1519.0	91.64
8-bit multiplier	40803.3	40538.6	99.35
10-bit multiplier	711487.3	711052.8	99.93

burden a major part of the total workload, the memory resource requirement is still considerable for the consumer in our current prototype tool flow.

We further provide an estimate on the overhead incurred by adding a proof trace to the delivered bitstream. The accuracy of the estimate is limited due to two facts. First, we do not synthesize for an FPGA of given size but operate VPR in the area minimization mode. For each circuit, VPR chooses a logic block array just large enough to accommodate the circuit and a channel width just sufficient to route the circuit. Consequently, our overheads are relative to the circuit size which results in a slightly pessimistic estimate. Second, our tool flow does not yet specify a binary bitstream format as ComPose merely forms a concatenation of several text files. Hence, we measure the size of the prototype bitstream which is composed of the corresponding text files, being a text file itself. Table 6 lists for each test function the size of the bitstreams as created by ComPose, the proof trace as calculated by PicoSAT, and the corresponding overhead, that is, the percentage of the bitstream made up by the resolution proof trace. The results show a wide variation in overheads from the rather low 19% for the EBCDIC-ASCII converter to 99% for the 10-bit multiplier. Although these numbers are overly pessimistic and coarse estimates, they point to the need of investigating a binary format for the proof-carrying bitstream using compression techniques.

6. Conclusion and Future Work

In this paper, we present proof-carrying hardware (PCH) as a novel approach to reconfigurable system security. We describe the main concept of PCH and detail its advantage over known reconfigurable hardware security approaches. Then, we apply the PCH concept to runtime combinational equivalence checking (CEC). CEC guarantees the adherence to functional specifications of a module which eliminates a major security risk for the consumer when loading reconfigurable modules at runtime. We elaborate on a prototype tool flow for runtime CEC, show its implementation, and discuss experimental results. Using proof traces generated by a modern SAT solver, the prototype clearly indicates a significant shift of computational cost and memory usage from the reconfigurable consumer platform to the producer of a reconfigurable hardware module.

While the experiments in this paper already demonstrate the high potential of the PCH approach, we will complete the tool flow shown in Figure 3 as a next step. We will extend the tool flow to formally verify all production steps and finalize all required tools on the consumer side. Future work also includes the definition of a binary proof-carrying bitstream format and the investigation of compression techniques, which will allow us to demonstrate a complete PCH framework.

Furthermore, we are interested in applying PCH to safety properties other than combinational equivalence. The range of safety policies to be investigated includes the following.

(i) First is the sequential equivalence of modules based on ABC which is capable of building miter functions not only for combinational but also for sequential circuits using and-inverter graphs as described in [32]. Alternatively, we intend to look into the VIS tool that uses BDDs to perform sequential equivalence checks; see [33].

(ii) Second is the absence of short circuits which is another functional property. As [34] has shown the partial reconfiguration process can result in short-term, middle-term, or long-term short circuits damaging the system in different ways.

(iii) Third is the physical (structural) isolation through primitives such as moats and drawbridges introduced in [11]. Here, we plan to extend VPR to enforce such primitives during place and route and annotate the bitstream accordingly to allow the consumer to efficiently verify isolation.

(iv) Fourth are the other nonfunctional properties such as specific timing and routing constraints which might be more efficiently guaranteed when the producer annotates the bitstream properly.

This overview indicates the variety of safety concerns and possible safety policies to which the PCH concept might be applicable. It is important to note that the PCH concept is broad enough to include different methods for verifying safety properties, ranging from classic formal verification techniques to checking annotated bitstreams. The main characteristics of PCH is the shift of work from the consumer to the producer which enables embedded target systems with limited resources to obtain security guarantees at runtime.

Acknowledgment

This work is partially funded by the International Graduate School Dynamic Intelligent Systems, University of Paderborn, Germany.

References

[1] G. Necula and P. Lee, "Proof-carrying code," Tech. Rep. 15213, School of ComputerScience, Carnegie Mellon University, Pittsburgh, Pa, USA, CMU-CS-96-165, November 1996.

[2] R. Kastner and T. Huffmire, "Threats and challenges in reconfigurable hardware security," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '08)*, pp. 334–345, CSREA Press, July 2008.

[3] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: towards runtime verification of reconfigurable modules," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 189–194, IEEE, Cancun, Mexico, December 2009.

[4] S. Drimer, "Volatile FPGA design security—a survey," December 2007.

[5] T. Huffmire, C. Irvine, T. D. Nguyen, T. Levin, R. Kastner, and T. Sherwood, *Handbook of FPGA Design Security*, Springer, New York, NY, USA, 1st edition, 2010.

[6] R. Chaves, G. Kuzmanov, and L. Sousa, "On-the-fly attestation of reconfigurable hardware," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 71–76, IEEE, September 2008.

[7] S. Drimer and M. Kuhn, "A protocol for secure remote updates of FPGA configurations," in *Reconfigurable Computing: Architectures,Tools and Applications*, vol. 5453, pp. 50–61, Springer, New York, NY, USA, 2009.

[8] S. Drimer, "Authentication of FPGA bitstreams: why and how," in *Reconfigurable Computing: Architectures,Tools and Applications*, vol. 4419, pp. 73–84, Springer, New York, NY, USA, 2007.

[9] B. Badrignans, R. Elbaz, and L. Torres, "Secure FPGA configuration architecture preventing system downgrade," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 317–322, IEEE, September 2008.

[10] T. Huffmire, S. Prasad, T. Sherwood, and R. Kastner, "Policy-driven memory protection for reconfigurable hardware," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS '06)*, vol. 4189 of *LNCS*, pp. 461–478, Springer, September 2006.

[11] T. Huffmire, B. Brotherton, G. Wang et al., "Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems," in *Proceedings of the Symposium on Security and Privacy*, pp. 281–295, IEEE, Oakland, Calif, USA, May 2007.

[12] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin, "Enforcing memory policy specifications in reconfigurable hardware," *Computers and Security*, vol. 27, no. 5-6, pp. 197–215, 2008.

[13] T. Huffmire, B. Brotherton, N. Callegari et al., "Designing secure systems on reconfigurable hardware," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–24, 2008.

[14] T. Huffmire, B. Brotherton, T. Sherwood et al., "Managing security in FPGA-based embedded systems," *IEEE Design and Test of Computers*, vol. 25, no. 6, pp. 590–598, 2008.

[15] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, Article ID 4544862, pp. 1165–1178, 2008.

[16] S. Singh and C. J. Lillieroth, "Formal verification of reconfigurable cores," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 25–33, IEEE, April 1999.

[17] N. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Agents and Security*, vol. 1419 of *LNCS*, pp. 61–91, Springer, New York, NY, USA, 1998.

[18] K. Klohs and U. Kastens, "Memory requirements of java bytecode verification on limited devices," in *Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification(COCV '04)*, vol. 132, 2004.

[19] K. Klohs, "A summary function model for the validation of interprocedural analysis results," in *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification (COCV '08)*, 2008.

[20] S. Myagmar, A. Lee, and W. Yurcik, "Threat modeling as a basis for security requirements," in *Proceedings of the IEEE Symposium on Requirements Engineering for Information Security (SREIS '05)*, August 2005.

[21] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proceedings of the Design Automation Conference (DAC '06)*, pp. 532–535, ACM, July 2006.

[22] S. Chatterjee, A. Mishchenko, R. Brayton, and A. Kuehlmann, "On resolution proofs for combinational equivalence," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 600–605, June 2007.

[23] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proceedings of the IEEE/ACM International Conference on*

Computer-Aided Design (ICCAD '06), pp. 836–843, ACM, New York, NY, USA, 2006.

[24] "Odin A Verilog RTL Synthesis Tool for Heterogeneous FPGAs," http://www.eecg.toronto.edu/~jayar/software/odin/index.html.

[25] P. Jamieson and J. Rose, "A verilog RTL synthesis tool for heterogeneous FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 305–310, IEEE, Tampere, Finland, August 2005.

[26] "Going Places with ABC, Berkeley Logic Synthesis and Verification Group," http://www.eecs.berkeley.edu/~alanmi/abc/abc%20tutorial.ppt.

[27] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *Proceedings of the nternational Conference on Field Programmable Logic and Applications (FPL '97)*, vol. 1304, pp. 213–222, Springer, London, UK, 1997.

[28] V. Betz, "VPR and T-VPack User's Manual," (Version 5.0), 2008.

[29] A. Biere, "PicoSAT essentials," *Journal on Satisfiability , Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[30] C. Sinz and A. Biere, "Extended resolution proofs for conjoining BDDs," in *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, vol. 3967 of *LNCS*, pp. 600–611, Springer, 2006.

[31] J. Seward, N. Nethercote, and T. Hughes, "Valgrind Documentation," August 2009, http://valgrind.org/.

[32] R. Brayton and A. Mishchenko, "Scalably-verifiable sequential synthesis," ERl Technical Report, EECS Dept. UC Berkeley, 2007.

[33] T. V. Group, "Vis: a system for verification and synthesis," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, R. Alur and T. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, pp. 428–432, Springer, July 1996.

[34] D. Beckhoff, C. Koch, and J. Torresen, "Short-circuits on fpgas caused by partial runtime reconfiguration," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 596–601, IEEE, August 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 480208, 17 pages
doi:10.1155/2010/480208

Research Article

Robotic Mapping and Localization with Real-Time Dense Stereo on Reconfigurable Hardware

John Kalomiros[1] and John Lygouras[2]

[1] *Department of Informatics and Communications, Technological Educational Institute of Serres, Terma Magnisias, 62124 Serres, Greece*
[2] *Section of Electronics and Information Systems Technology, Department of Electrical Engineering & Computer Engineering, School of Engineering, Democritus University of Thrace, 67100 Xanthi, Greece*

Correspondence should be addressed to John Kalomiros, ikalom@teiser.gr

Received 1 March 2010; Revised 20 July 2010; Accepted 20 November 2010

Academic Editor: Viktor K. Prasanna

A reconfigurable architecture for dense stereo is presented as an observation framework for a real-time implementation of the simultaneous localization and mapping problem in robotics. The reconfigurable sensor detects point features from stereo image pairs to use at the measurement update stage of the procedure. The main hardware blocks are a dense depth stereo accelerator, a left and right image corner detector, and a stage performing left-right consistency check. For the stereo-processor stage, we have implemented and tested a global-matching component based on a maximum-likelihood dynamic programming technique. The system includes a Nios II processor for data control and a USB 2.0 interface for host communication. Remote control is used to guide a vehicle equipped with a stereo head in an indoor environment. The FastSLAM Bayesian algorithm is applied in order to track and update observations and the robot path in real time. The system is assessed using real scene depth detection and public reference data sets. The paper also reports resource usage and a comparison of mapping and localization results with ground truth.

1. Introduction

Vision-based robotic localization and mapping is one of the most active research areas in mobile robotics. Next generation intelligent vehicles technology depends heavily on the successful implementation of vision-based systems for navigation, real-time obstacle detection, and path planning [1]. In particular, the simultaneous localization and mapping (SLAM) problem has been given immense attention in recent years, since it holds the key to robust navigation in unknown environments. SLAM is a class of stochastic algorithms that address the problem of estimating concurrently the robot's path and a map of the surrounding environment [2]. These algorithms are based on a nonlinear model for robot dynamics and on a landmark observation model derived from the particular nature of the sensor used for data gathering.

Feature extraction is a prerequisite for robot mapping and localization, which in turn is a central problem for the navigation of autonomous vehicles [3]. Active sensors like sonars and laser range finders are commonly used for the purpose of feature extraction for localization and mapping [4]. They are effective in static, low density, and low noise environments, but are also expensive, heavy, and prone to environmental interference. Passive systems, like vision systems, are much less sensitive to environmental interference. Recently attention has been given to monocular or stereo-vision systems, and methods have been proposed to extract reliable features from image data [5]. Various vision-based features have been used in the literature. Such are Shi-Tomasi features [6], SIFT descriptors [7], edge segments [8], Harris corners [9], and so forth. Visual feature extraction and tracking in real-time is computationally demanding, particularly since the data rates coming from camera are much higher than those from other sensors.

Stereo-vision provides a solid framework for the extraction of 3D structure from image data. Finding the correspondences between left and right image frames allows depth computation for scene points. Solving the correspondence problem is not trivial, however, since occlusion, specularities

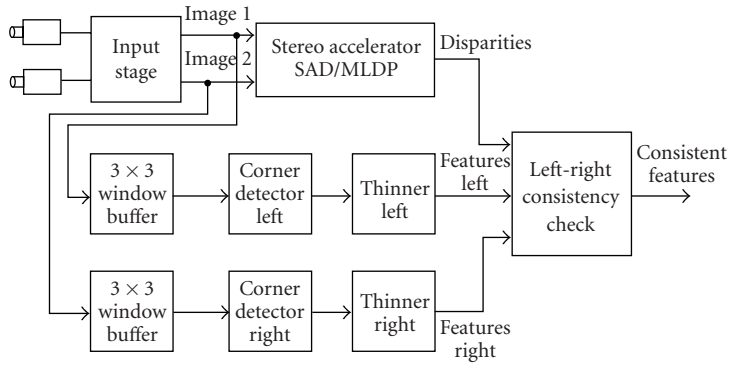


Figure 1: Block diagram of the point-feature detector.

or lack of texture can lead to wrong matches. The derivation of dense disparity maps from stereo-pairs is again a very demanding computational problem, since it requires extensive searching and optimization along scanlines [11]. A taxonomy and evaluation of different correspondence methods is given in [12]. Cost aggregation methods for real-time stereo matching are evaluated in [13].

Vision-based feature extraction for robotic mapping and localization is a very intensive computational procedure and is known as the main source of delay in visual SLAM. Moreover, stereo-assisted feature extraction has high-computational demands, especially for global stereo matching. Field programmable gate arrays represent a flexible and efficient solution for accelerating stereo matching computations and other complex image processing tasks. Their fine-grain structure of small logic elements allows parallelism combined with high processing speeds. They also present an advantage over Application Specific Integrated Circuits (ASICs) because they are reprogrammable and much cheaper for prototyping.

In this paper, a real-time point-feature extraction technique based on stereo vision is proposed and is implemented in reconfigurable hardware. Left and right image information is input to the system as a stream of 8-bit gray-scale pixels and is processed by several hardware stages integrated in a system-on-a-programmable-chip. The first stage is a stereo-processor able to produce dense depth in the form of 8-bit disparities in parallel with input data. A global matching maximum-likelihood algorithm is implemented and its suitability for consistent feature extraction is tested. The algorithm provides optimized disparity computation across scanlines and addresses the problem of depth extraction at occlusion boundaries but has considerable computational cost. Other stages are left and right image corner detectors and thinning stages resulting in a pair of binary images with well-localized point features. The final stage performs a left-right consistency check, taking into account the disparity values produced by the stereo stage. Point-features surviving the check are output from the final stage. The integrated system features also a Nios II processor for data control, an external memory interface, DMA functions, and a USB 2.0 controller for communication with a host computer. An introductory description of this implementation was given in [14] and here the system is evaluated and tested extensively.

The above system is adapted to a simple mobile vehicle and is used as the sensor part in a simultaneous localization and mapping experiment. On the host part, an application receives the stream of feature positions with their respective disparities. A FastSLAM algorithm is used to track and optimize the vehicle path and the map feature locations.

The main contribution of this paper is twofold. It presents an original integrated sensor producing landmark measurements, based on computationally demanding dense stereo techniques. Second, the system is tested and assessed in terms of accuracy and real-time performance in a state-of-the-art Bayesian algorithm for simultaneous localization and mapping. The presented system can be used as a framework to develop more sophisticated dense depth solutions to vision-based robotic navigation.

The rest of the paper is organized as follows. In Section 2, the overall system is outlined in a block diagram. In Section 3 the stereo processing stage, the corner detection and the stage for left-right consistency check are presented. In Section 4 the system is evaluated in terms of frame rates and hardware resource usage. In Section 5, the presented hardware is used as the measurement stage for a 3D FastSLAM localization and mapping experiment. Section 6 compares the results with similar efforts presented in the literature and Section 7 concludes the paper.

2. The Overall Feature Detection System

Figure 1 shows a block diagram of the overall system. A stereo head is developed in the laboratory with two parallel cameras carefully adjusted in order to produce rectified image pairs. First, a dense depth map is produced by the stereo accelerator stage. Left and right image pixel streams are processed in parallel by the stereo processor. The stereo processor finds the correspondences between left and right image pixels and produces the disparities $d = u_L - u_R$, where u_L and u_R are pixel coordinates on a scanline. It can produce 16, 32, 49, or even more levels of disparity, based on the same principles but making use of additional hardware resources. The stereo stage can implement any method of local correlation or global scanline optimization provided

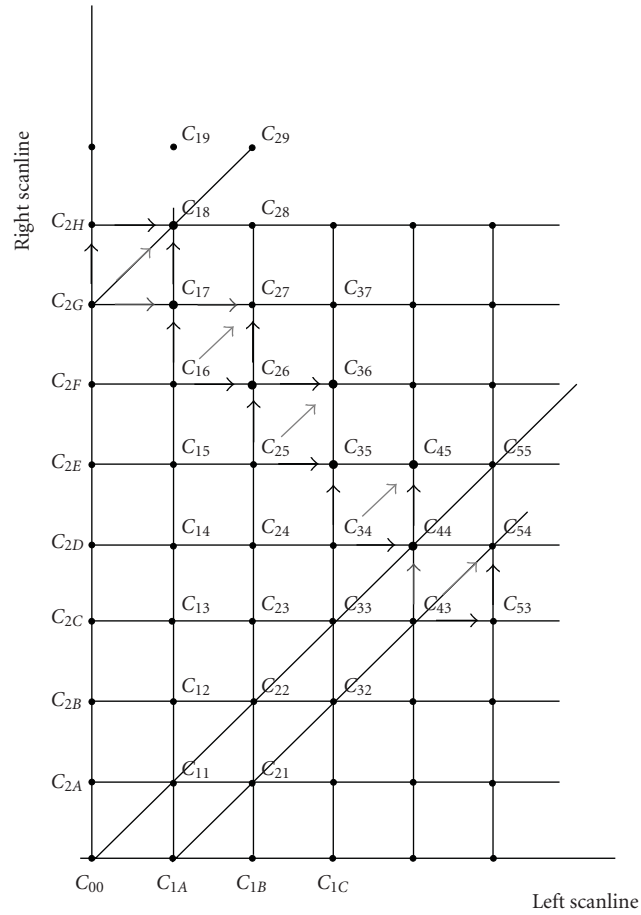


Figure 2: Cost-plane matrix and a diagonal slice for parallel computations.

that they produce a synchronized output in parallel with the input data streams. In our stereo-assisted feature extraction scheme the stereo correspondence accelerator implements a demanding maximum-likelihood optimization algorithm based on dynamic programming (DP).

In the feature extraction stage, corner detectors implement a simple edge detection principle based on convolution with 3×3 Prewitt horizontal and vertical masks. In principle, any edge detection method is applicable; however the Prewitt masks are hardware friendly since they perform convolution computations only with adders and subtractors. The same 3×3 window used for correlation in the stereo stage can be used for the purpose of convolution in the corner detection stage. A thresholding scheme is applied in order to produce binary horizontal and vertical edge images from each input image. Corners are produced by simply performing a binary AND operation between horizontal and vertical edges. Thinning stages apply a thinning algorithm based again on convolution with Prewitt masks and result in a well-defined point-feature at each image corner. In the final stage the consistency of point-features in the left and right image frame is tested by comparing their horizontal displacement with the disparity values produced by the stereo stage. Only features surviving the test are transmitted to the host application and are processed in the 3D map reconstruction phase.

3. Analysis and Design of the Embedded System Stages

3.1. Hardware Design of the DP Stereo Accelerator. Dense depth is extracted by matching left and right rectified image pairs captured by the stereo head. In this paper a semiglobal matching algorithm is used, based on a dynamic programming optimization method. The algorithm is computationally demanding and difficult to implement in real-time without acceleration. As pointed out by Cox et al. [15] and Brown et al. [11] the computational complexity of an algorithm matching all pixels in a stereo pair using dynamic programming is $O(N^2 \times M)$, where N, M are the horizontal and vertical image dimensions.

Dynamic programming for stereo is mathematically and computationally more complex than local correlation methods, since stereo correspondence is derived as a globally optimum solution for the whole scan line [16]. The algorithm used in this paper is a method for maximizing likelihood, which is equivalent to minimizing a cost function [15]. The

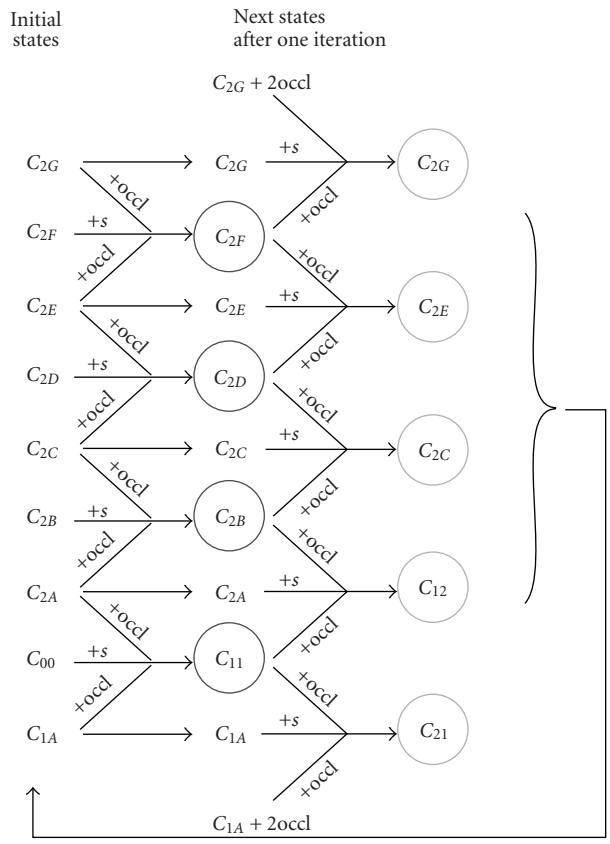


Figure 3: Successive cost-state computation with feedback, using an example nine-state machine, for the cost-plane of Figure 2.

algorithm is developed in two phases, namely, the cost-plane building phase and the backtracking phase. The cost-plane is computed as a two-dimensional matrix of minimum costs, one cost-value for each possible correspondence $I_i \leftrightarrow I_j$ between left and right image intensity values, along a scan line. One always proceeds from left to right, as a result of the ordering constraint. This procedure is shown in Figure 2, where each point of the two-dimensional cost function is derived as the minimum transition cost from the three neighboring cost values. Transition costs result from previous costs, adding a matching or occlusion cost, s_{ij} or occl, according to the following recursive procedure:

$$C(i,j) = \min\Big\{C(i-1,j-1) + s_{ij},\ C(i-1,j) + \text{occl} \\ C(i,j-1) + \text{occl}\Big\}. \tag{1}$$

In the above equations, the matching cost s_{ij} is minimized when there is a match between left and right intensities. We used the following dissimilarity measure:

$$s_{ij} = \frac{\left(I_l(i) - I_r(j)\right)^2}{\sigma^2}, \tag{2}$$

where σ represents the standard deviation of pixel noise. Typical values are $\sigma = 0.05 - 0.12$ for image intensities in the range $[0,\ 1]$. In our implementation, we calculate s_{ij} within a 3×3 window applied in both images. The occlusion cost occl is the cost of pixel j in the right scanline being skipped in the search for a matching pixel for i and in our tests takes a value occl = 0.2.

The cost-matrix computation is a recursive procedure in the sense that for each new cost $C(i, j)$ the preceding costs on the same raw and column are needed, according to (1). In turn, previous costs need their precedent costs, rendering the parallel computation of costs intractable. In order to parallelize the cost-matrix computation in hardware, we design a variation of the DP algorithm, using an adequate slice of the cost-matrix above the diagonal of the cost plane, as shown in Figure 2. We take into account only the part of the cost plane, where the minimum cost path is contained. Working within this slice, along the diagonal, allows a subset of D cost states perpendicular to the diagonal to result in parallel from the preceding subset of cost states, in step with the input stream of left and right image scanlines. D represents the maximum disparity range. Figure 2 shows a slice along the diagonal supporting, by way of example, a maximum disparity of 9 pixels. Starting from a known initial state (here c_{1A}, c_{00}, c_{2A}, c_{2B}, c_{2C}, c_{2D}, c_{2E}, c_{2F}, and c_{2G} lying on the axes), the next subset of cost states is calculated. For this purpose, the cost states are grouped as triads and occlusion (occl), and matching costs (s_{ij}) are added to the previous states, as shown in Figure 3. In this way, the processing element computes the cost of the

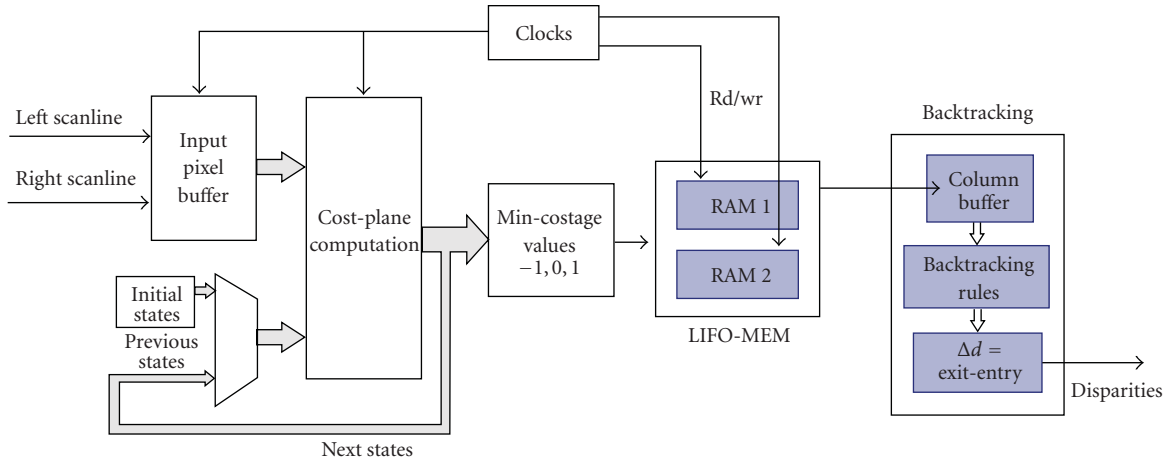


Figure 4: Hardware system based on maximum-likelihood dynamic programming algorithm.

diagonal, vertical, and horizontal path to each adjacent point. The minimum among the three cost values is produced by an appropriate min-computation parallel stage. Tag values $(-1, 0, 1)$ are attributed to all three possible paths and the wining tag, corresponding to the minimum cost, is stored in RAM memory, at each point of the cost plane. As shown in Figure 3, at each-one iteration the next subset of cost states are calculated in two phases. The nine states shown in circles are all the cost states needed for the next round of computations.

In this way, it is possible to calculate all states in the slice, up to the end of the cost-plane, using the same hardware stage. This stage receives as input the previous costs, along with the current left and right pixel intensities and produces the next subset of costs, like a state machine. Pixel intensities are used in the matching cost computation of (2).

RAM memory is implemented as M4K blocks, N positions deep, where N is the number of pixels per scanline. A number of D RAM blocks are needed (nine in the case of the state machine working on the cost plane of Figure 2). Each position is 2-bit wide, since it only stores a tag value -1, 0, or 1.

During backtracking, we compute N disparities for each one of the N scanline pixels, in N clock cycles. Winning tags are read from RAM memory in reverse order and are ordered according to the columns of the cost plane, using shift registers. At each step, we move from one column to the next, following the optimum path, according to the retrieved tag values and using well-defined rules [15]. At each step, the optimum path traverses vertically a tag column by a number of pixels equal to the change Δd of disparity at this particular step. Starting with $d = 0$ at the Nth pixel, the system tallies the disparity values down to the first pixel, adding Δd at each step.

Figure 4 shows a block diagram of the stereo accelerator hardware system.

3.2. The Corner Detector Stage. As mentioned above, corners in the left and right images are formed at the cross-section of vertical and horizontal edges. In order to produce image gradients, we convolute the images with 3×3 Prewitt masks. Image areas are formed with shift lines that store streaming input pixels. Using 3×3 windows results in a sensitive edge detector and at the same time keeps the required hardware resources low. Since the Prewitt masks contain only zeros and ones, the convolution is implemented with adders and subtractors, as shown in Figure 5. In the inset, the Prewitt mask for horizontal gradient is shown, along with the respective image window where the convolution is applied.

Vertical gradient is produced with the transposed Prewitt mask. A threshold value equal to 80 is used in order to produce horizontal and vertical binary edges. A binary AND operation produces thick white spots at the cross section of horizontal and vertical edges. White spots are ones, black areas are zeros. In 8-bit intensity terms ones are 255.

A thinning procedure is applied in the next stage, implementing the steps of the following algorithm:

(i) Convolute a 3×3 area of the binary corner image with horizontal Prewitt mask *M1*;

(ii) Convolute the same 3×3 area with the vertical Prewitt mask *M2*;

(iii) IF the central pixel in the 3×3 area is zero OR any of the convolutions in steps a and b is greater than 1

  (a) THEN central pixel remains zero;

  (b) ELSE central pixel is 1.

The idea behind the above hardware-friendly steps is to crop clusters of ones until only a central point remains. Point (iii) in the above algorithm is implemented according to Figure 6.

Finally, the consistency check is implemented by combining the disparity values produced in the stereo stage with the point features found above. This stage is shown in Figure 7. Stereo and corner stages work in parallel and the disparities stream can be easily aligned with the corner image stream by means of a delay line. Each corner feature in the left stereo frame is compared to the corresponding pixel in the right image frame. The corresponding feature is found by

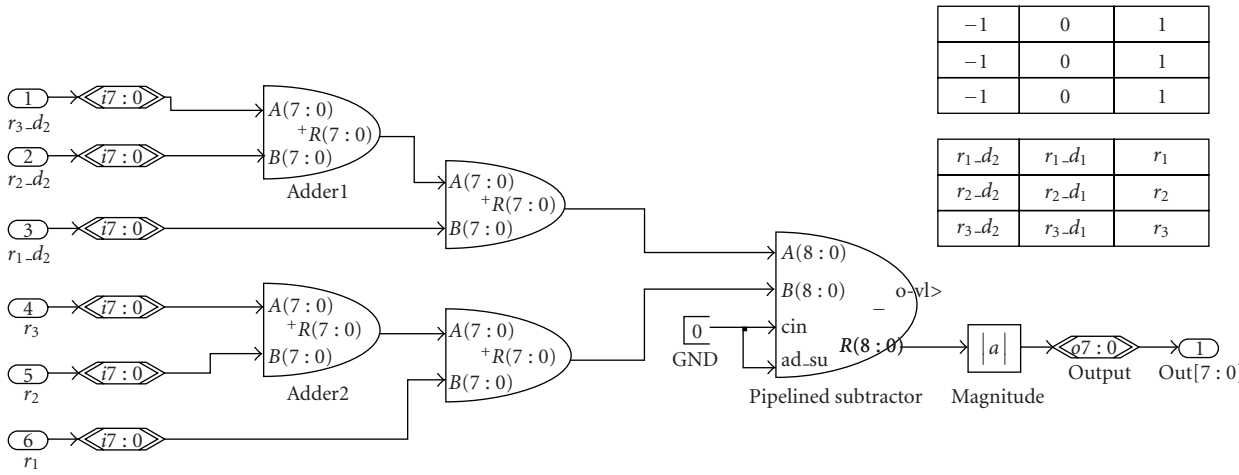


Figure 5: Application of the Prewitt mask for horizontal gradient computation.

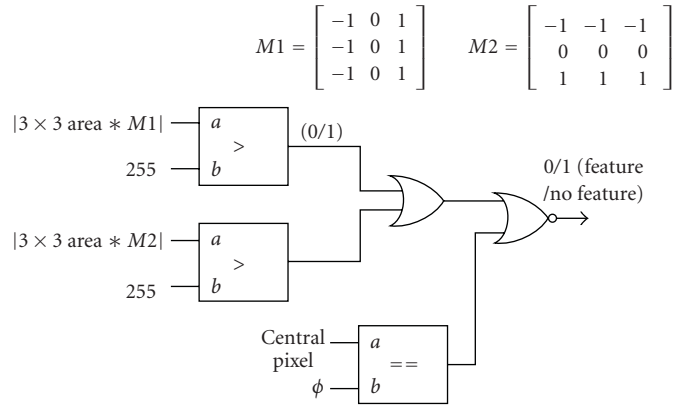


Figure 6: Implementation of the thinning step.

displacement according to the disparity value. In Figure 7 we simply use shift taps and a multiplexer in order to perform the above check.

3.3. System-on-A-Chip and Host Application. The system was modeled using CAD tools like DSPBuilder by Altera and was exported as an HDL library component, ready for integration in a System-on-a-programmable-chip. The embedded system was designed using SOPC Builder which is part of Quartus II, the Altera tool for system integration. The system is controlled by a 32-bit Nios II processor, and includes a DDR2 external memory controller and a USB 2.0 high speed controller for communication with a host computer. The feature detection hardware is integrated in the form of HDL library component. A set of DMA functions transfers data between units. The SOPC system is shown in Figure 8 as a block diagram.

We implement the system targeting a Cyclone II DSP board featuring a 2C35 FPGA device with total resources 33,000 logic elements and 480,000 bits of on-chip memory. On the host side, the stereo-head is connected to a National Instruments frame-grabber and is controlled by a LabVIEW application. The same application controls the USB 2.0 communication with the reconfigurable hardware part achieving a practical transfer rate greater than 60 Mbps. The host application receives the output from the hardware accelerator and uses feature positions and disparities as discrete landmark measurements. These measurements are used in the update stage of a FastSLAM simultaneous localization and mapping algorithm, running in the host side. This procedure is presented in Section 5.

4. Evaluation of the Hardware System

The more sophisticated stage in the system presented in the previous section is the stereo accelerator stage. It performs a demanding computational task and has a considerable cost in hardware resources. Table 1 shows the typical resources needed for a basic and more advanced implementation of the stereo processor. Increasing the range of disparities increases proportionally the necessary resources. Our DP-based accelerator requires on-chip memory for the storage of minimum cost tag values. Increasing image resolution increases proportionally the memory needed for the storage of tag-values per scanline. The feature detector stages including the corner detectors and left-right consistency check can

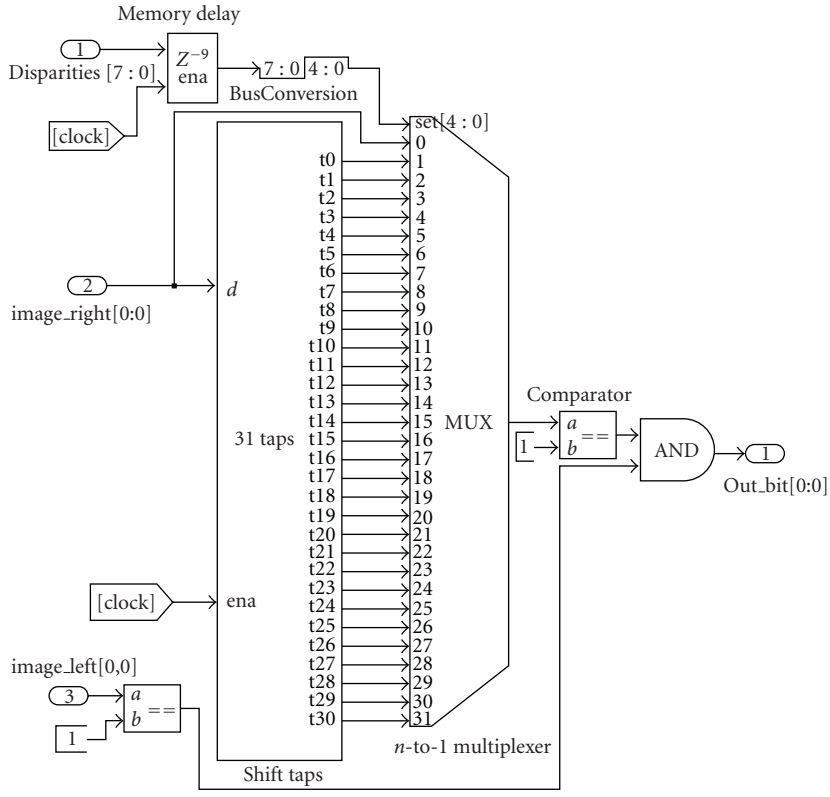


Figure 7: Left-right consistency check.

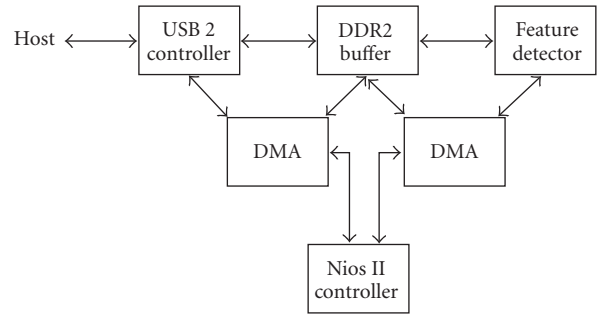


Figure 8: The main components of the system-on-a-programmable chip. The block *Feature detector* represents the system in Figure 1.

be implemented with 4000 additional logic elements. Nios II processor and peripheral controllers require an additional overhead of about 7000 LEs and 160000 bits of embedded memory. The system was implemented targeting a Cyclone II EP2C35 FPGA device.

The implemented stereo accelerator and feature detection system is able to process 49 levels of disparity and produce 640×480 8-bit depth maps and point features at clock rate. The higher possible frequency for our present implementation is 50 MHz. This timing restriction is caused by feedback loops, as for example in the cost-matrix computation stage and can potentially be resolved using appropriate hardware optimization. A pair of 640×480 images is processed at 12.28 ms, which is equivalent to 81 frames/s or 25 Mpixels/s. The reported throughput is high and suitable for demanding real-time applications, like navigation or environmental mapping. Table 2 presents timing requirements and nominal frame rates for the total system. In practice, apart from the stereo accelerator and feature detector throughput, the frame rate depends also on the camera type and other system parts. For comparison, the stereo matching procedure described in Section 3.1 has been implemented in software and was found to process a stereo pair with resolution 640×480 in about 20 seconds.

The stereo processor is the heart of the system and its performance was at first evaluated using reference stereo images. In addition to the Tsukuba University stereo pairs, the "cones," "sawtooth," "books," "arts," and "bowling" images from the public Middleburry data-set [10] were used for this evaluation. Figure 9 shows a series of ground

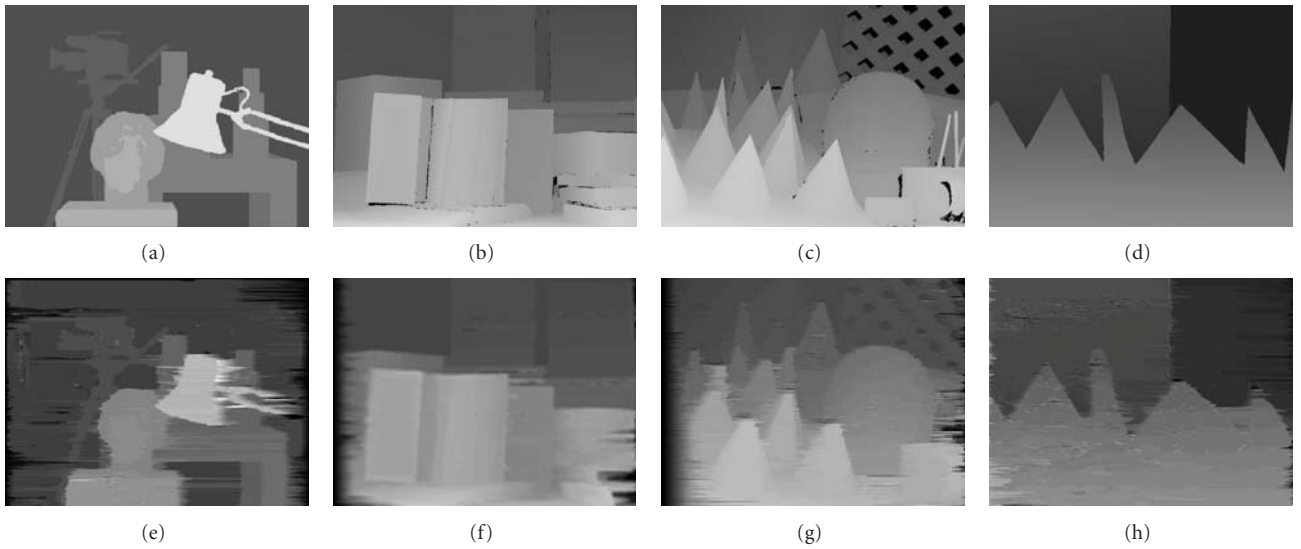


Figure 9: True disparities (upper row) and depth maps produced by the reconfigurable stereo processor (lower row), using the reference stereo set from the Middlebury data-base [10]. From left to right: "Tsukuba," "books," "cones," and "sawtooth" stereo images.

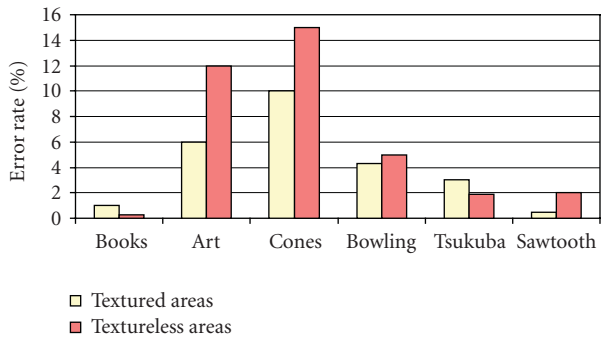


Figure 10: Error rate in terms of bad matches, produced by the reconfigurable stereo processor, using the Middlebury data set.

Table 1: Resource utilization for system implementation.

	Image resolution	Logic elements	Memory bits
Stereo processor/33 disparity levels	320×240	10000	74048
49 disparity levels	640×480	14700	173632
65 disparity levels	640×480	21500	270240
Feature detector	640×480	4,000	10,000
Nios II processor + Cashe	—	2,000	120,000
USB 2.0 controller + EP buffer	—	2520	80,000
Other controllers	—	4000	30,000
Total resources 49 disparity levels	640×480	27220	413632

truths in comparison with the corresponding disparity maps computed by the reconfigurable processor, based on dynamic programming. For quantitative quality assessment, the error

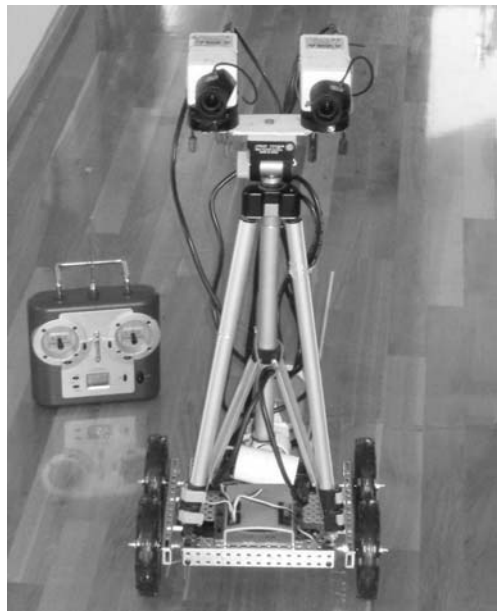
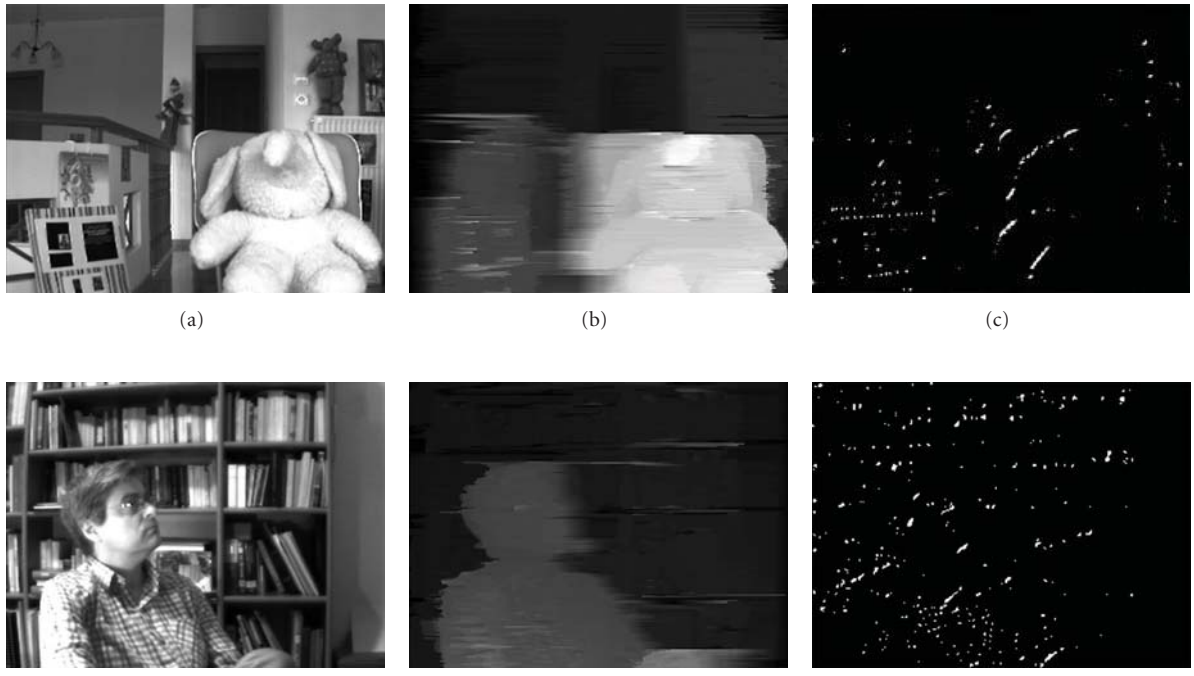


Figure 11: Mobile platform with stereo-head.

Table 2: Processing speed for the feature detection system.

Image resolution	Maximum achieved frequency (MHz)	Maximum throughput (Mpps)	Frame rate (fps)
640×480	50	25	81

metric proposed by Scharstein and Szeliski [12] was used. This measure is based on true disparities and computes the error rate as the percentage of pixels that differ from ground truth by more than one disparity pixel. Figure 10 shows the percentage of bad matches for the stereo-sets used in



(a) (b) (c)

(d) (e) (f)

Figure 12: (a,d) Left image of a stereo pair, (b,e) the disparity map produced by the reconfigurable stereo detector, and (c,f) corners prior to thinning.

this evaluation. The error rate in textured and textureless regions is examined separately. Textureless regions are image areas where the average horizontal gradient is below a given threshold. Occlusion boundaries are excluded from the above evaluation. As shown in Figure 10, bad matches are in average lower than 10%. This result confirms quantitatively that the proposed stereo processor produces reliable depth maps independently of image content. It also confirms that the proposed global method implemented in hardware is generally better than common local correlation techniques, like normalized cross correlation or SAD, although the later are easier to fit in specific hardware architectures. An assessment of different stereo-matching techniques can be found in [12].

The stereo processor was also evaluated in terms of depth maps acquired from familiar real scenes. Although this is a subjective measure, it can be a complementary indication of the quality of the proposed system. The stereo head shown in Figure 11 was used to capture and process real scenes. Typical results are shown in Figure 12. Left captured images are followed by depth maps produced by the proposed stereo processor. As shown in Figure 12, depth maps produced by the hardware stereo system are smooth and accurate in the most part of the image. Although some streaking effect is blurring object boundaries, the overall subjective result is satisfactory. Figure 12 also presents the corner points produced by the feature detector described above, prior to thinning. Results of the last two steps (thinning and consistency checking) are pixel-size and cannot be easily displayed.

5. 3D Mapping Experiments

5.1. The Localization and Mapping Procedure. A calibrated stereo head is adapted to a simple mobile vehicle, shown in Figure 11, which can be guided indoors using remote control. In this experimental setup, the mobile platform is tethered and transmits images to a desktop computer. The computer interfaces with the FPGA board through USB 2.0 high speed port. A pair of gray-scale 8-bit video sequences is captured at each step and dense depth disparities are produced along with consistent features in real-time using the system-on-a-chip presented above. The result is transmitted back in the form of pixel stream to the host computer via the USB 2.0 interface. The output pixel array contains zeros, where no feature is found, and the corresponding disparity value in every place where a point feature has been established. In the host part a measurement vector is attributed to every feature, giving the disparity and the feature position in left image coordinates at each time step t

$$z_t = [d, u_{\mathrm{R}}, v_{\mathrm{R}}]^T, \tag{3}$$

where u_{R}, v_{R} are the horizontal and vertical coordinates of a right image feature and $d = u_{\mathrm{L}} - u_{\mathrm{R}}$.

This measurement set is used as the observation part in a real-time simultaneous localization and mapping experiment. The vehicle is moving on a plane and the vision system reconstructs a complex map based on point features corresponding to 3D landmarks in the world frame.

In order to estimate vehicle motion, our algorithm uses odometry data for translational speed while it derives

Table 3: Performance comparison between hardware stereo matching systems.

Reference	Method of correspondence	Area utilization/memory bits	Image resolution/max disparity	Performance	Quality assessment (average bad matches)	Technology
Present work	Dynamic programming/maximum likelihood	27220 logic elements/413632	640 × 480/49 pixels	25 Mpps/81 fps	<10%	Altera Cyclone II FPGA board + Nios II controller
Diaz et al. (2007) [17]	Phase-based	13048 slices/1308672	1280 × 960/29 pixels	65 Mpps/52 fps	—	Custom FPGA, Xilinx Virtex-II
Ambrosch et al. (2009) [18]	Correlation-SAD	106658 logic elements/425984	330 × 375/120 pixels	136 fps	~38%	FPGA, Altera Stratix EP2S130
Darabiha et al. (2003) [19]	Local weighted phase correlation	~67000 4-input LUT/800000	360 × 276/20 pixels	2.8 Mpps/30 fps	<10%	Custom FPGA board Xilinx Virtex 2000
Liang et al. (2009) [20]	Tile-based belief propagation	2.5 Mgates total	640 × 480/64 pixels	8.2 Mpps/27 fps	—	ASIC
Niitsuma and Maruyama (2004) [21]	Correlation-SAD	31000 slices/405504	640 × 480/27 pixels	9.2 Mpps/30 fps	—	Custom FPGA Xilinx Virtex-II
Kalomiros and Lygouras (2008) [22]	Correlation-SAD	15000 logic elements/196000	320 × 240/32 pixels	25 Mpps/325 fps	~26%	FPGA, Altera Cyclone II
Wang et al. (2006) [23]	Dynamic programming	—	640 × 480/48 pixels	1 Mpps/3 fps	<10%	Graphics processing unit

rotational speed from visual data. This combination is proved to be robust. We find that measuring rotation by tracking image intensity features between frames can be very exact, while estimating translation using image data can be prone to error. This is true especially in the regime of high rotational speed, where image content tends to change rapidly.

Assuming a stereo system with parallel optical axes and a pinhole camera model, our nonlinear measurement model $\hat{z}_t = h(s_t, \hat{\theta})$ can be written in terms of an observed landmark's world coordinates $\hat{\theta} = (x_i,\ y_i,\ z_i)$ and camera position and rotation $s_t = (x_C,\ y_C,\ \psi)$

$$z_t = \begin{bmatrix} u_L - u_R \\ u_R \\ v_R \end{bmatrix} = \begin{bmatrix} \dfrac{fb}{(x_i - x_C)\cos\psi + (y_i - y_C)\sin\psi} \\ u_0 + \dfrac{f[(x_i - x_C)\sin\psi - (y_i - y_C)\cos\psi - b/2]}{(x_i - x_C)\cos\psi + (y_i - y_C)\sin\psi} \\ v_0 + \dfrac{f(z_i - z_C)}{(x_i - x_C)\cos\psi + (y_i - y_C)\sin\psi} \end{bmatrix}. \tag{4}$$

In the above equations, f is the camera focal length, b is the baseline of the stereo head, and $(u_0,\ v_0)$ is the central image pixel. The reference frame of the stereo head is shown in Figure 13.

FastSLAM algorithm proposed by Montemerlo and colleagues [24] is adapted to the case of our stereo-assisted point feature observations captured by the hardware system. This method is a Bayesian algorithm for the estimation of the robot's path and environmental map based on landmark observations. It approaches the localization and mapping problem by maintaining low-dimensional Bayesian filters instead of the multidimensional state vector and covariance matrix of the majority of SLAM approaches. Low-dimensionality reduces computational complexity and it is a prerequisite for real-time implementation of localization and mapping in the case of large 3D maps that contain thousands or millions of landmarks. FastSLAM factorizes a posterior over maps and robot paths

$$p(s^t, \Theta \mid z^t, u^t, n^t) = p(s^t \mid z^t, u^t, n^t) \prod_{n=1}^{N} p(\theta_n \mid s^t, z^t, u^t, n^t), \tag{5}$$

where s^t is the robot's path until time t, which is the set of all positions $\{s_t\}$. Θ is the map which consists of the mean position and the covariance of all landmarks. z^t is the set of observed features $\{z_t\}$ from time 0 until time t, where the measurement z_t at each time step is given by (3). n^t is the set of correspondences between observed features z_t and stored landmarks θ_t in the map. The history of control parameters is represented by u^t. Superscripts denote the whole set of data until time t, while subscripts denote data values at a given time step.

The above factorization, first developed by Murphy [25], is derived from the fact that given the vehicle's path,

the position of every landmark can be estimated as an independent quantity. In FastSLAM, the distribution for the robot posterior $p(s^t \mid n^t, z^t, u^t)$ in (5) is estimated using a particle filter, and the remaining N conditional landmark posteriors $p(\theta_n \mid s^t, z^t, u^t, n^t)$ are estimated using Extended Kalman Filters (EKF). Each particle m in the particle filter contains a robot path s^t and N independent EKFs, each one tracking a single landmark position [26].

In order to estimate the map feature positions and the vehicle's path according to the posterior (5) we implement the steps summarized below.

(1) Obtain the controls u_t for the last step and sample a new vehicle pose for each particle. This sampling step is performed using the vehicle's motion model and represents the particle filter's proposal distribution. In our implementation, vehicle controls u_t represent translational speed $\Delta s/\Delta t$ which is derived from odometry and rotational speed $\Delta\psi/\Delta t$ which is derived by minimizing absolute intensity differences between successive video frames.

(2) A new set of measurements is captured according to (3) and the resulting point features are associated to previous measurements in the map. Data association n_t is chosen by calculating the minimum Mahalanobis distance between the measured and stored landmarks

$$\hat{n}_t = \arg\min_{n_t} \left\{ (z_t - \hat{z}_{n,t})^T Z_{n,t}^{-1} (z_t - \hat{z}_{n,t}) \right\}, \tag{6}$$

where z_t is the current measurement and $\hat{z}_{n,t}$ is the measurement prediction of the nth landmark stored in the map Θ. The prediction of measurements for stored landmarks is performed by the measurement model (4). $Z_{n,t}$ is the measurement innovation covariance for the nth landmark at time step t, given in the context of the EKF by

$$Z_{n,t} = H_{\theta_{n_t}} P_{n_t,t-1}^{[m]} H_{\theta_{n_t}}^T + R_t, \tag{7}$$

where R_t is the measurement error covariance matrix and $P_{n_t,t-1}^{[m]}$ is the error covariance for the nth landmark in the mth particle. $H_{\theta_{n_t}}$ is the observation matrix, calculated as the Jacobian of the measurement model, with respect to landmark coordinates

$$\hat{z}_t = h\left(s_t^{[m]}, \hat{\theta}_{n_t,t-1}\right), \tag{8}$$

$$H_{\theta_{n_t}} = \nabla_{\theta_{n_t}} h(s_t, \theta_{n_t}). \tag{9}$$

Equation (8) gives the measurement prediction according to the robot path and map in the mth particle and is given analytically by (4). $H_{\theta_{n_t}}$ is calculated from (4) differentiating with respect to x_i, y_i, z_i:

$$H = \begin{bmatrix} \frac{\partial(u_L - u_R)}{\partial x_i} & \frac{\partial(u_L - u_R)}{\partial y_i} & \frac{\partial(u_L - u_R)}{\partial z_i} \\ \frac{\partial u_R}{\partial x_i} & \frac{\partial u_R}{\partial y_i} & \frac{\partial u_R}{\partial z_i} \\ \frac{\partial v_R}{\partial x_i} & \frac{\partial v_R}{\partial y_i} & \frac{\partial v_R}{\partial z_i} \end{bmatrix}$$
$$= \begin{bmatrix} \frac{-fb\cos\psi}{X^2} & \frac{-fb\sin\psi}{X^2} & 0 \\ \frac{f(y_i - y_C + (b/2)\cos\psi)}{X^2} & \frac{-f(x_i - x_C - (b/2)\sin\psi)}{X^2} & 0 \\ \frac{-f(z_i - z_C)\cos\psi}{X^2} & \frac{-f(z_i - z_C)\sin\psi}{X^2} & \frac{f}{X} \end{bmatrix}, \tag{10}$$

where $X = (x_i - x_C)\cos\psi + (y_i - y_C)\sin\psi$.

(3) There follows the update stage of the landmarks' EKFs, according to the update equations of the EKF theory [27]. Updating the filter produces the posterior at time t from the one at time $t-1$. Features with null correspondences are initialized in terms of their mean and covariance and are inserted in each particle as new landmarks.

Steps (2) and (3) are repeated for all features captured in a frame at each time step t.

(4) After processing all measurements in a given stereo pair, each particle m in the current generation of particles is weighted according to the probability of the current observation, conditioned on the robot path. In our implementation, this is equal to the probability M of all measurements at time t, and can be defined in terms of log-likelihoods as

$$\begin{aligned} \log M &= \sum_i \log p\left(z_{t,i} \mid s^{t,[m]}, z^{t-1}, u^t, n^t\right) \\ &= \sum_i -\frac{1}{2} \min\left\{ (z_{t,i} - \hat{z}_{n_t,t})^T Z_{n,t}^{-1} (z_{t,i} - \hat{z}_{n,t}) \right\}. \end{aligned} \tag{11}$$

The sum is taken over all observations in a given stereo pair. Each measurement probability is equal to the probability of the innovation $z_{t,i} - \hat{z}_{n_t,t}$, given by the Mahalanobis distance for the correct correspondence n_t. The weight update for particle m is calculated by

$$\hat{w}_{m,t} = \frac{\exp(\log M)}{\sum_{i=1}^N \exp(\log M)} w_{m,t-1}, \tag{12}$$

where N is the number of observations in the stereo pair. Particle weights are finally normalized

$$w_{m,t} = \frac{\hat{w}_{m,t}}{\sum_{m=1}^M \hat{w}_{m,t}}. \tag{13}$$

(5) Particles are redistributed in order to correct the difference between the proposal distribution and the desired posterior (5). The resampling procedure in our implementation is performed according to the technique "Select with replacement" [28].

The above basic operations are repeated at each time step. At the final time step, the vehicle path and the map are

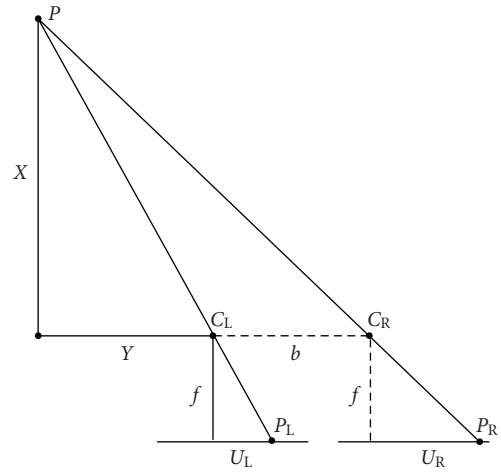


Figure 13: Top-down view of two identical parallel cameras with focal length f at distance b to each other.

estimated according to the particle possessing the maximum weight. The main steps in the above algorithm are derived from Montemerlo's original implementation of FastSLAM 1.0 [24, 26] while (10) represents the observation model developed in the present paper. Equations (11), (12), and (13) represent our preferred method for weight update in terms of the "select with replacement" resampling algorithm.

5.2. Experimental Results. Localization and mapping experiments were carried out using the observation model described in the previous sections. The vehicle is guided indoors by a human operator, recording features in a 5.5×4.2 m room, following an almost circular path, and completing one or two rounds. The speed on the direction of motion is kept constant and equal to 0.15 m/s. A nonzero rotational speed is maintained so that the ability of the system to tolerate errors in odometry is tested. Multiple rounds can test the behavior of the system at loop closure.

Figure 14(a) presents a result of mapping the room and simultaneously localizing the vehicle, employing just one particle in our FastSLAM implementation. In this figure, map features are shown with blue dots at the room's perimeter, while the red line records the vehicle path as estimated by the algorithm. For comparison the true path is presented with crosses, as it was recorded on the floor by a marker adapted on the vehicle. Perimetric lines show the true position of the walls and furniture. The metric map presented in Figure 14(a) is referenced to the initial position of the vehicle at point (0,0). Figure 14(b) shows the result after two rounds of the vehicle and utilizing five particles in the estimation procedure. New landmarks observed during the second round are shown in this figure with purple dots. Figure 14(c) shows the result of the same experiment utilizing ten particles. Figure 15 shows a 3D map of the room obtained using two independent particles.

As is shown in these figures, acceptable maps and robot paths can be estimated even with just one particle, while ten particles make a slight improvement over five particles.

In spite of the low dimensionality of the measurement vector, features produced by the stereo sensor are tracked efficiently from frame to frame and are updated appropriately, keeping the total number of features in the map relatively low. At loop closure, revisited features are recognized and updated. As shown in Figure 14(b), relatively few point features are acknowledged as new features in the second round, since most of them have already been observed during the first round of the vehicle.

The accuracy of the above procedure mainly depends on the accuracy of odometry measurements and the accuracy of disparity measurements. The Bayesian algorithm can partially correct odometry errors by tracking feature points and updating the filter. However, systematic errors in pixel disparity, especially at occlusion boundaries and at regions without texture, can introduce erroneous observations into the map. Although FastSLAM tends to remove wrong associations from the map, by propagating high probability particles in the resampling step and attenuating improbable ones, this does not work in case the error distribution is not Gaussian. The proposed semiglobal stereo-matching algorithm implemented in hardware is proved to produce quite accurate measurements, as compared to less sophisticated techniques tested for the same purpose. For example, an implementation of the Sum of Absolute Differences (SAD) in reconfigurable hardware [22] was also tested, in the place of the stereo processor stage of our sensor. It is found that the map produced by the SAD stereo processor includes a considerable number of outliers; it presents fewer consistent features and a more dispersed cloud of points. The result of this experiment is shown for comparison in Figure 16.

Next, the real-time processing ability of the proposed system is explored. Figure 17 shows processing rate (steps or iterations per second) as a function of the number of particles in the FastSLAM filter, for the first one hundred steps. Adequate maps were produced even with a single particle. Using more particles has as a result the most probable map and robot path, since each particle contains its own vehicle path and a map conditioned on this path. In each processing step the vehicle is promoted forward, a new stereo pair is captured, the dense depth map is produced, and a set of point features is extracted using the proposed

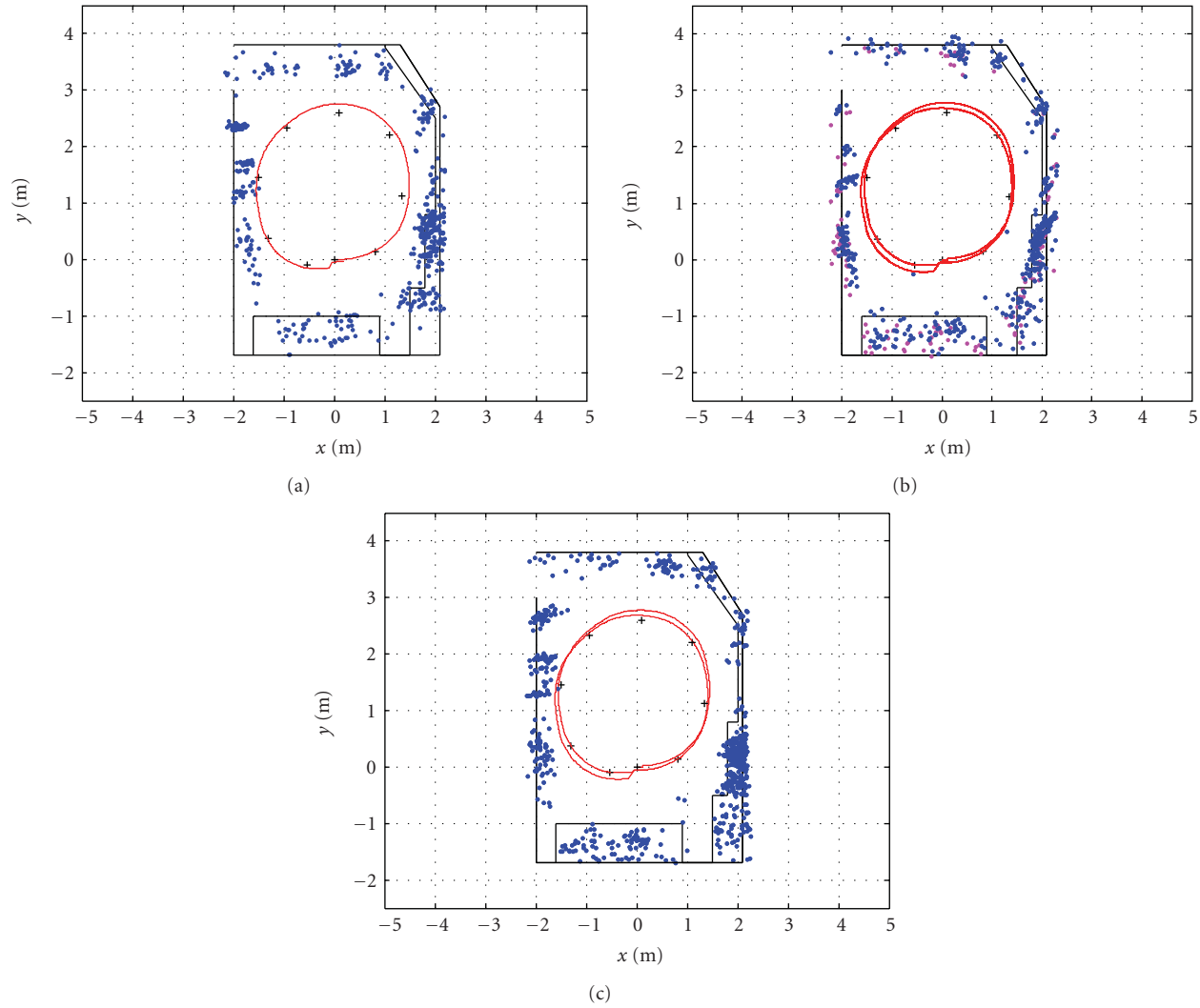


Figure 14: Projection of the estimated map (blue dots) on the *xy* plane. Red line is the estimated vehicle path. Black lines represent the ground truth of the objects in the map and crosses are the true vehicle positions during the first round. (a): result after one round, with one particle in the filter. (b): five particles, two rounds. Purple dots represent new landmarks observed during the second round. (c): ten particles.

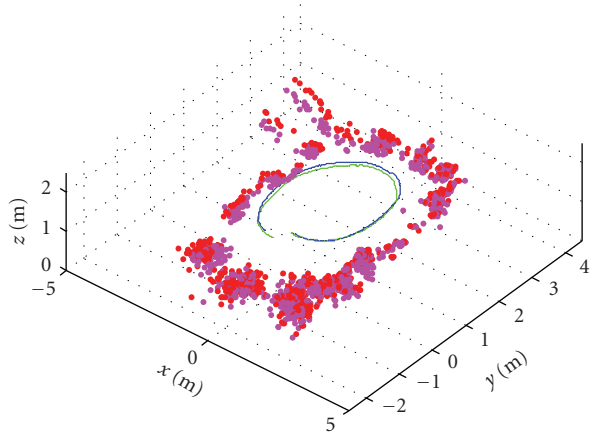


Figure 15: 3D graph of the estimated map. Two independent particles are projected in the same map.

reconfigurable system. The measurement set is input to the FastSLAM filter, which uses the measurements in order to update the particle weights and the positions of the observed landmarks. As it is shown in Figure 17, the proposed system can process up to 12 full iterations per second with just one particle and slows down to approximately two iterations per second for one hundred particles. Using more than ten particles, however, deteriorates the real-time efficiency of the algorithm, especially when thousands of features are gathered in the map, after several minutes of exploration. Certainly, the notion of real-time processing here depends on the required speed of the vehicle. Our experiments were conducted with the controller preserving constant speed 0.15 m/s on the direction of motion. Using this standard, the system can process at least four iterations per second without loosing track of the observed landmarks in subsequent camera frames. The ability to track and correlate features

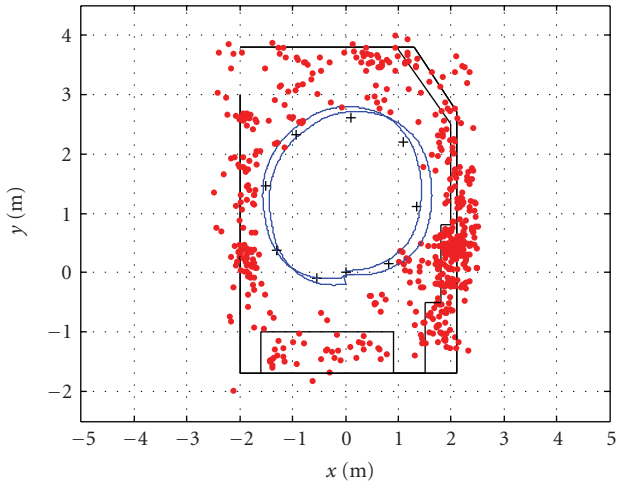


Figure 16: Result of a localization and mapping experiment in the same premises as in Figure 14 using a SAD reconfigurable processor in the place of the proposed dynamic programming stereo system.

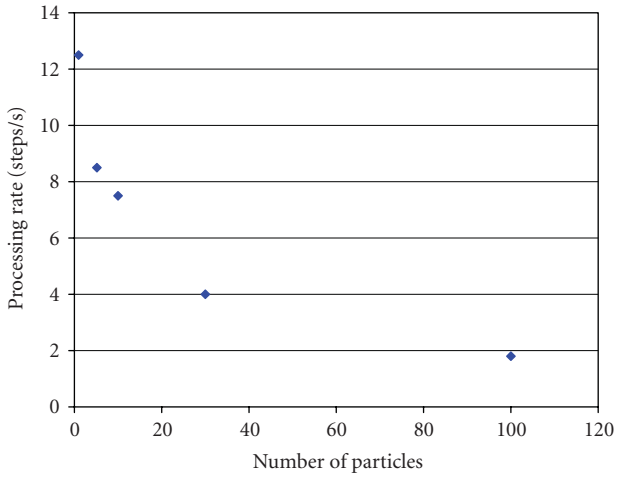


Figure 17: Processing rate in the first one hundred steps of the vehicle, as a function of the number of particles in the FastSLAM filter.

between frames is imperative for the correct function of the proposed system.

On the other hand, the processing rate is a function of the number of landmarks in the map. The map is augmented as the vehicle moves forward gathering new measurements. Figure 18 presents the evolution of processing time as a function of the number of processing steps, for different number of particles in the filter. For more than ten particles, the processing time increases superlinearly with the number of steps, hence the processing rate decreases. Choosing between five and ten particles, we find that the proposed system is well within limits of real-time execution of the FastSLAM algorithm and also keeps its processing rate almost constant. We also note that utilizing more than ten particles in the filter does not result in perceptible improvements in the estimation of the robot path and environmental map.

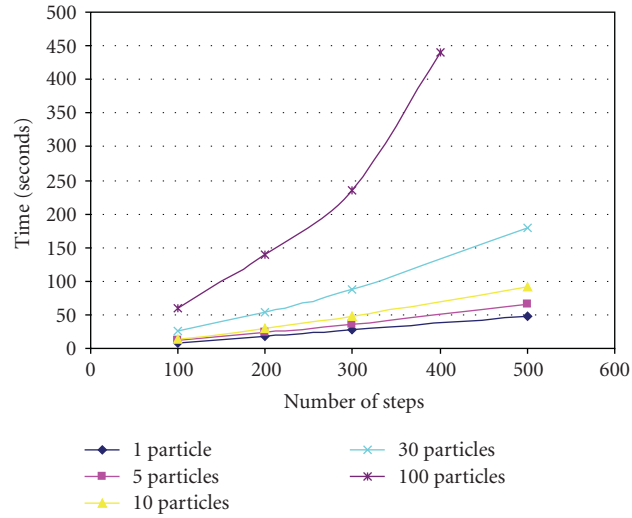


Figure 18: Processing time with increasing number of steps, for different number of particles in the filter.

Processing rate limitations in the proposed system are due to the computational load in the software part of the system, which implements the FastSLAM algorithm on a desktop computer with a quad core CPU running at 2.4 GHz. The configurable part of the system can respond well within the rate allowed by the software part and does not add to the computational load. This is characteristic of the significance of specific hardware in a robotic localization and mapping procedure. As it is noted by other researchers, the typical time required for each iteration can measure up to several seconds and most of this time is usually spent on the feature extraction stage [29, 30]. Attributing feature extraction to dedicated hardware can ease the computational load and allow real-time response. This is particularly true for stereo-assisted feature extraction.

The present experiments were conducted indoors with a stereo baseline adjusted for optimum operation at several meters. A proper autonomous vehicle and stereo head are under development in order to conduct tests in larger or outdoors environments.

6. Comparison with Other Systems

A meaningful comparison can be made with other stereo sensors designed in reconfigurable hardware. A number of real-time stereo systems were presented in the literature in recent years. Several systems were built using FPGA devices, like the Xilinx Virtex series or the Altera Cyclone and Stratix families. Most such systems are based on area correlation methods, using techniques like SAD [31, 32] and rank transforms [33]. Ambrosch and co-authors have recently implemented SAD-based stereo matching, rank and cencus transform using FPGAs [18, 34]. They use 106658 logic elements out of a Stratix EP2S130 device, and their quality assessment yields 61.12% and 79.85% correct matches for their SAD and census transform, respectively, while frame rates range to several hundreds.

Niitsuma and Maruyama [21] described a real-time system for the detection of moving objects, based on SAD stereo matching. Their implementation uses a Virtex-II FPGA and processes 30 frames per second with resolution 640×480 pixels.

Darabiha et al. [19] and Masrani and MacLean [35] implemented phase-based stereo in FPGAs using the equivalent of 70000 logic elements and about 800 Kbits of on-chip memory. Their system is built with Stratix S80 devices and supports a maximum disparity range of 128 pixels. Diaz et al. implemented a fine-grain pipeline structure for phase-based disparity estimations. Their implementation uses a Xilinx Virtex-II FPGA device and is able to produce one pixel of disparity per system clock cycle [17]. A similar phase-based implementation is also reported in [36].

Global optimization VLSI architectures are just beginning to emerge. Recently, hardware-efficient algorithms for realizing belief propagation were proposed. Belief propagation (BP) provides global optimality and good matching performance when applied to disparity estimation. However, it has great computational complexity and requires huge memory and bandwidth. Cheng et al. [37] propose a tile-based BP algorithm in order to overcome the memory and bandwidth bottlenecks. In a related paper Liang et al. [20] implemented the tile-based approach as well as a method for parallel message construction on a dedicated VLSI chip. The chip consists of 2.5 M gates and processes images in VGA resolution with 64 pixels maximum disparities at 27 frames per second.

The problem of mapping in hardware belief propagation for global stereo estimation is also addressed by Park and Jeong [38]. They implement a BP algorithm using a Xilinx XC2vp100 FPGA chip. They process 16 disparity levels and image resolution 256×240 at 25 frames per second, using 9564 slices (approximately 20000 LE) and more than 10 Mbits of memory partitioned in on-chip RAM blocks and external SRAM.

Many researchers investigate the performance of commodity graphics cards in accelerating stereo algorithms. Most implementations investigate local correlation techniques and achieve real-time performance [13, 39]. Global algorithms produce good results but are computationally intensive and not very appropriate for real-time speed. Some attempts to parallelize dynamic programming on Graphics Processing Units (GPUs) achieve good balance between quality and speed and can be a good choice for PC-oriented applications [23, 40]. However, using dedicated hardware, like in our proposed system, is faster and more suitable for autonomous systems equipped with vision sensors.

Table 3 presents a performance comparison between recently published stereo matching hardware systems. Most systems in this table are implemented using FPGAs, while one system is prototyped as Application Specific Integrated Circuit (ASIC) and one is a GPU implementation, shown here for comparison. Area utilization is given in various units, as published. We note that one logic element is approximately equivalent to one 4-input Lookup Table (LUT) and one slice is approximately equivalent to two 4-input LUTs.

In Tables 1 and 2 of the present paper resource usage and processing speed are presented for our reconfigurable stereo system designed to assist detection of point features for robot mapping. These data compare favorably with implementations reported in the aforementioned literature. The strong point of the architecture proposed in this paper is its full parallelism, resulting in one output disparity pixel at every clock cycle. Parallelism of the stereo algorithm is achieved by means of a state machine allowing cost computation of D states along the diagonal of the cost plane in one clock cycle. The system processes 25 Mpps or 81 frames per second in full VGA resolution (640×480). Resolving timing restrictions can result in even higher performance.

As shown in the previous section, the features computed by the presented hardware can be used successfully in the measurement part of a real-time simultaneous localization and mapping experiment. Landmark-based SLAM with stereo vision has also been explored by a number of other researchers. Most of them exploit the properties of discrete multidimensional image features, like SIFT features [41] and Shi-Tomasi features [42]. Others track 3D line segments [8]. They all use sparse disparity values computed at the location of interest by some optimized software technique. In our work, we exploit stereo acceleration provided by reconfigurable hardware and produce high accuracy dense depth maps using a global method of correspondence based on dynamic programming. Dense and accurate depth information makes it possible to extract low-dimensional point features, using an integrated multistage system on a chip. Processing simple features makes it possible to manage large maps and robot paths in real-time by applying a suitable FastSLAM algorithm, as presented in the previous sections.

7. Conclusions

In this paper, an integrated system-on-a-chip is presented for the detection of stereo-assisted point features, suitable for robotic mapping and localization. The system is based on a stereo accelerator implementing a parallelized semiglobal dynamic programming technique. Corners are extracted from the stereo pair using horizontal and vertical edge detectors and a hardware-friendly thinning technique. A final hardware stage implements left-right consistency check based on the disparity values extracted from the dense depth map at corner pixels. The system is implemented as a system-on-a-chip with a Nios II processor for data control and a USB 2.0 high speed interface for communication with a host computer. A stereo head is adapted on a mobile platform and the reconfigurable feature detector is used to implement the measurement part in a simultaneous localization and mapping experiment. A state-of-the-art FastSLAM algorithm on the host part is used to test mapping accuracy and real-time performance of the system. The experimental results illustrate that the system is able to perform in real-time and produce robot paths and maps of indoors environments. Future work will further explore the use of dense depth to real-time vision-based robotic navigation.

References

[1] W. van der Mark and D. M. Gavrila, "Real-time dense stereo for intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 7, no. 1, pp. 38–50, 2006.

[2] S. Thrun, *Robotic Mapping: A Survey. Exploring Artificial Intelligence in the New Millenium*, Morgan Kaufmann, Boston, Mass, USA, 2002.

[3] S. Se, D. Lowe, and J. Little, "Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks," *International Journal of Robotics Research*, vol. 21, no. 8, pp. 735–758, 2002.

[4] M. W. M. Gamini Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 3, pp. 229–241, 2001.

[5] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "MonoSLAM: real-time single camera SLAM," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1052–1067, 2007.

[6] J. Shi and C. Tomasi, "Good features to track," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '94)*, pp. 593–600, June 1994.

[7] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV '99)*, vol. 2, pp. 1150–1157, Kerkyra, Greece, September 1999.

[8] M. N. Dailey and M. Parnichkun, "Landmark-based simultaneous localization and mapping with stereo vision," in *Proceedings of the Asian Conference on Industrial Automation and Robotics*, 2005.

[9] C. J. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, Manchester, UK, 1988.

[10] May 2010, http://cat.middlebury.edu/stereo/data.html.

[11] M. Z. Brown, D. Burschka, and G. D. Hager, "Advances in computational stereo," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 8, pp. 993–1008, 2003.

[12] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, no. 1–3, pp. 7–42, 2002.

[13] M. Gong, R. Yang, L. Wang, and M. Gong, "A performance study on different cost aggregation approaches used in real-time stereo matching," *International Journal of Computer Vision*, vol. 75, no. 2, pp. 283–296, 2007.

[14] J. Kalomiros and J. Lygouras, "A reconfigurable architecture for stereo-assisted detection of point-features for robot mapping," in *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 404–409, Cancun, Mexico, December 2009.

[15] I. J. Cox, S. L. Hingorani, S. B. Rao, and B. M. Maggs, "A maximum likelihood stereo algorithm," *Computer Vision and Image Understanding*, vol. 63, no. 3, pp. 542–567, 1996.

[16] Y. Ohta and T. Kanade, "Stereo by intra- and inter-scanline search using dynamic programming," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 2, pp. 139–154, 1985.

[17] J. Díaz, E. Ros, R. Carrillo, and A. Prieto, "Real-time system for high-image resolution disparity estimation," *IEEE Transactions on Image Processing*, vol. 16, no. 1, pp. 280–285, 2007.

[18] K. Ambrosch, W. Kubinger, M. Humenberger, and A. Steininger, "Flexible hardware-based stereo matching," *EURASIP Journal on Embedded Systems*, vol. 2008, Article ID 386059, 12 pages, 2008.

[19] A. Darabiha, J. Rose, and W. J. MacLean, "Video-rate stereo depth measurement on programmable hardware," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '03)*, vol. 1, pp. 203–210, Madison, Wis, USA, June 2003.

[20] C. K. Liang, C. C. Cheng, Y. C. Lai, L. G. Chen, and H. H. Chen, "Hardware-efficient belief propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '09)*, pp. 80–87, Miami, Fla, USA, June 2009.

[21] H. Niitsuma and T. Maruyama, *Real-Time Detection of Moving Objects*, vol. 3203 of *Lecture Notes in Computer Science*, Springer, New York, NY, USA, 2004.

[22] J. A. Kalomiros and J. Lygouras, "Hardware implementation of a stereo co-processor in a medium-scale field programmable gate array," *IET Computers and Digital Techniques*, vol. 2, no. 5, pp. 336–346, 2008.

[23] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister, "High-quality real-time stereo using adaptive cost aggregation and dynamic programming," in *Proceedings of the 3rd International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT '06)*, pp. 798–805, Washington DC, USA, June 2006.

[24] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: a factored solution to the simultaneous localization and mapping problem," in *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02), the 14th Innovative Applications of Artificial Intelligence Conference (IAAI '02)*, pp. 593–598, Edmonton, Canada, July-August 2002.

[25] K. Murphy, "Bayesian map learning in dynamic environments," in *Advances in Neural Information Processing Systems (NIPS)*, MIT Press, Cambridge, Mass, USA, 1999.

[26] M. Montemerlo and S. Thrun, "FastSLAM, a scalable method for the simultaneous localization and mapping problem in robotics," in *Springer Tracts in Advanced Robotics*, B. Sicilinao, O. Khatib, and F. Groen, Eds., vol. 27, Springer, Berlin, Germany, 2006.

[27] M. S. Grewal and A. P. Angus, *Kalman Filtering, Theory and Practice*, Prentice-Hall, Upper Saddle River, NJ, USA, 1993.

[28] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEE Proceedings: Radar, Sonar and Navigation*, vol. 146, no. 1, pp. 2–7, 1999.

[29] R. Sim, P. Elinas, M. Griffin, and J. Little, "Vision-based SLAM using the Rao-Blackwellized particle filter," in *Proceedings of IJCAI Workshop on Reasoning with Uncertainty in Robotics*, pp. 9–16, Edinburgh, UK, 2005.

[30] M. H. Li, B. R. Hong, R. H. Luo, and Z. H. Wei, "Novel method for mobile robot simultaneous localization and mapping," *Journal of Zhejiang University: Science*, vol. 7, no. 6, pp. 937–944, 2006.

[31] Y. Miyajima and T. Maruyama, "A real-time stereo vision system with FPGA," in *Field Programmable Logic and Applications*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 448–457, Springer, Berlin, Germany, 2003.

[32] M. Hariyama, Y. Kobayashi, H. Sasaki, and M. Kameyama, "FPGA implementation of a stereo matching processor based on window-parallel-and-pixel-parallel architecture," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E88-A, no. 12, pp. 3516–3521, 2005.

[33] J. Woodfill and B. von Herzen, "Real-time stereo vision on the PARTS reconfigurable computer," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 201–210, April 1997.

[34] K. Ambrosch, M. Humenberger, W. Kubinger, and A. Steininger, "SAD-based stereo matching using FPGAs," in *Embedded Computer Vision, Part II*, B. Kisacanin, S. Bhattacharyya, and S. Chai, Eds., Springer, London, UK, 2009.

[35] D. K. Masrani and W. J. MacLean, "A real-time large disparity range stereo-system using FPGAs," in *Proceedings of the 4th IEEE International Conference on Computer Vision Systems (ICVS '06)*, pp. 13–20, January 2006.

[36] J. Díaz, E. Ros, S. Mota, E. M. Ortigosa, and B. del Pino, "High performance stereo computation architecture," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 463–468, Tampere, Finland, August 2005.

[37] C. C. Cheng, C. K. Liang, Y. C. Lai, H. H. Chen, and L. G. Chen, "Analysis of belief propagation for hardware realization," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS '08)*, pp. 152–157, Washington, DC, USA, October 2008.

[38] S. Park and H. Jeong, "High-speed parallel very large scale integration architecture for global stereo matching," *Journal of Electronic Imaging*, vol. 17, no. 1, Article ID 010501, 2008.

[39] F. Tombari, S. Mattoccia, L. D. Stefano, and E. Addimanda, "Classification and evaluation of cost aggregation methods for stereo correspondence," in *Proceedings of the 26th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '08)*, pp. 1–8, June 2008.

[40] J. Congote, J. Barandiaran, I. Barandiaran, and O. Ruiz, "Realtime dense stereo matching with dynamic programming in CUDA," in *Proceedings of the 19th Spanish Congress of Graphical Informatics (CEIG '09)*, pp. 231–234, San Sebastian, Spain, September 2009.

[41] R. Sim, P. Elinas, M. Griffin, A. Shyr, and J. J. Little, "Design and analysis of a framework for real-time vision-based SLAM using Rao-Blackwellised particle filters," in *Proceedings of the 3rd Canadian Conference on Computer and Robot Vision (CRV '06)*, Quebec, Canada, June 2006.

[42] A. Cumani, S. Denasi, A. Guiducci, and G. Quaglia, "Robot localization and mapping with stereo vision," *WSEAS Transactions on Circuits and Systems*, vol. 3, no. 10, pp. 2116–2121, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 727909, 10 pages
doi:10.1155/2010/727909

Research Article

A Reconfigurable System Approach to the Direct Kinematics of a 5 *D.o.f* Robotic Manipulator

Diego F. Sánchez, Daniel M. Muñoz, Carlos H. Llanos, and José M. Motta

Departamento de Engenharia Mecânica, Universidade de Brasília, 70910-900 Brasilia, DF, Brazil

Correspondence should be addressed to Carlos H. Llanos, llanos@unb.br

Received 10 March 2010; Revised 19 October 2010; Accepted 17 December 2010

Academic Editor: Lionel Torres

Hardware acceleration in high performance computer systems has a particular interest for many engineering and scientific applications in which a large number of arithmetic operations and transcendental functions must be computed. In this paper a hardware architecture for computing direct kinematics of robot manipulators with 5 degrees of freedom (5 *D.o.f*) using floating-point arithmetic is presented for 32, 43, and 64 bit-width representations and it is implemented in Field Programmable Gate Arrays (FPGAs). The proposed architecture has been developed using several floating-point libraries for arithmetic and transcendental functions operators, allowing the designer to select (pre-synthesis) a suitable bit-width representation according to the accuracy and dynamic range, as well as the area, elapsed time and power consumption requirements of the application. Synthesis results demonstrate the effectiveness and high performance of the implemented cores on commercial FPGAs. Simulation results have been addressed in order to compute the Mean Square Error (MSE), using the Matlab as statistical estimator, validating the correct behavior of the implemented cores. Additionally, the processing time of the hardware architecture was compared with the same formulation implemented in software, using the PowerPC (FPGA embedded processor), demonstrating that the hardware architecture speeds-up by factor of 1298 the software implementation.

1. Introduction

Currently, there exists a real demand in many robotics applications for higher operational speeds, and the solutions improving performance would have clear benefits in terms of manufacturing efficiency, precision, and processing time. In general, a fully operational robotic system running in real-time requires the repeated execution of a variety of complex algorithms, which involve the use of several transcendental functions and arithmetic operations. Most of these algorithms, if not all, need to be computed within milliseconds (ms) or microseconds (μs) in order to keep several real-time constraints. Such algorithms require massive computing power that surpasses the capabilities of many sequential computers [1].

A classical but important problem in the robotic manipulator is the direct kinematics. The direct kinematics allows the Cartesian location of the end effector to be calculated from measured values of the joint angles [2]. This mathematical formulation frequently needs a large number of arithmetic and trigonometric computations that must ideally be performed in a floating point, because of precision requirement [2]. Design and implementation of floating-point arithmetic operations in FPGAs has a relevant importance in a variety of scientific applications (such as robotics) due to the large dynamic range for representing real numbers, which permits suitable representation of both very small and large numbers in a fixed bit-width with proper precision. Additionally, the software implementations of direct kinematics computations are very expensive in processing time due to the sequential behavior of general purpose processors (GPPs).

Floating-point-based algorithms for arithmetic operators are commonly implemented on software and executed in microprocessors. Typically this solution requires to pay a performance penalty given that the conventional approaches require to perform the data transfer between the ALU (Arithmetic Logic Unit) and the program and

the instruction memories. This problem, well known as von Neumann bottleneck, has been partially overcome by using multicores microprocessors reducing the execution time. Recently, Graphic Processor Units (GPUs) have been used for implementing complex algorithms taking advantage of the parallel floating-point units and increasing in this way their throughput in execution time. Although the GPU-based implementations achieves a noticeable speed-up, it is important to point out the following aspects: (a) the GPU-based solution presents bandwidth bottlenecks when all the source data are accessed from global memory or when simultaneous accesses from different threads to memory have to be addressed, (b) GPUs are not tailored for specific applications and commonly are difficult to fine-tune for executing only the operations required by an algorithm, and (c) these integrated circuits operate at high frequencies leading to large power consumption. This is a drawback for embedded system applications.

Due to the high capacity of parallel processing, FPGAs are now being used to accelerate processes such as digital signal processing, image processing, robotics, encryption and decryption, and communication protocol decoding [3, 4]. Robot tasks with higher operational speeds are one of the applications that require high processing capabilities. Therefore, the FPGA implementation of direct kinematics can be an important solution in order to achieve several real-time constraints.

There are two important aspects that must be considered when parallel processing computations using floating-point operators are implemented in FPGA: (a) the tradeoff between the need of reasonable accuracy and the cost of logic area and (b) the choice of a suitable format such that dynamic range is large enough to guarantee that saturation will not occur for a general-purpose application.

Current advances in VLSI technology raised the density integration fast enough for allowing the designers to develop directly in hardware several algorithms commonly implemented on software, thus, obtaining an expressive processing speed-up [5]. Moreover, computation of direct kinematics has high parallelization capabilities, and then, the performance can be improved by implementing it on FPGAs. In this way, a hardware architecture of direct kinematics taking advantage of these features could be useful in robotics applications that require high-speed movements.

In this paper, an FPGA implementation for computing the direct kinematics of a spheric robot with five degrees of freedom (5 *D.o.f*) is described. The hardware architecture considers a floating-point arithmetic, parameterizable by bit-width, allowing the designer to choose a suitable format according to the available hardware resources, accuracy and dynamic range requirements. The main contributions of this work can be summarized as follows: (a) the proposed architecture makes use of several floating-point arithmetic and trigonometric libraries, allowing the performance to be improved in comparison with previous works implementations, in which the floating-point operations are executed in software using DSPs, GPUs, or CPUs (see Section 2), (b) this work presents an error analysis for different bit-width representations (32, 43, and 64 bits), allowing the designer to analyze the tradeoff between the bit-width representation, which directly affects the cost in logic area and the accuracy requirements. Comparison of the processing time between the hardware architecture and a software implementation, using a PowerPC embedded microprocessor, is also presented, (c) the proposed hardware architecture has been developed taking into account a resource constrained methodology, allowing the arithmetic and trigonometric units to be scheduled between different states in order to achieve the lowest execution time according to the available hardware resources.

Section 2 presents the related works covering hardware implementations of direct and inverse kinematics. Section 3 describes the direct kinematics mathematical formulation. Section 4 describes the IEEE-754 standard for the floating-point number representation and a tradeoff analysis of the FPGA implementation of the floating-point operators required by the direct kinematics. Section 5 presents the FPGA implementations and, before concluding, Section 6 presents the synthesis and simulation results.

2. Related Works

Several previous works have presented hardware architectures for implementing only the servo control loop of robotic manipulators, being in these cases the direct and inverse kinematics computed on software [6–9]. In [6, 7] a hardware structure for controlling a SCARA robotic manipulator is developed, relieving the computational cost of general purpose microprocessor by implementing on FPGAs the servo control loop of the robot manipulator.

In [8, 9] a hardware-software codesign for controlling an articulated robot arm is presented, using a NIOS processor for implementing the inverse kinematics. In the same context, in [10] the case study is a neural controller for 3 *D.o.f* parallel robot for milling. This controller is based on neural model of the inverse dynamics of the manipulator, trained on data collected with the use of a computed torque stabilizing controller, and the authors propose that good candidates for hardware implementation are those fragments of an algorithm that can be calculated using only fixed-point operations, due to the fact that they require less FPGA resources and, therefore, are faster whenever are compared with floating-point modules. The other parts of the algorithm (including those working with floating-point) are good candidates to be implemented in an embedded processor. Additionally, [11] proposes hardware solutions based on an ARM processor and fixed-point FPGA modules for computing the trigonometric and square root functions of inverse kinematics. They are based on existing pipeline arithmetic circuits, which employ the CORDIC (Coordinate Rotation Digital Computer) algorithm.

Taking into account previous works, the hardware/software codesign is a very applied approach. In this direction, [12] implements the direct and inverse kinematics of a multifinger hand system, computing the floating-point arithmetic and trigonometric operations (for instance inverse kinematics) using a high-speed DSP processor and several

hardware peripherals (PWM controllers and interfaces for data communication), connected among them throughout a NIOS embedded processor. On the other hand, [13] shows the implementation of the direct kinematic and force feedback algorithm using both CORDIC (for fixed-point) and Xilinx arithmetic library. This work has measured the error of the direct kinematics with respect to a PC implementation, in which numerical analysis shows that Cartesian position error is always below 0.1 mm and the error in orientation is less than 0.001 deg. Although a 32 bit fixed-point data representation is used, details about the same (for example, bit number for fractional part) are not described.

In [14] a hardware implementation of inverse kinematics and a servo controller for a robot manipulator are proposed, allowing the FPGA to compute the high complex computations, such as the transcendental functions, as well as exploring the parallel capabilities of the inverse kinematics. As previous approaches, all hardware embedded operations are performed in fixed-point.

In [15] a generic controller for a multiple-axis motion system (that can be applied to a manipulator) is implemented, which includes several modules such as velocity profile generator, interpolation calculator, inverse kinematics calculator, PID controller, among others. In this case, the trigonometric operations (such as sin, cos, arc tang and arc cosine) have been also implemented by using lookup tables and a fixed-point arithmetic representation.

Additionally, [16] shows the direct and inverse kinematics computation of a 6 *D.o.f* space manipulator using an ARM processor and FPGA coprocessor. Additionally, it considers the hardware implementation of a pipelined CORDIC library in an FPGA device (using fixed-point representation). The experiment shows that the absolute accuracy of the end-effector is less than 3 mm error. In [17] an optimized inverse kinematics approach is proposed for controlling an arm of a virtual human, in which an FPGA device is used for accelerating the computations. In order to improve the performance, a floating to fixed point conversion is performed; however, it imposes a limitation on the dynamic range of the operations.

In summary, almost all previous works describe a hardware-software codesign for implementing the direct and inverse kinematics, in which critical parts are developed in hardware accelerators, developed in FPGAs. On the other hand, these previous works do not consider explicitly floating-point arithmetic for performing the computations using appropriated arithmetic libraries for FPGA, and this can become very important as required by several engineering applications such as robotic manipulators, in which high accuracy and a large dynamic range are important requirements, apart from attending both good performance and low cost in logic area. In this context, taking the importance and dramatical growth of embedded application for automation, control, and robotics areas [18], the full hardware implementation of a kinematics is very important to be researched, in terms of comparing the cost, performance and precision with respective software implementations, specially developed over FPGA embedded processors. Additionally, most of the previous works do not show the error analysis comparing both hardware and software results of the same kinematics, and only few ones measure the error of the direct/inverse kinematics (for instance, [13, 16]) using only fixed-point representation.

3. Background

3.1. Direct Kinematics. A robot manipulator can be described as a series of links, which connect the end effector to the base, with each link connected to the next by an actuated joint. Attaching a coordinate frame to each link, the relationship between two links can be described with a homogeneous transformation matrix—an A matrix. Therefore, a sequence of these A matrices relates the based to the hand of the manipulator (see (1)) [2].

$$^{R}T_{H} = A_1 A_2 \cdots A_{n-1} A_n. \quad (1)$$

Thus, the direct kinematics problem is summarized by finding a homogeneous matrix transformation T that relates the end effector pose (position and orientation) to the based coordinate frame, knowing the angles and displacement between the links and the geometric parameters of the manipulator [2].

The joint coordinate frames have to be assigned by using a rational convention, associated to a zero position (where all joint variables are set to zero). So, it is assumed that for the assignment of coordinate frames to each link the manipulator has to be moved to its zero position. The zero position of the manipulator is the position where all joint variables are zero. This procedure may be useful to check if the zero positions of the model constructed are the same as those used by the controller, avoiding the need of introducing constant deviations to the joint variables (joint positions). Subsequently the z-axis of each joint should be made coincident with the joint axis. This convention is used by many authors and in many robot controllers [2, 19]. For a prismatic joint, the direction of the z-axis is in the direction of motion, and its sense is away from the joint. For a revolute joint, the sense of the z-axis is towards the positive direction of rotation around the z-axis. The positive direction of rotation of each joint can be easily found by moving the robot and reading the joint positions on the robot controller display. According to the [2, 19], the base coordinate frame (robot reference) may be assigned with axes parallel to the world coordinate frame. The origin of the base frame is coincident with the origin of joint 1 (first joint). This assumes that the axis of the first joint is normal to the $x - y$ plane. This location for the base frame coincides with many manufacturers' defined base frame. Afterwards coordinate frames are attached to the link at its distal joint (joint farthest from the base). A frame is internal to the link it is attached to (there is no movements relative to it), and the succeeding link moves relative to it. Thus, coordinate frame i is at joint $i + 1$, that is, the joint that connects link i to link $i + 1$. The origin of the frame is placed as following: if the joint axes of a link intersect, then the origin of the frame attached to the link is placed at the joint axes intersection; if the joint axes are parallel or do

not intersect, then the frame origin is placed at the distal joint; subsequently, if a frame origin is described relative to another coordinate frame by using more than one direction, then it must be moved to make use of only one direction if possible. Thus, the frame origins will be described using the minimum number of link parameters. The x-axis or the y-axis have their direction according to the convention used to parameterize the transformations between links. At this point the homogeneous transformations between joints must have already been determined. The other axis (x or y) can be determined using the right-hand rule. A coordinate frame can be attached to the end of the final link, within the end-effector or tool, or it may be necessary to locate this coordinate frame at the tool plate and have a separate hand transformation. The z-axis of the frame is in the same direction as the z-axis of the frame assigned to the last joint $(n-1)$. The end-effector or tool frame location and orientation is defined according to the controller conventions. The homogeneous transformations between joints follow the Denavit-Hartemberg convention (D-H), making two consecutive links related through 4 basic transformations that only depend on the link geometry [2]. The transformations are:

(1) a rotation about the z_{n-1} axis by the angle between links (θ_n),

(2) a translation along the z_{n-1} axis of the distance between the links (d_n),

(3) a translation along the x_n axis (rotated x_{n-1} axis) of the length of the link (l_n),

(4) a rotation about the x_n axis of the twist angle (α_n).

$$A_n = R(z,\theta_n)T(0,0,d_n)T(l_n,0,0)R(x,\alpha_n). \quad (2)$$

θ_n, d_n, l_n, and α_n represent the D-H parameters of the link n. Identifying these parameters, the A matrix that relates two consecutive links can be obtained. Multiplying the sequence of the A matrices (see (1)) a homogeneous matrix transformation T is obtained, solving the direct kinematics problem.

3.2. Direct kinematics for a Spherical Robot Manipulator. The robot described in this paper has a spherical topology with 5 degrees of freedom, with two rotational joints in the base followed by a prismatic joint and two rotational joints located in manipulator hand. The first three joints are responsible for the wrist position and the two last ones are responsible for the hand or tool orientation. Figure 1 shows the proposed robot topology.

Figure 2 shows the assigned coordinate frame to each link following the convention D-H for the manipulator depicted in Figure 1. Table 1 presents the D-H parameters.

Table 1 can be used for computing the matrix that relates two consecutive axis frames of the robot, and finally, by multiplying the sequence of the A matrices, a homogeneous matrix transformation T is obtained. The matrix T can be formulated as shown in Equation (3), where the entries

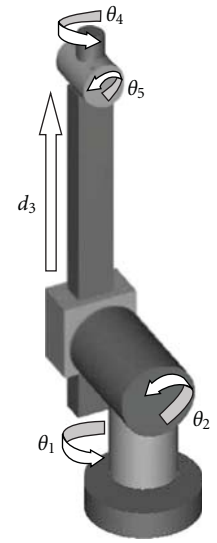


Figure 1: Five degrees of freedom Robot Manipulator.

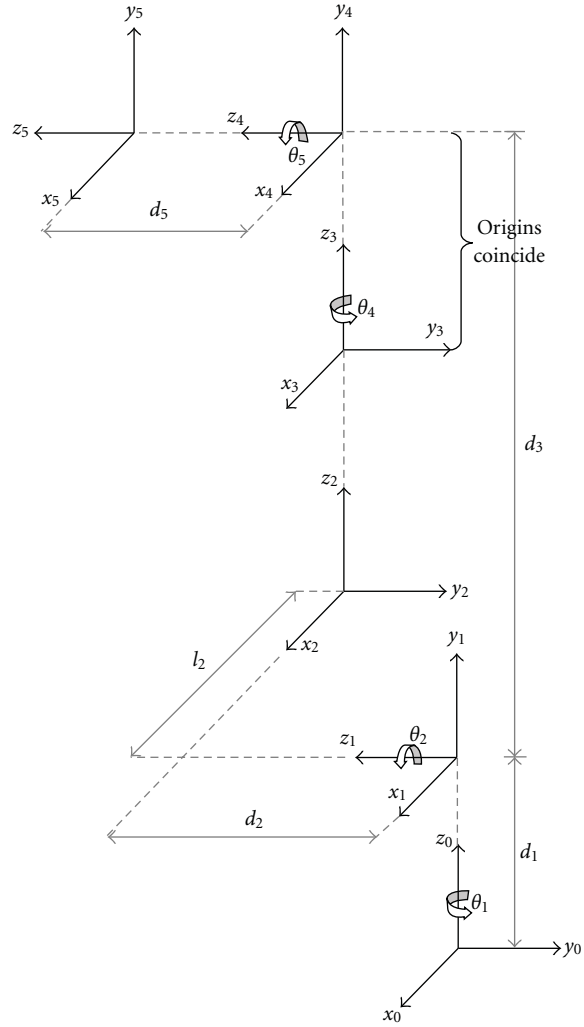


Figure 2: Assignment of coordinate frames.

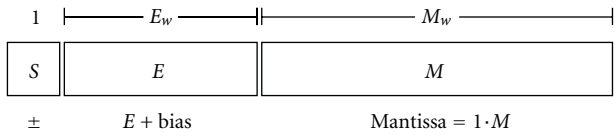


Figure 3: IEEE-754 format.

Table 1: Denavit-Hartenberg parameter of robot manipulator.

Joint variable	Angle θ_n	Displacement d_n	Length l_n	Twist α_n
θ_1	θ_1	d_1 (106 mm)	0	90
θ_2	$\theta_1 + 90$	d_2 (130 mm)	l_2 (0 mm)	−90
d_3	0	d_3	0	0
θ_4	θ_4	0	0	90
θ_5	θ_5	d_5 (0 mm)	0	0

of T follow a notation that represents, for example, y_x as the projection of the y axis of the last coordinate frame in the x axis of the base frame, and p_x is the x component of the origin coordinates of the last frame represented in the base frame.

$$T = \begin{bmatrix} x_x & y_x & z_x & p_x \\ x_y & y_y & z_y & p_y \\ x_z & y_z & z_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

By using the D-H notation, the matrix T is summarized by the (4) to (15). In these equations the notation $C_i = \cos(\theta_i)$ and $S_i = \sin(\theta_i)$ was used.

$$x_x = (-C_1S_2C_4 - S_1S_4)C_5 - C_1C_2S_5, \tag{4}$$

$$x_y = (-S_1S_2C_4 + C_1S_4)C_5 - S_1S_2S_5, \tag{5}$$

$$x_z = C_2C_4C_5 - S_2S_5, \tag{6}$$

$$y_x = -(-C_1S_2C_4 - S_1S_4)S_5 - C_1C_2C_5, \tag{7}$$

$$y_y = -(-S_1S_2C_4 + C_1S_4)S_5 - S_1C_2C_5, \tag{8}$$

$$y_z = -C_2C_4S_5 - S_2C_5, \tag{9}$$

$$z_x = -C_1S_2S_4 + S_1C_4, \tag{10}$$

$$z_y = -S_1S_2S_4 - C_1C_4, \tag{11}$$

$$z_z = C_2S_4, \tag{12}$$

$$p_x = -C_1C_2d_3 - C_1S_2l_2 + S_1d_2, \tag{13}$$

$$p_y = -S_1C_2d_3 - S_1S_2l_2 - C_1d_2, \tag{14}$$

$$p_z = -S_2d_3 + l_2C_2 + d_1. \tag{15}$$

It can be observed in the (4) to (15) that some intermediate computations are found in two equations. For example, the intermediate computation $(C_1C_2C_4 - C_1S_2S_4)$ is presented in the (4) and (7). In this way, the intermediate computation should be computed only once, saving processing time.

Although parameter l_2 is equal to zero, it will be taken into account for future calibration purposes.

4. Description and Analysis of the Floating-Point Operators

4.1. IEEE-754 Format. The IEEE-754 format [20] is a floating-point number representation characterized by three components: a sign S, a biased exponent E with E_w bit-width and a mantissa M with M_w bit-width as shown in Figure 3. A zero sign bit denotes a positive number and a one sign bit denotes a negative number. A constant (bias) is added to the exponent in order to make the exponent's range nonnegative and the mantissa represents the magnitude of the number. The standard also includes extensive recommendations for advanced exception handling, additional operations (such as trigonometric functions), and expression evaluation for achieving reproducible results.

This standard allows the user to work not only with the 32-bit single precision and 64-bit double precision, but also with a suitable precision according to the application. This is suitable for supporting variable precision floating-point operations [21].

4.2. An Architectural Approach for Floating-Point Operators. As stated in (4) to (15), the direct kinematics computation of the spherical robot described above requires the computation of add/sub and multiplication arithmetic operators, as well as the sine and cosine transcendental functions. Previous works covering hardware implementations of floating-point transcendental functions on FPGAs are based on CORDIC algorithms using simple and double precision formats [22–24]. In [25] a HOTBM method for computing trigonometric functions in FPGAs is presented, achieving a reduced area and high performance without sacrificing accuracy. However, these works have not received enough attention on the cost in area associated with the precision level, as well as, their respective error analysis.

In this work, the Taylor series expansion has been implemented on FPGAs for computing, in an integrated approach, the sine, cosine, and arctangent functions, using a floating-point arithmetic based on the IEEE-754 standard. Our previous results point out that the Taylor expansion approach has a lower execution time and a lower cost in logic area than the CORDIC-based solution. However, the CORDIC algorithm presents a better performance in terms of precision [26–28]. As will be explained below, this work focuses on solving the resource and timing constrains

(see Section 4); therefore, the choice of using a Taylor expansion approach for computing the floating-point trigonometric operators can be justified. This work considers accuracy as a design criterion and provides an error analysis associated to the bit-width representation and the area cost, as well as, an error analysis associated to the number of iterations of the Taylor series. These results are useful for choosing a suitable format for representing real numbers according to general-purpose applications.

The Taylor series of sine, cosine and arctangent functions, around 0 are given, respectively, as

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \tag{16}$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \tag{17}$$

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots. \tag{18}$$

The factors $1/n!$ and $1/n$ are precomputed and stored in a ROM avoiding additional operations. Before the first iteration, the x^2 term is computed and stored. In each iteration one floating-point addition of the accumulated approximation with the ith term is necessary. Additionally, for computing the ith term two floating-point multiplications are needed: (1) for computing either x^2x^{2i-1} or $x^2x^{2(i-1)}$ for sine/arctangent or cosine, respectively, and (2) for computing either $1/(2i+1)!x^{2i+1}$ for sine, $1/(2i+1)x^{2i+1}$ for arctangent and $1/(2i)!x^{2i}$ for cosine.

Figure 4 shows the proposed architecture for the Taylor series expansion. This approach was achieved using only one $\text{FP}_{\text{multiplier}}$, an $\text{FP}_{\text{add/sub}}$, a Finite State Machine (FSM), and three ROMs for storing the precomputed $1/n!$ and $1/n$ factors. The op signal chooses between a sin, cos or atan functions. The FSM synchronizes the $\text{FP}_{\text{multiplier}}$, $\text{FP}_{\text{add/sub}}$ units and alternates the $\text{op}^{+/-}$ signal. At the first iteration, the factor x^2 is computed and stored in a register and fed back to the FSM. Afterward, the precomputed coefficients are selected in order to compute the current term either $x^{2n-1}/(2n-1)!$ for sine, $x^{2n-1}/(2n-1)$ for arctangent or $x^{2n}/2n!$ for cosine. The add/sub unit computes and accumulates both the addition or subtraction operations. After N iterations, the ready signal indicates a valid output.

Table 2 shows the Mean Square Error (MSE) results of the arithmetic and transcendental functions architectures for different bit-width representations (24, 32, 43, and 64 bits), using the Matlab results as statistical estimators. The Taylor architecture uses 5 nonzero powers (5 terms of the series expansion). As expected, the best results were achieved using the double precision format (64 bits); however, one can expect a large cost in logic area.

Table 3 presents the MSE and latency in clock cycles for the Taylor series expansion, using a simple precision format (32 bits). It can be observed that the smaller error achieved by the $\text{FP}_{\text{Taylor}}$ core is over E^{-15}, for five powers (x^{11} polynomial degree). With more than five powers in the series expansion, the error due to the Runge's phenomenon is increased [29] when using polynomial interpolation with polynomials of high degree.

Table 2: MSE of the floating-point operators.

FP-Core	24(6, 17)	32(8, 23)	43(11, 31)	64(11, 52)
Add/sub	$2.27E-7$	$2.76E-11$	$3.24E-15$	$1.26E-17$
Multiplier	$9.53E-4$	$1.53E-7$	$1.49E-11$	$5.87E-16$
Taylor sin/cos	$6.31E-9$	$8.80E-9$	$6.26E-9$	$1.94E-16$
Taylor atan	0.073	0.0014	0.015	0.015

Table 3: MSE and Latency of the FP_{Taylor} architecture.

Powers	$\sin(x)$ $[-\pi/2\ \pi/2]$	$\cos(x)$ $[-\pi/2\ \pi/2]$	$\text{atan}(x)$ $[-100\ 100]$	Elapsed time clock cycles
2	$1.59E-03$	$3,52E-02$	0.00137	11
3	$1.56E-06$	$5,58E-05$	0.00136	15
4	$7.01E-13$	$3.62E-11$	0.00136	19
5	$3.13E-15$	$1.15E-14$	0.00136	23
6	$2.13E-14$	$5.91E-15$	0.00136	27

Table 4: Synthesis results. Virtex2 XC2VP30.

Bit-width (Exp, Man)	Slices (13696)	LUTs (27392)	Mult18 × 18 (136)	Freq (MHz)
32(8, 23)	10528	20017	48	54.83
43(11, 31)	14589	27807	48	48.31
64(11, 52)	34031	64687	180	39.51

Table 5: Synthesis results. Virtex5 XC5VLX110T.

Bit-width (Exp, Man)	Slices (69120)	LUTs (69120)	DSP48Es (64)	Freq (MHz)
32(8, 23)	4051	14457	24	83.61
43(11, 31)	5349	19262	48	66.22
64(11, 52)	7901	98755	111	63.93

5. The FPGA Implementations

The FPGA implementation of the direct kinematics is carried out using Time-Constrained Scheduling (TC) [30], as showed in Figure 5. In fact, the circuit explicitly implements (4) to (15). To do that, some strategies can be used in order to allocate the respective hardware resources, achieving a previously defined cost function, which can represent time or/and resource restrictions. Time constrained and resource constrained scheduling algorithms are well-known techniques for leading with obtaining a suitable time execution and allocating the available hardware resources (for instance, arithmetic operators). For achieving the circuit depicted in Figure 5 (from (4) to (15)), only resource restrictions were imposed, where 4 multipliers and two add/sub units were used, apart from 8 trigonometric units. With the proposed scheduling the lowest time execution was tried to be accomplished. In contrast, a time constrained scheduling would need more arithmetic units for accomplishing

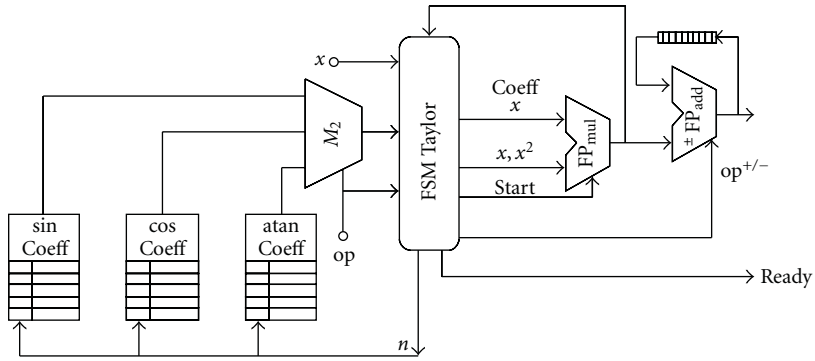


Figure 4: IEEE-754 format.

a predefined restriction in execution time. Anyway, this kind of algorithms does not guarantee an optimal solution [30].

The control of the overall architecture was achieved by a Finite State Machine (FSM), and the overall architecture was described in VHDL using the Xilinx ISE 10.1 development tool, making use of the floating-point arithmetic units. Eight cores for computing transcendental functions were required: four for computing the sin function and the other four for computation the cos function. The necessary addition and subtraction operations were achieved by applying the $FP_{add/sub}$ and $FP_{multiplier}$ cores presented in [28] (see Section 3). For computing the sin and cos functions, a Taylor series expansion was implemented, using a $FP_{add/sub}$ and a $FP_{multiplier}$ cores, with a similar architecture to the presented in [26]. The $FP_{Taylorcos}$ and $FP_{Taylorsin}$ were achieved using 10 and 11 nonzero powers for cosine and sine respectively and an elapsed time of 23 clock cycles for cosine and 26 for sine.

It can be noticed that the orientation and position results are calculated from the following steps: in step 3 the x_z and y_z, in step 4 the parameter p_z, in step 5 the z_z and p_x, in step 6 the p_y parameter is computed, in step 7 the z_x and z_y, in step 10 the x_x and y_x, and finally in step 11 the x_y and y_y (see Figure 5).

In order to compare the performance of the proposed architecture, the direct kinematics was also implemented in software using a C code language running on an FPGA embedded PowerPC processor, using the Xilinx Platform Studio development tool. This approach allows the designer for having a just comparison between both hardware and software developments, which are running in the same environment (namely, the same FPGA with the same clock). To do that, two hardware peripherals were added to the PLB bus of the PowerPC processor: the first one for counting the clock cycles and the second one for accomplishing a RS-232 serial communication to a Matlab environment. The counter is used for measuring the processing time in software by means of enable and stop ports (both addressed by the PLB bus), which are controlled by specific software instructions. Figure 6 shows the connection between the PowerPC processor and the hardware peripherals.

6. Results

6.1. Synthesis Results. Synthesis results are shown in Table 4 for the Xilinx Virtex2 family (chip xc2vp30) using the ISE 10.1 development tool. It can be observed that the cost and performance of the architecture are presented in Figure 5 for different bit-width (exponent and mantissa), including both the simple precision and double precision (IEEE-754 standard). As expected, large bit-width representations are more expensive in terms of area if compared with small bit-width representations. Additionally, the performance is better when using small bit-widths.

It can be seen that the 43 and 64 bit-widths representations exceeded the hardware resources available in the specific Virtex2 FPGA (see Table 4). However, it is important to take into account that the xc2vp30 is not the largest device of the Xilinx Virtex2 FPGA family. Modern FPGAs devices have a large number of configurable logic blocks (CLB), dedicated DSP blocks, among other facilities which are suitable for mapping complex algorithms directly in hardware. Table 5 shows the cost in logic area of the same architecture using a Virtex5 FPGA (chip xc5vlx110t). It is important to take into account that the Virtex5 FPGA family uses embedded DSP48Es blocks, which considers 18×25 bits multipliers. It can be observed that the operational frequency has been increased; however the double precision representations exceed the available number of LUTs. However, as will be explained below, simulation results point out that the single precision format (32 bits) achieves similar error results to large bit-width representations. Therefore, one can conclude that the 32 bit implementation is suitable in terms of hardware resources consumption and error associated for computing the direct kinematics of a 5 *D.o.f* spheric robot.

6.2. Simulation Results. The implemented direct kinematics hardware architecture (see Figure 5) was simulated using the ModelSim 6.3 g simulator tool. A simulation environment to validate the architecture behavior was developed in Matlab. This developed tool created the floating-point test vectors for different bit-widths. The binary floating-point results were analyzed in the simulator environment in order to calculate the Mean Square Error (MSE) of the implemented

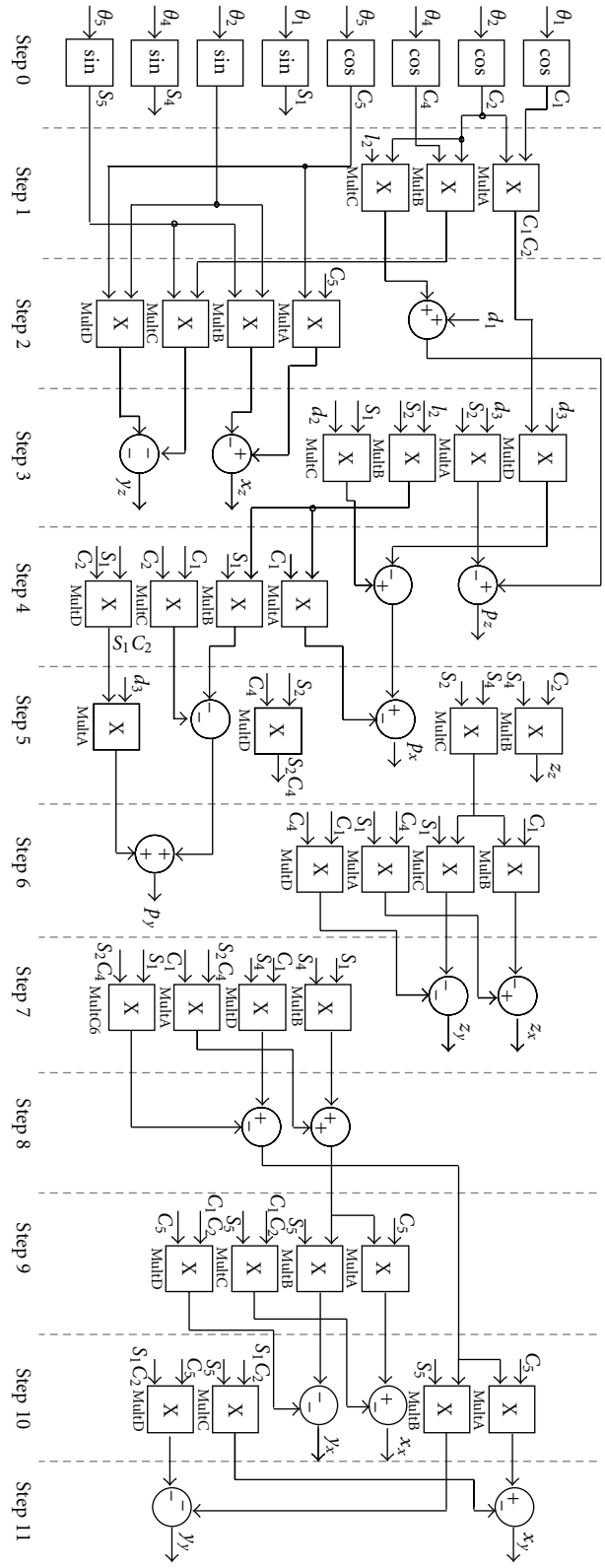


Figure 5: Hardware architecture for computing the direct kinematics using FSM.

Table 6: MSE for different bit-width representation.

Bit-width	axis	x	y	z	p
32(8, 23)	x	$3.61E-12$	$1.44E-11$	$1.75E-14$	$5.42E-06$
	y	$9.08E-13$	$3.55E-12$	$1.06E-14$	$1.33E-06$
	z	$1.44E-11$	$3.63E-12$	$4.46E-12$	$2.07E-10$
43(11, 31)	x	$3.63E-12$	$1.45E-11$	$1.62E-14$	$5.44E-06$
	y	$8.88E-13$	$3.58E-12$	$3.28E-16$	$1.34E-06$
	z	$1.45E-11$	$3.64E-12$	$4.45E-12$	$7.07E-11$
64(11, 52)	x	$3.63E-12$	$1.45E-11$	$1.62E-14$	$5.44E-06$
	y	$8.88E-13$	$3.58E-12$	$3.29E-16$	$1.34E-06$
	z	$1.45E-11$	$3.64E-12$	$4.45E-12$	$7.07E-11$

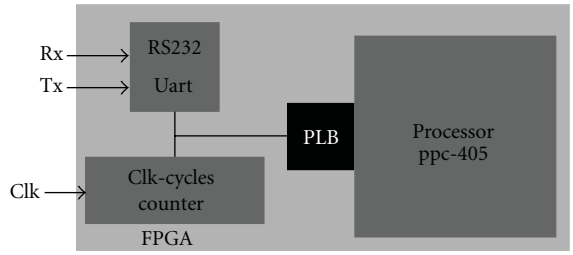


Figure 6: FPGA Architecture for Software Implementation.

Table 7: Latency for each variable.

Stage	Variable	Latency
3	x_z, y_z	43
4	p_z	46
5	z_z, p_x	49
6	p_y	52
7	z_x, z_y	55
10	x_x, y_x	64
11	x_y, y_y	67

architecture, for which the set of Matlab results in double precision arithmetic were used as a statistical estimator.

For computing the MSE of the hardware architecture, 100 input values were used. The input ranges were -90° to 90° for the angles θ_1, θ_4, and θ_5, -35° to 20° for the angle θ_2, and 160 mm to 760 mm for the D_3 parameter.

Table 6 shows the MSE for different bit-width representations. Notice that the magnitude order of MSE does not have significant changes. Simulations using a bit-width representation smaller than a 32 bit-width representation did not present satisfactory results, due to saturation in the floating-point operators.

The software implementation (see Figure 6) was validated using a simulation environment developed in Matlab and the serial RS232 interface for sending the input values and receiving the output values (position, orientation, and time processing in clock cycles). The aim of this simulation is to compare the processing time of the direct kinematics between software and hardware approaches and not to compare the error in the computations.

The elapsed time using the hardware architecture for each variable and each step is shown in Table 7. The overall direct kinematics is computed in 67 clock cycles. According to synthesis results, the maximum operational frequency is around 54 MHz for a single precision format, thus, all the computations are performed in 1.24 μs. The software implementation has a processing time of 1.61036 ms for all computations of the direct kinematics in floating-point using a single precision. For comparison purposes we have only used the obtained operational frequency on the VirtexII FPGA family (54 MHz) due to the fact that this device has a PowerPC embedded processor. It can be observed that the hardware architecture is over 1298 times faster than the software implementation, which means a considerable speed-up in the processing time of the direct kinematics. These elapsed time results can be dependent on the operation scheduling used in the architecture.

7. Conclusion

An FPGA implementation of the direct kinematics of a spherical robot manipulator using floating-point units was presented. The proposed architecture was designed using a Time-Constrained Scheduling, using 8 cores for computing transcendental functions (sine and cosine), and sharing 4 multiplier floating-point cores and 2 addition/subtraction floating-point cores for computing the arithmetical operations. Synthesis results show that the proposed hardware architecture for direct kinematics of robots is feasible in modern FPGAs families, in which there are plenty of logic elements available.

The overall computations were successfully performed and were validated using the Matlab results as a statistical estimator. The computation time to implement the direct kinematics in hardware is over 1.24 μs, whereas the same formulation implemented in software using a PowerPC processor needs 1.61 ms, obtaining a considerable speed-up in the processing time.

As future works, a hardware architecture for computing the inverse kinematics of the proposed robot manipulator will be implemented using the floating-point cores currently developed, such as division, square root, and arctangent functions. In addition, we intend to perform a performance

comparison of the proposed hardware architecture with a multicore embedded processor approach.

References

[1] A. Y. Zomaya, "Parallel processing for robot dynamics computations," *Parallel Computing*, vol. 21, no. 4, pp. 649–668, 1995.

[2] P. McKerrow, *Introduction to Robotics*, Adison-Wesley, New York, NY, USA, 1991.

[3] S. Kilts, *Advanced FPGA Design*, John Wiley & Sons, Hoboken, NJ, USA, 2007.

[4] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, Berlin, Germany, 2001.

[5] R. Andraka, "Survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 191–200, February 1998.

[6] X. Shao, S. Dong, and J. K. Mills, "A new motion control hardware architecture with FPGA-based IC design for robotic manipulators," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '06)*, pp. 3520–3525, May 2006.

[7] X. Shao, D. Sun, and J. K. Mills, "Development of an FPGA-basedmotion control ASIC for robotic manipulators," in *Proceedings of the IEEE International Intelligent Control and Automation*, pp. 8221–8225, 2006.

[8] Y. S. Kung and G. S. Shu, "Development of a FPGA-based motion control IC for robot arm," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT '05)*, pp. 1397–1402, December 2005.

[9] Y. S. Kung and G. S. Shu, "Design and implementation of a control IC for vertical articulated robot arm using SOPC technology," in *Proceedings of the IEEE International Conference on Mechatronics (ICM '05)*, pp. 532–536, July 2005.

[10] M. Petko and G. Karpiel, "Hardware/software co-design of control algorithms," in *Proceedings of the IEEE International Conference on Mechatronics and Automation (ICMA '06)*, pp. 2156–2161, June 2006.

[11] R. Wei, M. Jin, J. J. Xia, Z. Xie, and H. Liu, "Reconfigurable parallel VLSI co-processor for spacerobots using FPGA," in *Proceedings of the International Conference on Robotics and Biomimetics (ROBIO '06)*, pp. 374–379, 2006.

[12] R. Wei, X. Gao, M. Jin et al., "An FPGA based hardware architecture for HIT/DLR hand," in *Proceedings of the International Conference on Intelligent Robots and Systems*, pp. 523–528, 2005.

[13] S. Galvan, D. Botturi, and P. Fiorini, "FPGA-based controller for haptic devices," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '06)*, pp. 971–976, October 2006.

[14] Y.-S. Kung, K.-H. Tseng, C.-S. Chen, H.-Z. Sze, and A-P. Wang, "FPGA-implementation of inverse kinematics and servo controller for robot manipulator," in *Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO '06)*, pp. 1163–1168, 2006.

[15] J. U. Cho, Q. N. Le, and J. W. Jeon, "An FPGA-based multiple-axis motion control chip," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 3, pp. 856–870, 2009.

[16] Z. Yili, S. Hanxu, J. Qingxuan, and S. Guozhen, "Kinematics control for a 6-DOF space manipulator based on ARM Processor and FPGA Co-Processor," in *Proceedings of the International Conference on Industrial Informatics (INDIN '08)*, pp. 129–134, 2008.

[17] D. Hildenbrand, H. Lange, F. Stock, and A. Koch, "Efficient inverse kinematics algorithm based on conformal geometric algebra—using reconfigurable hardware," in *Proceedings of the 3rd International Conference on Computer Graphics Theory and Applications (GRAPP '08)*, pp. 1–8, Springer, 2008.

[18] R. Hartenstein, "Why we need reconfigurable computing education," in *Proceedings of the IEEE Workshop on Reconfigurable Computing Education*, pp. 1–4, 2006.

[19] R. P. Paul, *Robot Manipulators—Mathematics, Programming, and Control*, MIT Press, Boston, Mass, USA, 1981.

[20] "IEEE standard for binary floating-point arithmetic," ANSI/IEEE Std 754-1985, August 1985.

[21] X. Wang, *Variable precision floating-point divide and square root for efficient FPGA implementation of image and signal processing algorithms*, Ph.D. dissertation, Northeastern University, 2007.

[22] J. Zhou, Y. Dong, Y. Dou, and Y. Lei, "Dynamic configurable floating-point FFT pipelines and hybrid-mode CORDIC on FPGA," in *Proceedings of The International Conference on Embedded Software and Systems (ICESS '08)*, pp. 616–620, 2008.

[23] J. Valls, M. Kuhlmann, and K. K. Parhi, "Evaluation of CORDIC algorithms for FPGA design," *Journal of VLSI Signal Processing*, vol. 32, no. 3, pp. 207–222, 2002.

[24] F. E. Ortiz, J. R. Humphrey, J. P. Durbano, and D. W. Prather, "A study on the design of floating-point functions in FPGAs," *Field Programmable Logic and Applications*, vol. 2778, pp. 1131–1135, 2003.

[25] J. Detrey and F. De Dinechin, "Floating-point trigonometric functions for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 29–34, August 2007.

[26] D. M. Muñoz, D. F. Sanchez, C. H. Llanos, and M. Ayala-Rincón, "Tradeoff of FPGA design of floating-point transcendental functions," in *Proceedings of the IEEE International Conference on Very Large Scale Integration*, pp. 1–4, 2009.

[27] D. M. Muñoz, D. F. Sanchez, C. H. Llanos, and M. Ayala-Rincón, "FPGA based floating-point library for CORDIC algorithms," in *Proceedings of the 6th Southern Programmable Logic Conference (SPL '10)*, pp. 55–60, March 2010.

[28] D. F. Sánchez, D. M. Muñoz, C. H. Llanos, and M. Ayala-Rincón, "Parameterizable floating-point library for arithmetic operations in FPGAs," in *Proceedings of the 22nd ACM Symposium on Integrated Circuits and Systems Design (SBCCI '09)*, pp. 253–254, 2009.

[29] C. Runge, "ber empirische funktionen und die interpolation zwischen quidistanten ordinaten," *Zeitschrift für Mathematik und Physik*, vol. 46, pp. 224–243, 1901.

[30] D. Gajski, *Principles of Digital Design*, Prentice Hall, New York, NY, USA, 1997.