International Journal of Reconfigurable Computing

Selected Papers from ReConFig 2008

Guest Editors: Lionel Torres and Cesar Torres

Selected Papers from ReConFig 2008

International Journal of Reconfigurable Computing

Selected Papers from ReConFig 2008

Guest Editors: Lionel Torres and Cesar Torres

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 869329, 2 pages
doi:10.1155/2009/869329

Editorial
Selected Papers from ReConFig 2008

Lionel Torres[1] and Cesar Torres[2]

[1] *Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), UMR CNRS 5506, University of Montpellier, 34090 Montpellier, France*
[2] *Information Technology Laboratory (LTI), CINVESTAV, 87130 Tamaulipas, Mexico*

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

Received 31 December 2009; Accepted 31 December 2009

The fourth edition of the International Conference on Reconfigurable Computing and FPGAs (ReConFig 2008) was held in Cancun, Mexico, from December 3 to 5, 2008. 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 2008 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 12 articles in this issue. We begin the special issue with two papers that extend across the digital signal processing domain. The paper by G. Caffarena et al. "Architectural synthesis of fixed-point datapaths using FPGAs" addresses the automatic synthesis of fixed-point datapaths by combining a multiple wordlength approach with a wise mapping of operations to the embedded FPGA resources leading improvements around 50%. In "Pipeline FFT architectures optimized for FPGAs," by B. Zhou et al. optimized implementations of two pipeline FFT processors using general optimization and architecture-specific optimizations and different rounding schemes to achieve better performances with lower resources than previous works.

Four papers are within the broad area of automated design and multiprocessors systems on chips. The paper by H. Ishebebi et al. presents an automated design approach for multiprocessor systems on FPGAs which customizes architectures for parallel programs by simultaneously solving the problems of task mapping, resource allocation, and scheduling. Results show performance improvements by three orders of magnitude highlighting the potential for solving difficult instances of automated synthesis. In a second paper, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," by F. Redaelli et al. proposes an exact ILP formulation for the task scheduling problem on a 2D dynamically and partially reconfigurable architecture taking into account physical constraints of the target device. Also, this work proposes a reconfiguration-aware heuristic scheduler in an HW/SW codesign method reducing the schedule length of application by a factor of 2 in the best case. M. Saldaña et al., in "A message-passing hardware/software co-simulation environment for reconfigurable computing systems," address the need for cosimulating a complete heterogeneous application in high-performance configurable computers by providing a message-passing simulation framework to simulate and develop an interface enabling an MPI-based approach to exchange data between $\times 86$ processors and hardware engines embedded in FPGAs. The development, exploration, and validation methodology of real-time operating systems for reconfigurable Systems-on-Chip is covered in "OveRSoC: a framework for the exploration of RTOS for RSoC platforms," by Benoît Miramond. They present a method for the distribution of operating services on such platforms with an accurate modeling of the dynamic and deterministic behavior of applications and RTOs.

Three articles are presented in the area of algorithms and implementations mapped on reconfigurable hardware. J. H. B. Zambrano et al. in "Parallel processor for 3D recovery from optical flow" present a parallel processor for 3D recovery from optical flow under real-time constraints. The

presented design exhibits a good trade-off between hardware resource usage, image resolution, and the processing speed. S. Lloyd et al. in "Hardware accelerated sequence alignment with traceback" present a space-efficient, global sequence alignment algorithm and 256 processing elements architecture to speedup sequence alignment used in molecular biology and biomedical applications. Performance gains over 300 times that of a desktop computer demonstrated showing the potential to analyze genetic data in a timely manner. B. Girau et al. in "Reaction-diffusion and chemotaxis for decentralized gathering on FPGAs," describe the feasibility of gathering multiple computational units by means of decentralized and simple local rules by a stochastic model and discuss a fully parallel hardware implementation so as to study its ability to provide a massively distributed computational model for decentralized gathering.

The next article, by Y.-H.E. Yang et al., "Software toolchain for large-scale RE-NFA construction on FPGA," presents a toolchain which automates the construction and optimizations of regular expression matching engines (REMEs) on FPGA. The automated REME optimizations include centralized character classifications, multicharacter matching, and staged pipelining. Also, authors designed a benchmark generator which can produce RE-NFAs with configurable pattern complexity parameters, including state count, state fan-in, loop-back, and feed-forward distances. A. Mendon et al. in "A hardware filesystem implementation with multi-disk support," describes a file system implementation with four basic operations (open, read, write, and delete) and the potential of improving the performance of data-intensive applications by connecting secondary storage directly to FPGA compute accelerator. Last but not least, the article "Analysis and enhancement of random number generator in FPGA based on oscillator rings" by Knut Wold addresses the fast implementation of a true random number generator into an FPGA device. The proposed implementation passes the standard statistical test without postprocessing showing good quality randomness characteristics. The throughput of the TRNG is 100 Mbps and the resources used in the FPGA are less than 100 logic elements in an Altera Cyclone II FPGA.

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 2008 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
Cesar Torres

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 703267, 14 pages
doi:10.1155/2009/703267

Research Article

Architectural Synthesis of Fixed-Point DSP Datapaths Using FPGAs

Gabriel Caffarena,[1] Juan A. López,[1] Gerardo Leyva,[2] Carlos Carreras,[1] and Octavio Nieto-Taladriz[1]

[1] *Departamento de Ingeniería Electrónica, Universidad Politécnica de Madrid, Ciudad Universitaria s/n, 28040 Madrid, Spain*
[2] *Departamento de Sistemas Electrónicos, Universidad Autónoma de Aguascalientes, Ciudad Universitaria s/n, 20100 Aguascalientes, Mexico*

Correspondence should be addressed to Gabriel Caffarena, gabriel@die.upm.es

Received 25 February 2009; Accepted 28 August 2009

Recommended by Cesar Torres

We address the automatic synthesis of DSP algorithms using FPGAs. Optimized fixed-point implementations are obtained by means of considering (i) a multiple wordlength approach; (ii) a complete datapath formed of wordlength-wise resources (i.e., functional units, multiplexers, and registers); (iii) an FPGA-wise resource usage metric that enables an efficient distribution of logic fabric and embedded DSP resources. The paper shows (i) the benefits of applying a multiple wordlength approach to the implementation of fixed-point datapaths and (ii) the benefits of a wise use of embedded FPGA resources. The use of a complete fixed-point datapath leads to improvements up to 35%. And, the wise mapping of operations to FPGA resources (logic fabric and embedded blocks), thanks to the proposed resource usage metric, leads to improvements up to 54%.

1. Introduction

This paper addresses the architectural synthesis (AS) of Digital Signal Processing (DSP) algorithms implemented using modern FPGAs. High levels of optimization are achieved by means of the use of Multiple wordlength (MWL) fixed-point descriptions of the algorithms and also the use of both LUT-based and embedded FPGA resources. The former reduces implementation costs notably, and the latter minimizes area in FPGA implementations.

The MWL implementation of fixed-point DSP algorithms [1–4] has proved to provide significant cost savings when compared to the traditional uniform wordlength (UWL) design approach. The introduction of MWL issues in AS increases optimization complexity, but it opens the door to significant cost reductions [2, 3, 5, 6].

FPGA devices have been extensively used in the implementation of DSP algorithms, especially due to the recent introduction of specialized embedded blocks (i.e., memory blocks, DSP blocks, etc.). Traditional approaches to estimate FPGA resource usage do not apply to modern FPGAs, which present a heterogeneous architecture composed of both logic fabric and embedded blocks, since they only account for lookup table- (LUT-) based resources [7]. This situation calls for new resource usage metrics that can be integrated as part of automatic synthesis techniques to fully exploit the possibilities that embedded resources offer [8–10].

The current approaches to perform MWL-oriented architectural synthesis are not tuned to modern FPGAs [2, 3] or an efficient distribution between logic fabric and specialized embedded blocks is not applied [11, 12]. Also, the resource set used during the optimization process does not include the multiplexers necessary to transfer data from memory elements to arithmetic resources.

The main contributions of this paper are the following.

(i) The presentation of a novel resource usage metric that guarantees minimum resource usage for heterogeneous FPGA implementations if integrated within an optimization framework.

(ii) The presentation of an architectural synthesis procedure tuned to fixed-point implementations, that handles a complete datapath (functional units, multiplexers, and registers).

(iii) A novel strategy for fixed-point data multiplexing.

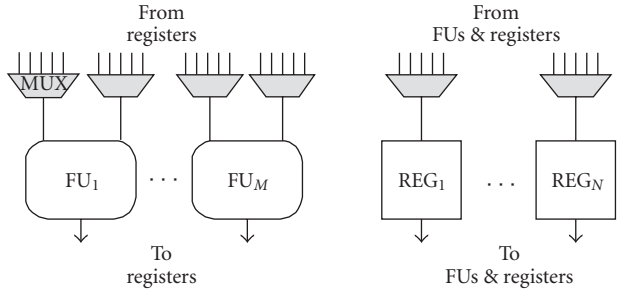


Figure 1: Datapath architecture: FUs, registers, and multiplexers.

The paper is divided as follows. In Section 2, the architectural synthesis of DSP datapaths using multiple wordlength systems and modern FPGAs is introduced. Section 3 deals with the implementation results from synthesizing several DSP benchmarks for different latency constraints and output noise constraints. Finally, in Section 4, the conclusions are drawn.

2. Synthesis of Fixed-Point Datapaths

2.1. Formal Description. This work focuses on the time-constrained resource minimization problem [13]. The notation used is based on [13], and it is similar to that in [2, 4, 6].

Given a sequencing graph $G_{\text{SEQ}}(O, S)$, a maximum latency λ, and a set of resources R (e.g., functional units R_{FU}, registers R_{REG}, and steering logic R_{MUX}), it is the goal of AS to find the time step when each operation is executed (*scheduling*), the types and number of resources forming R (*resource allocation*), and the binding between operations and variables to functional units and registers (*resource binding*) that comply with the constraints, while minimizing cost (i.e., area). As a result, a datapath able to compute the algorithm's operations (see Figure 1) as well as the required control logic is generated.

$G_{\text{SEQ}}(O, S)$ is a formal representation of a single iteration of an algorithm, where O is the set of operations and $S \subset O \times O$ is the set of signals that determine the data flow. We consider $O = O_M \cup O_G \cup O_A \cup O_D \cup O_I \cup O_O$ composed of typical DSP operations: multiplications, gains (multiplication by a constant value), additions, unit delays, and input and output nodes. Signals are in two's complement fixed-point format, defined by the pair (p, n), where p is number of integer bits [4] and n is the wordlength of the signal not including the sign bit (see Section 2.5). The values of the couples (p, n) have been computed during a previously performed wordlength optimization (WLO) [1, 14–16]. See Section 2.5 for more information about the wordlength optimization process.

Functional units (R_{FU}) are in charge of executing the set of operations from O. Registers (R_{REG}) store the data produced by FUs and some intermediate values. Finally, steering logic (R_{MUX}) interconnects FUs and registers by means of multiplexers. The set of functional units $R_{\text{FU}} = R_{A_{\text{LUT}}} \cup R_{M_{\text{LUT}}} \cup R_{M_{\text{EMB}}}$ is composed of LUT-based adders, LUT-based generic multipliers, and embedded multipliers. This set of FUs covers a representative set of modern FPGA devices. An FU $r \in R_{\text{FU}}$ is defined by its type $\text{type}(r) = \{\text{Adder}_{\text{LUT}}, \text{Multiplier}_{\text{LUT}}, \text{Multiplier}_{\text{EMB}}\}$ and by its *size*, that depends on the input wordlengths. An operation is compatible with an FU if they have compatible types and if the size of the operation is smaller than or equal to the size of the FU [4, 6].

Scheduling is expressed by means of function $\varphi : O \to Z^+$, which assigns a start time to each operation. Resource binding is divided into *FU binding* and *register binding*. *FU binding* makes use of the compatibility graph $G_{\text{COMP}}(O \cup R, C)$ [2], which indicates the compatible resources for each $o \in O$ by means of the set of edges $C \subset O \times R$. The binding between operations and resources is expressed by means of function $\beta : O \to R \times Z^+$, where $\beta(o) = \{r, i\}$ indicates that operation o is bound to the ith instance of resource r. The compatibility rules impose that $(o, r) \subset C$. In a similar fashion, register binding links variables $d \in D$ to registers $r \in R_{\text{REG}}$ by means of function $\gamma : D \to R_{\text{REG}} \times Z^+$. The set of variables D is extracted from O considering that there is a variable assigned to the output of each operation from the subset $O_M \cup O_G \cup O_A$ and to each delay o_D connected to another delay. Registers have an associated size n_r that determines the maximum allowed wordlength of the variables bound to them.

The steering logic consists of multiplexers connected at the inputs of FUs and registers. They are in charge of sending data to and from these two types of resources. R_{MUX} is determined by φ, β, and γ, since φ determines when data are generated, β when data are used by FUs, and γ where data are stored.

2.2. Handling Resource Heterogeneity. The recent appearance of specialized blocks in FPGAs calls for new design methods to efficiently exploit their advantages. In [8], it is proposed to use a normalized resource usage vector. Given an FPGA with M different types of resources R_i ($i = 0 \cdots M-1$), each type with a maximum number of $|R_i|$ resources, the resource requirements of a particular design implementation d can be expressed as the following normalized area vector:

$$\hat{\mathbf{A}} \equiv \left\langle \frac{\#r_0}{|R_0|}, \frac{\#r_1}{|R_1|}, \ldots, \frac{\#r_{M-1}}{|R_{M-1}|} \right\rangle, \tag{1}$$

where $\#r_i$ is the number of resources of type R_i used. Two useful norms are the ∞-norm and the 1-norm:

$$\left\|\hat{\mathbf{A}}\right\|_\infty = \max\left\{ \frac{\#r_0}{|R_0|}, \frac{\#r_1}{|R_1|}, \ldots, \frac{\#r_{M-1}}{|R_{M-1}|} \right\}, \tag{2}$$

$$\left\|\hat{\mathbf{A}}\right\|_1 = \sum_{i=0}^{i=M-1} \frac{\#r_i}{|R_i|}. \tag{3}$$

The inverse of ∞-norm represents the number of times that the same implementation of design d can be replicated within the FPGA device (see [8]), and the 1-norm gives information about the overall resource usage of the implementation. Each norm is interesting on its own, but they have some pitfalls. On the one hand, if two implementations have the same ∞-norm this implies that they can be replicated

the same number of times, but there is no way to know which implementation requires less resources. On the other hand, the 1-norm can tell if a design implementation requires less resources than other, but that does not guarantee that the implementation with less resources can be replicated more times than the other. In the work presented here a combination of ∞-norm and 1-norm , called +-norm (*plus-norm*), is proposed and applied. A metric $\|\cdot\|_+$ that exploits the benefits of both norms but none of the drawbacks should fulfill the following conditions:

$$\forall i,j : \left\|\hat{\mathbf{A}}_i\right\|_+ < \left\|\hat{\mathbf{A}}_j\right\|_+ \Longrightarrow \left(\left\|\hat{\mathbf{A}}_i\right\|_\infty < \left\|\hat{\mathbf{A}}_j\right\|_\infty\right) \vee \left(\left(\left\|\hat{\mathbf{A}}_i\right\|_\infty = \left\|\hat{\mathbf{A}}_j\right\|_\infty\right) \wedge \left(\left\|\hat{\mathbf{A}}_i\right\|_1 < \left\|\hat{\mathbf{A}}_j\right\|_1\right)\right). \tag{4}$$

This can be expressed by means of a combination of the two norms

$$\left\|\hat{\mathbf{A}}\right\|_+ = K \cdot \left\|\hat{\mathbf{A}}\right\|_\infty + \left\|\hat{\mathbf{A}}\right\|_1. \tag{5}$$

A feasible solution for K can be found by trying to comply with (6) for areas $\mathbf{A}_1$ and $\mathbf{A}_2$, such that $\mathbf{A}_1$ requires only one type of resource, $\|\hat{\mathbf{A}}_1\|_\infty > \|\hat{\mathbf{A}}_2\|_\infty$ and $\|\hat{\mathbf{A}}_2\|_1$ has the biggest value that $\|\hat{\mathbf{A}}_1\|_\infty$ allows

$$\left\|\hat{\mathbf{A}}_1\right\|_+ > \left\|\hat{\mathbf{A}}_2\right\|_+. \tag{6}$$

First let us find upper bounds for $\|\hat{\mathbf{A}}_2\|_\infty$ and $\|\hat{\mathbf{A}}_2\|_1$

$$\begin{aligned} \left\|\hat{\mathbf{A}}_2\right\|_\infty &\le \left\|\hat{\mathbf{A}}_1\right\|_\infty - \frac{1}{\max(|R_i|)}, \\ \left\|\hat{\mathbf{A}}_2\right\|_1 &\le M \cdot \left(\left\|\hat{\mathbf{A}}_1\right\|_\infty - \frac{1}{\max(|R_i|)}\right). \end{aligned} \tag{7}$$

Substituting (5) and (7) into (6) allows

$$K \cdot \left\|\hat{\mathbf{A}}_1\right\|_\infty + \left\|\hat{\mathbf{A}}_1\right\|_1 > (K+M) \cdot \left(\left\|\hat{\mathbf{A}}_1\right\|_\infty - \frac{1}{\max(|R_i|)}\right). \tag{8}$$

Since $\|\hat{\mathbf{A}}_1\|_1 = \|\hat{\mathbf{A}}_1\|_\infty$ and $\|\hat{\mathbf{A}}_1\|_\infty \le 1$, a possible range of values of K that complies with (4) is expressed in terms of the number of types of resources (M) and the maximum number resources of any type ($\max(|R_i|)$)

$$K > (M-1) \cdot (\max(|R_i|)) - M. \tag{9}$$

K guarantees that for any two implementations d_i and d_j: (i) if $\|\mathbf{A}_i\|_+ < \|\mathbf{A}_j\|_+$ then d_i can be replicated more times than d_j; (ii) if $\|\mathbf{A}_i\|_+ \le \|\mathbf{A}_j\|_+$ then d_i can be replicated more times than d_j, or the same number of times consuming less resources. Therefore, minimizing +-norm implies that the design can be replicated within the FPGA the maximum possible number of times while using the minimum possible number of resources.

The metric +-norm has a low computational cost and it is suitable for integer linear programming approaches [4, 15] and heuristic approaches [6, 17].

2.3. Resource Modeling. Resources are divided into three types: functional units (R_{FU}), registers (R_{REG}), and steering logic (R_{MUX}). The area and latency of FUs and registers (i.e., $\mathbf{A}(r)$ and $l(r)$) are expressed as functions of the input and output wordlength information (p and n). They are obtained by applying curve fitting to hundreds of synthesis results. The use of accurate delay cost functions proved to provide significant performance improvements compared to some other existent naive approaches (from 12% to 63%, see [6]). Registers are assumed to have a zero latency in terms of clock period, which is true as long as the clock frequency enables to comply with setup and hold times.

Note that $\mathbf{A}$ is a vector with as many components as types of FPGA resources. Thus, it is possible to apply the +-norm to $\mathbf{A}$ in order to optimize the total datapath area. Multiplexers and wiring latencies are neglected, which could be easily overcome by means of multiplying the clock period by an empirical factor [18].

2.3.1. MWL Multiplexers. The area of multiplexers in UWL systems is only affected by the data wordlength, which sets the multiplexer size, and by the number of different data sources (e.g., registers or FUs), which determines the multiplexer width. An estimation of the area of an N-input multiplexer of wordlength M for Virtex-II devices is given by

$$A_{\mathrm{MUX}} = M \cdot \frac{N}{4} \quad \text{slices.} \tag{10}$$

This estimation is specific for Virtex-II, Spartan-3, and Virtex-4, since the implementation of multiplexers relies on the combination of 4-input LUTs and dedicated multiplexers. Another FPGA architectures (i.e., Altera's Stratix-II) that make use only of 4-input LUTs would require a different estimation.

In MWL systems, data must be aligned before being processed by FUs or stored by registers. In [19] the problem of data alignment and multiplexing is tackled by means of alignment blocks introduced before multiplexers. In this work, multiplexers are used for both data multiplexing and data alignment, since the combination of these two tasks leads to a reduction in the number of control signals, and therefore, control logic. In addition, the chances for logic optimization are greater than if two separate blocks (an alignment block and a multiplexer) are used.

Alignment is required at the inputs of adders and at the outputs of both adders and multipliers. On one hand, adders require the alignment of their inputs in order to obtain a meaningful result. If an adder is shared to compute several additions (i.e., a_1 and a_2), an alignment block is required to arrange the MSB of the inputs in the right position for each operation (different alignments will be necessary for a_1 and a_2). On the other hand, the output of the different arithmetic operations—both additions and multiplications—in an algorithm can have the MSBs in different positions. Again, if the FUs are shared the output's MSB changes its position depending on the operation executed, therefore, it is necessary to dynamically align the FU's output using in order to store the data in a register.

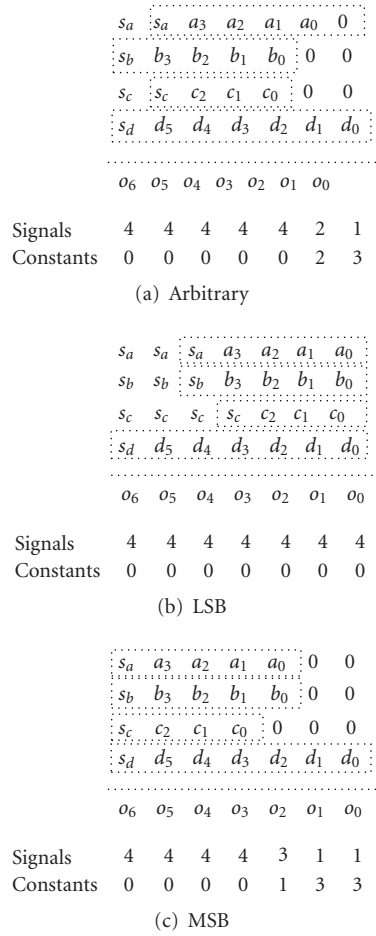


Figure 2: Signal alignment.

Figure 2 presents three different types of alignments for a 4-input multiplexer with inputs signals a, b, c, and d and output o: arbitrary alignment (see Figure 2(a)), least significant bit (LSB) alignment (see Figure 2(b)), and most significant bit (MSB) alignment (see Figure 2(c)). Note that sign extension (see Figures 2(a) and 2(b)) does not offer any opportunity for logic optimization, while zero padding (see Figures 2(a) and 2(c)) does offer it, due to the reduction in the number of signals and the introduction of constant bits (zeros) that can be hard-wired into the multiplexer logic. In fact, it is MSB alignment (see Figure 2(c)) the option which allows a greater logic reduction. Therefore, it is recommended to apply this alignment whenever possible.

A lower bound on the multiplexers' area if the MSB alignment is adopted can be computed as

$$\underline{\mathbf{A}_{\mathbf{MUX}}} = \frac{1}{4}\sum_{i=0}^{N-1}(n_i + 1) \quad \text{slices}, \tag{11}$$

where M is the maximum wordlength present and n_i is the wordlength of signal i.

2.4. Optimization Procedure. In this subsection we extend on the work presented in [6, 17], where the optimization was steered by the ∞-norm and registers and multiplexers were not considered. The optimization procedure is based on Simulated Annealing (SA) [20] and it is shown in Algorithm 1. The inputs are the sequential graph $G_{\text{SEQ}}(O, S)$ and the total latency constraint λ. The optimization procedure determines the set of resources of the datapath $R = R_{\text{FU}} \cup R_{\text{REG}} \cup R_{\text{MUX}}$, the scheduling φ, the FU binding β, and the register binding γ, which define the datapath, the steering logic, and the timing of the control signals.

Table 1: Simulated annealing parameters and conditions.

λ	Latency constraint (in clock cycles)
α_T	Annealing factor (0.95)
$\lvert O\rvert$	Number of operations in algorithm
Equilibrium state	Accepted $> \lvert O\rvert \times \lvert O\rvert$
Frozen state	Iterations $> k \cdot \lvert O\rvert \times \lvert O\rvert$, $k > 1.0$
	3 consecutive times
Restart condition	$\mathbf{A}_{\text{last}} > 1.5 \cdot \mathbf{A}_{\text{min}}$

2.4.1. Simulated Annealing. First, the set of functional units R_{FU}, the set of registers R_{REG}, and the compatibility graph $G_{\text{COMP}}(O, R_{\text{FU}})$ are extracted (line 1). An initial resource mapping m_0 is selected by mapping each operation to the fastest LUT-based resource among the available compatible resources for that operation (line 2), and the area $\mathbf{A}_0$ occupied by the resulting datapath is used as the initial area (line 3). From this point (lines 5–30), the optimization proceeds following the typical SA behavior: the algorithm iterates while producing changes (line 8)—also referred to as *movements*—that modify the value of the cost function (i.e., area) until a certain exit condition is reached. If these changes lead to a cost reduction, they are accepted (line 11), if not, they are accepted with a certain probability which depends on the current temperature T (line 15). The temperature starts at a high value and decreases with time. Most movements are accepted at the beginning of the process, thus enabling a wide design space exploration. As temperature decreases, only those movements which produce small cost deviations are accepted. The temperature is decreased when the *equilibrium state* is reached (line 19). Sporadic *restarting* [21] is also allowed (line 27), which repositions the optimization variables at the last minimum state found.

A summary of simulated annealing parameters and conditions is in Table 1. The annealing factor of 0.95 was chosen empirically aiming at balancing the tradeoff between optimality and solving time.

The variation in cost ΔA is normalized with respect to the initial area $\mathbf{A}_0$ (line 10). This is a simple way to control that the behavior of SA is not affected by the complexity of the algorithm [22], which it is approximated by $\|\widehat{\mathbf{A}}_0\|_n$. The value of n must be set to 1 (or ∞) for homogeneous-architecture FPGAs, and to "+" for heterogeneous-architecture FPGAs.

Input: $G_{\text{SEQ}}(O, S)$, λ
Output: $R = R_{\text{FU}} \cup R_{\text{REG}} \cup R_{\text{MUX}}$, φ, β, γ
(1) Extract $G_{\text{COMP}}(O, R_{\text{FU}})$, R_{FU}, R_{REG}
(2) Find initial mapping m_0
(3) Compute initial area $\mathbf{A}_0$ from m_0
(4) $\mathbf{A}_{\text{min}} = \mathbf{A}_{\text{last}} = \mathbf{A}_0$
$m_{\text{min}} = m_{\text{last}} = m_0$
$T = T_0$
iteration = accepted = exit = 0
(5) **while** ¬ *exit condition* **do**
(6) $m = m_{\text{last}}$
(7) iteration = iteration + 1
(8) Perform change to current m
(9) Compute area $\mathbf{A}$ from m (Algorithm 2)
(10) $\Delta\text{A} = (\|\hat{\mathbf{A}}_{\text{last}}\|_n - \|\hat{\mathbf{A}}\|_n)/\|\hat{\mathbf{A}}_0\|_n$
(11) **if** $\Delta A < 0$ **then**
(12) $\hat{\mathbf{A}}_{\text{min}} = \hat{\mathbf{A}}_{\text{last}} = \mathbf{A}$
$m_{\text{min}} = m_{\text{last}} = m$
accepted = accepted + 1
(13) **else**
(14) $p = e^{\Delta A/T}$, $r = \text{rand}[0 \cdots 1)$
(15) **if** $r \leq p$ **then**
(16) $\mathbf{A}_{\text{last}} = \mathbf{A}$, $m_{\text{last}} = m$
accepted = accepted + 1
(17) **end if**
(18) **end if**
(19) **if** *equilibrium state* **then**
(20) $T = \alpha_T \cdot T$
(21) iterations = accepted = exit = 0
(22) **else if** *frozen state* **then**
(23) $T = \alpha_T \cdot T$
(24) iterations = accepted = 0
(25) exit = exit + 1
(26) **end if**
(27) **if** *restart condition* **then**
(28) $\mathbf{A}_{\text{last}} = \mathbf{A}_{\text{min}}$
$m_{\text{last}} = m_{\text{min}}$
(29) **end if**
(30) **end while**

Algorithm 1: Optimization procedure.

The changes on the cost function (line 8) are performed by applying with equal probabilities the following movements to the resource mapping function m:

(i) M_A: map an operation $o \in O$ to a non mapped resource,

(ii) M_B: map an operation o to another already mapped resource,

(iii) M_C: swap the mapping of two compatible operations mapped to different resources.

2.4.2. Area Computation. The computation of the area cost is shown in Algorithm 2. First, it is checked whether the current resource mapping m complies with latency λ (lines 1–4). If it does not, the actual latency λ' is computed. Later on (line 26), any deviation from the design constraints is penalized by means of increasing the area cost of the solution. Thus, solutions that do not meet the latency constraint are included within the design space exploration [23]. Even though these solutions are never accepted as valid, their inclusion allows a wider architecture exploration than rejecting solutions that do not comply with λ.

Then, the resource allocation and resource binding that minimizes FU area is sought by means of a loop where several list-based scheduling operations are performed (lines 5–18). The purpose of the loop is to check different combinations on the number of instances of the resources. Both lower [24] and upper bounds on the number of instances for each resource are computed (line 6). All combinations of possible instances are computed and stored in the set of vectors I. The list-based scheduling performs an ASAP scheduling to the operations sorted by mobility in ascendant order, providing a fast way to find a valid solution. Note that the size of I is being pruned while the loop iterates; all combinations of FU instances that require areas greater than the minimum found so far are removed (line 15). Thus, resource allocation is sped up.

Once the minimum FU area scheduling is found, the datapath is defined. The tasks of register binding and multiplexer allocation are not commonly included within the optimization loop, in spite of their impact in the final architecture. In this work, these two tasks are part of the optimization procedure.

Register binding is performed by applying a *left-edge* algorithm [13]. Inputs signals are supposed to be available for all λ cycles and do not require storing. Each variable assigned to a delay is initially assigned a register, and after that, the left-edge algorithm is applied as usual.

From sets R_{FU} and R_{REG} and functions φ, β, and γ it is possible to extract the steering logic resources R_{MUX}. Registers have a single multiplexer (see Figure 1), while FUs have two. A goal of multiplexer definition is to maximize the use of the MSB alignment. This aligment can be applied directly to registers and multipliers. However, adders require that the inputs must be aligned to each other. Thus, if an MSB alignment if applied to the mux connected to one of the inputs, it is not possible to do so for the remaining mux, and vice versa. Finally, the control signals can be easily extracted from the scheduling contained in φ.

The area vector is computed by adding the area of each resource multiplied by the number of instances required (line 25). If $\lambda' > \lambda$ the area is penalized by means of factor α_λ. If the implementation does not comply with the latency constraint and if the resulting penalized area is smaller than A_{min}, then the area is forced to be bigger than A_{min} (see line 28).

Summarizing, the optimization procedure is actually controlled by changing iteratively the mapping between operations and FUs. These changes impact on the structure of the datapath and, therefore, on its area cost, which is the function to be minimized. This method provides a robust way to simultaneously perform the tasks of scheduling, resource allocation, and resource binding for multiple wordlength systems. This procedure was satisfactorily applied in [25].

2.5. Wordlength Optimization: A Case Study. Let us introduce this section through a simple LTI case study (Algorithm 3).

Input: $G_{\text{SEQ}}(O, E)$, λ
Output: $R = R_{\text{FU}} \cup R_{\text{REG}} \cup R_{\text{MUX}}$, φ, β, γ, $\mathbf{A}$
(1) Compute minimum latency λ' for mapping m
(2) **if** $\lambda' < \lambda$ **then**
(3) $\lambda' = \lambda$
(4) **end if**
(5) Find set of functional units $R'_{\text{FU}} = \{r'_o, \ldots, r'_{N-1}\}$ with mapped operations
(6) for all $r' \in R'_{\text{FU}}$: Compute instances lower bound $\underline{\text{inst}}(r')$ [24] and upper bound $\overline{\text{inst}}(r')$
(7) $I = \{$all vectors ranging from $\langle \underline{\text{inst}}(r_0), \ldots, \underline{\text{inst}}(r_{N-1})\rangle$ to $\langle \overline{\text{inst}}(r_0), \ldots, \overline{\text{inst}}(r_{N-1})\rangle\}$
(8) $\mathbf{A}_{\text{FUmin}} = \infty$
(9) **for** $\mathbf{i} \in I$ **do**
(10) for all $r'_j \in R'_{\text{FU}}, \text{inst}(r') = \mathbf{i}[j]$
(11) $\mathbf{A}' = \sum_{r' \in R'_{\text{FU}}} \mathbf{A}(r') \cdot \text{inst}(r')$
(12) **if** $\|\hat{\mathbf{A}}'\|_n < \| \hat{\mathbf{A}}_{\text{FUmin}} \|_n$ **then**
(13) **if** list_scheduling(λ', m) **then**
(14) $\mathbf{A}_{\text{FUmin}} = \mathbf{A}$
$\hat{i}_{\text{min}} = \mathbf{i}$
$\varphi_{\text{min}} = \varphi$
$\beta_{\text{min}} = \beta$
(15) $I = \{\mathbf{i} \in I : \| \sum_{r' R'_{\text{FU}}} \hat{\mathbf{A}}(r') \cdot \text{inst}(r')\|_n < \| \hat{\mathbf{A}}_{\text{FUmin}} \|_n\}$
(16) **end if**
(17) **end if**
(18) **end for**
(19) for all $r'_j \in R'_{\text{FU}}, \text{inst}(r') = \mathbf{i}_{\text{min}}[j]$
(20) $\varphi_{\text{min}} = \varphi$
$\beta_{\text{min}} = \beta$
(21) Extract D, R_{REG}
(22) Compute register binding
(23) Extract R_{MUX}
(24) $R = R_{\text{FU}} \cup R_{\text{REG}} \cup R_{\text{MUX}}$
(25) $\mathbf{A}' = \sum_{r \in R} \mathbf{A}(r) \cdot \text{inst}(r)$
(26) $\alpha_\lambda = (\lambda' - \lambda)/\lambda$
(27) $\mathbf{A} = \mathbf{A}' \cdot (1 + \alpha_\lambda)$
(28) **if** $\|\hat{\mathbf{A}}\|_n < \|\hat{\mathbf{A}}_{\text{min}}\|_n$ **then**
(29) $\mathbf{A} = \mathbf{A}_{\text{min}} \cdot (1 + \alpha_\lambda)$
(30) **end if**

Algorithm 2: Computation of area cost.

Input: $a, b, c \in (-1/2, 1/2)$, uniformly distributed
Output: d
(1) **while** true **do**
(2) Get new value of a, b, and c
(3) $m1 = a * 2.384$
(4) $m2 = b * 0.0036$
(5) $s1 = m1 + m2$
(6) $s2 = s1 + c$
(7) New value of output: $d = s2$
(8) **end while**

Algorithm 3: Case study.

Algorithm 3 performs the weighted summation of three signals. The operations involved are two constant multiplications (i.e., gains) and two additions. There are a total of 8 signals.

The goal of WLO is to define the fixed-point format for each signal that enables to produce a hardware implementation of the algorithm. The fixed-point format, as mentioned in Section 2.1, is composed of the pair (p, n). Thus, the ultimate goal of WLO is to find the proper set of (p, n) pairs to optimize the hardware realization of an algorithm. Figure 3 depicts the meaning of this parameters: p is the distance in bits from the fractionary point to the MSB (a zero distance implies that there is no integer part in the number); n is the number of bits used to represent the number without considering the sign bit. A common way to address WLO is to split it into two sequential subtasks: *scaling*, where the values of p are selected, and *wordlength selection*, where the values of n are chosen.

Scaling is performed by means of performing a floating-point simulation and gathering the maximum absolute value of each signal s and computing:

$$p_s = \lfloor \log(\max|s|) \rfloor + 1. \tag{12}$$

Once scaling is accomplished, the values of p are fixed and the values of n are obtained through an optimization process (wordlength selection). The number of bits assigned

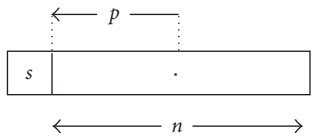


Figure 3: Fixed-point format.

Table 2: UWL versus MWL wordlength optimization.

Signal	$\sigma^2 = 10^{-5}$		$\sigma^2 = 10^{-6}$	
	UWL (p,n)	MWL (p,n)	UWL (p,n)	MWL (p,n)
a	(1,9)	(−1,8)	(1,11)	(−1,10)
b	(1,9)	(−1,2)	(1,11)	(−1,5)
c	(1,9)	(−1,7)	(1,11)	(−1,8)
$m1$	(1,9)	(1,9)	(1,11)	(1,10)
$m2$	(1,9)	(−5,3)	(1,11)	(−5,4)
$s1$	(1,9)	(1,9)	(1,11)	(1,10)
$s2$	(1,9)	(1,8)	(1,11)	(1,10)
d	(1,9)	(1,8)	(1,11)	(1,10)

to a signal (i.e., n) determines the quantization noise that the signal introduces, and, therefore, it has a high impact in the final precision of the system, producing an error in the output signal. During the optimization process different combinations of n are tried in order to look for a particular set that minimizes cost (i.e., area, speed, power) while complying with the output error constraint. The error of the system is typically measured in terms of the peak error value [5, 26], the signal to quantization noise ratio (SQNR) [11, 27], and the variance of the output error [15]. In this work, we adopt the variance of the output error (σ^2).

Table 2 contains the fixed-point formats (i.e., (p,n)) of the signals of Algorithm 3 for both UWL and MWL WLO approaches for different error constraints ($\sigma^2 = 10^{-5}$ and $\sigma^2 = 10^{-6}$). The UWL synthesis is achieved by computing the minimum values of p and n that if applied to all signals the fixed-point realization complies with the noise constraint [15]. The MWL synthesis is achieved by means of an SA-based approach, which minimized the area of a resource-dedicated implementation (with no resource sharing) [28].

Let us focus on the results for $\sigma^2 = 10^{-5}$. The UWL approach clearly requires longer wordlengths than the MWL approach. The main reason for this is that the UWL optimization is far too simple. Also, note that some signals' wordlengths are decreased considerably in the MWL approach (b and $m2$). This is due to the fact that signal b is multiplied by a small constant, so the quantization noise introduced is also small. Similar results are also present for $\sigma^2 = 10^{-6}$. In this case the values of n are bigger since the error constraint is more restrictive.

Summarizing, the MWL approach enables the generation of fixed-point realizations that require a small number of bits. The only drawback is that the complexity of the design process is increased and techniques, such as the proposal in this section, are required.

3. Results

Here, the implementation results are presented. The following benchmarks are used:

(i) ITU RGB to YCrCb converter (ITU) [15],

(ii) 3rd-order lattice filter (LAT_3) [29],

(iii) 4th-order IIR filter (IIR_4) [30],

(iv) 8th-order linear-phase FIR filter (FIR_8).

All algorithms are assigned 8-bit inputs and 12-bit constant coefficients. The algorithm implementations have been tested under different latency and output noise constraint scenarios assuming a system clock of 125 MHz. In particular, the noise constraints were $\sigma^2 = \{10^{-k}, 10^{-(k+1)}, 10^{-(k+2)}\}$, where k is the minimum number that makes 10^{-k} as close as possible to the variance of the quantization noise that would present the output of the benchmark if quantized to 8 bits ($\sigma^2 = (2^{-2n+2p}/12)|_{n=7}$).

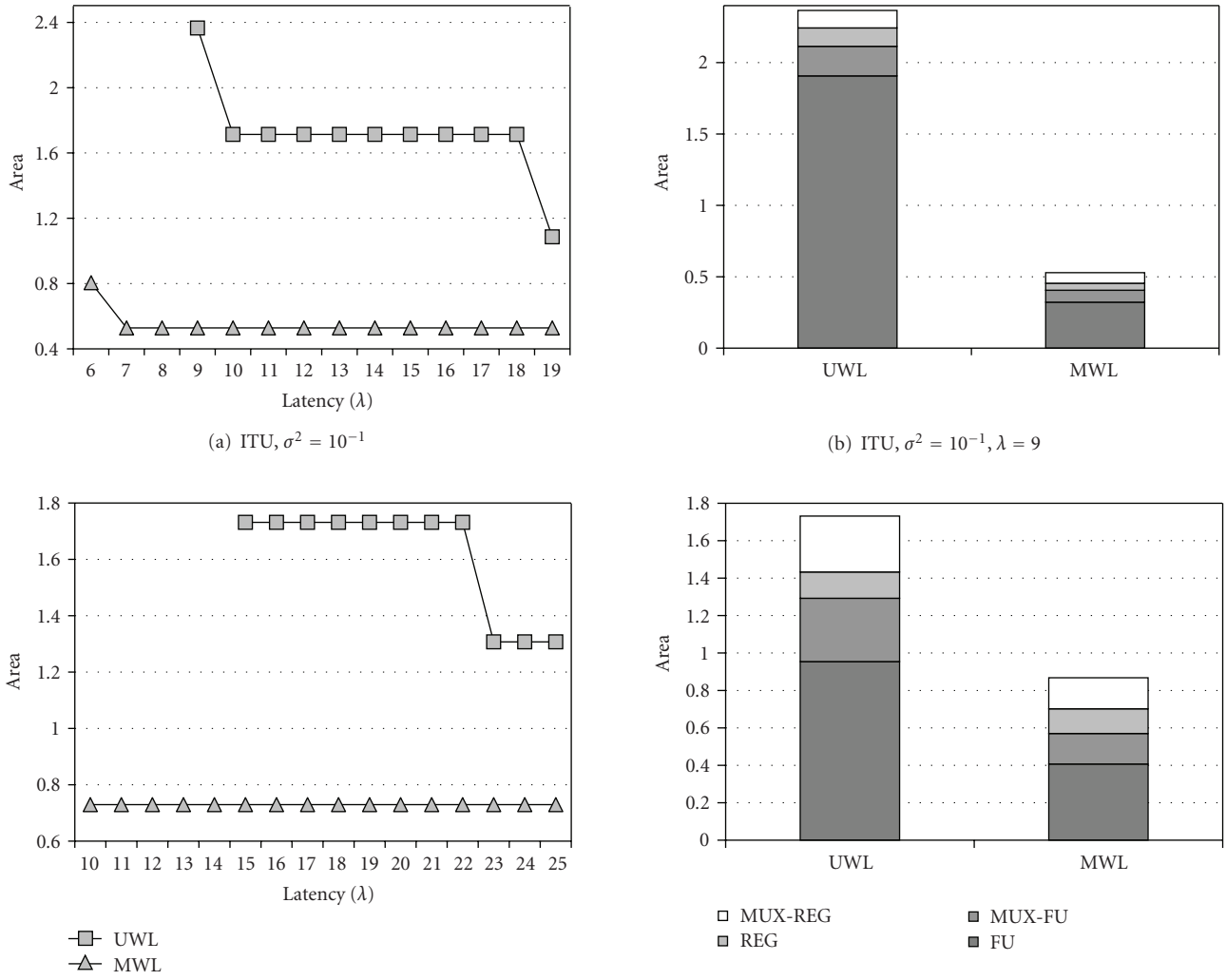
The target devices belong to the Xilinx Virtex-II family. The area results are normalized with respect to the XC2V40 device (256 slices, 4 embedded 18 × 18 multipliers) and expressed according to (2). For instance, an area vector with ∞-norm equal to or smaller than 1.0 implies that the device XC2V40 is the smallest-cost device able to hold the design; whereas a ∞-norm greater than 1.0 and equal to or smaller than 2.0 implies that the smallest-cost device able to hold the design is the XC2V80, and so on.

Before AS, each algorithm is translated to a fixed-point specification by means of two wordlength optimization procedures, that follow a UWL approach and an MWL approach, respectively.

The area results in this section are computed using the resource model explained in Section 2.3, which provides a good estimation of actual synthesis results.

3.1. Uniform Wordlength versus Multiple Wordlength Synthesis: Homogeneous Architectures. Figures 4 and 5 display results on the comparison of UWL versus MWL synthesis using a homogeneous-resource architecture (i.e., only LUT-based resources). Note that the subfigures are arranged in couples, which are related to the same benchmark. The left subfigures depict the area versus latency curves for a particular output noise constraint (see Figures 4(a), 4(c), 5(a), and 5(c)), while the right subfigures contain the detailed resource distribution graph of a particular point of its counterpart (see Figures 4(b), 4(d), 5(b), and 5(d)). Let us define $\lambda_{\min}^{\text{UWL-HOM}}$ as the minimum latency attainable for a UWL-homogeneous implementation of an algorithm, and $\lambda_{\min}^{\text{MWL-HOM}}$ the equivalent for an MWL-homogeneous implementation. The latency used for the experiments ranges from $\lambda_{\min}^{\text{MWL-HOM}}$ to $\lambda_{\min}^{\text{UWL-HOM}} + 10$.

Figures 4(a) and 4(b) contain the implementation results of the ITU benchmark with an output noise variance of 10^{-1}. Figure 4(a) depicts how both the UWL and MWL areas decrease as long as the latency increases. This is expected since the greater the latency the greater the chance of FU reuse. The comparison of the two implementation curves yields that the improvement obtained by means of



(a) ITU, $\sigma^2 = 10^{-1}$

(b) ITU, $\sigma^2 = 10^{-1}, \lambda = 9$

(c) LAT_3, $\sigma^2 = 10^{-5}$

(d) LAT_3, $\sigma^2 = 10^{-5}, \lambda = 22$

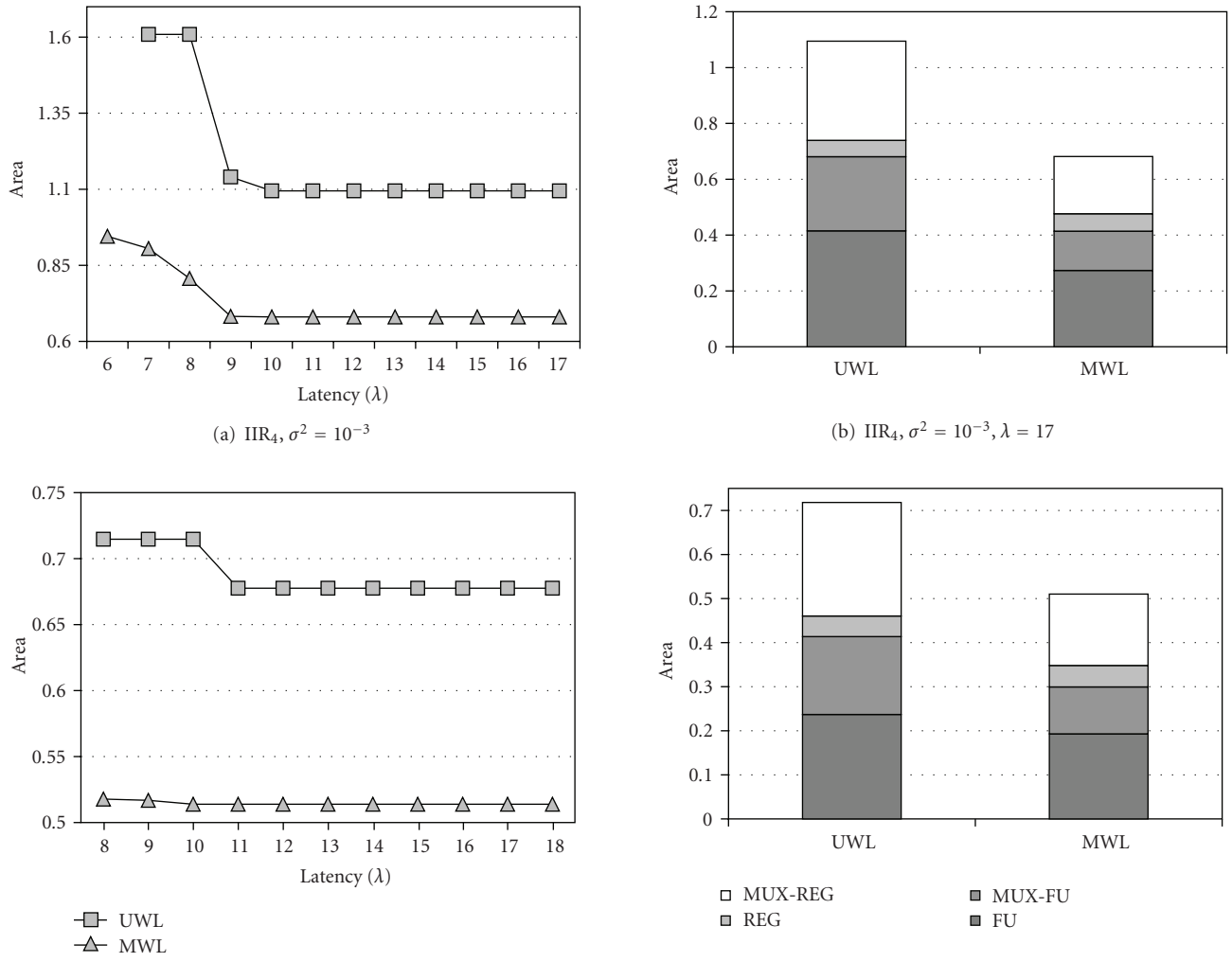
Figure 4: UWL versus MWL: homogeneous implementations (I).

using an MWL approach ranges from 51% to 77%. Also, the minimum latency that each implementation achieves differs considerably. The fine-grain tradeoff between area and quantization noise performed by the MWL approach allows important area reductions when compared to the UWL approach. Figure 4(b) displays the detailed resource distribution for the ITU UWL and MWL implementations correspondant to $\sigma^2 = 10^{-1}$ and $\lambda = 9$. The overall area savings are 77%, and it is due to the fact that the wordlengths of the majority of signals, which impact on FUs, multiplexers and registers, have been highly reduced; FUs' area has been reduced 83%, FU's multiplexers 59%, registers 62%, and registers' multiplexers 39%. It is important to highlight that the area due to multiplexers and registers, although smaller than the FUs' area, makes up a significant part of the total area (20% for UWL and 39% for MWL). Hence the importance of including its cost within the optimization loop is analyzed in Section 3.4.

The other benchmarks also show large area improvements: LAT_3 up to 49% (see Figure 4(c)), IIR_4 up to 49% (see Figure 5(a)), and FIR_8 up to 28% (see Figure 5(c)). As observed in the detailed resource distribution subfigures (see Figures 4(d), 5(b), and 5(d)), the area of the majority of the resources has been highly decreased. Also, it is noted that the percentage of area devoted to multiplexation and data storing is high in proportion to the overall implementation area. The minimum latency is also improved (see Figures 4(a), 4(b), and 5(a)).

In Figure 4(c) the MWL area does not decrease as long as the latency increases. This is due to the fact that the wordlengths are small enough as to allow maximum resource sharing for all latencies, thus the coincidence in the area results for the MWL implementations. This situation might change if a different error constraint (σ^2) is applied during WLO.

Table 3 contains the implementation results for all the benchmarks corresponding to three different quantization noise scenarios. For each quantization scenario the latency ranges from $\lambda_{\min}^{\text{UWL-HOM}}$ to $\lambda_{\min}^{\text{UWL-HOM}} + 10$, and the minimum, maximum, and mean values of the area improvements



(a) IIR_4, $\sigma^2 = 10^{-3}$

(b) IIR_4, $\sigma^2 = 10^{-3}$, $\lambda = 17$

(c) FIR_8, $\sigma^2 = 10^{-4}$

(d) FIR_8, $\sigma^2 = 10^{-4}$, $\lambda = 8$

Figure 5: UWL versus MWL: homogeneous implementations (II).

Table 3: UWL versus MWL for homogeneous architectures.

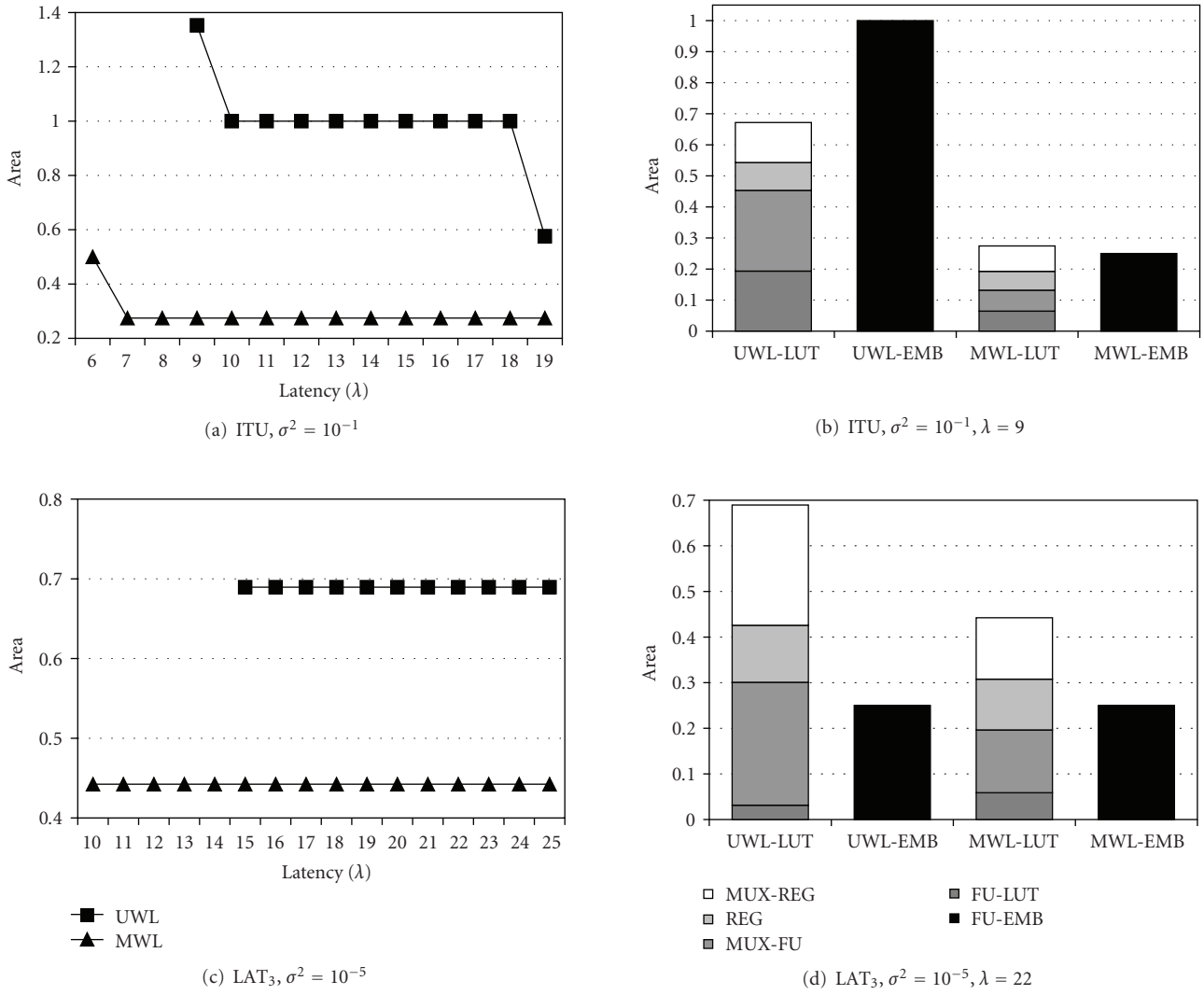
Bench.	σ^2	Area improvement (%)		
		Min	Max	Mean
ITU	10^{-1}	51.36	77.66	68.32
	10^{-2}	48.44	76.31	66.41
	10^{-3}	46.51	75.40	65.13
LAT_3	10^{-3}	44.07	44.07	44.07
	10^{-4}	33.66	33.66	33.66
	10^{-5}	33.62	49.89	45.42
IIR_4	10^{-3}	37.85	49.86	39.69
	10^{-4}	34.30	63.55	50.65
	10^{-5}	37.08	64.99	52.72
FIR_8	10^{-3}	40.28	47.68	43.54
	10^{-4}	24.16	28.10	25.14
	10^{-5}	22.04	25.73	22.82
All		22.04	77.66	46.46

obtained by the MWL implementations in comparison to the UWL implementations are computed. The first column in the table contains the name of the benchmark. The second, the output noise variance. And the third column contains area improvement values. The last row holds the minimum, maximum, and average improvements considering all results simultaneously.

The area improvements obtained are remarkable: mean improvements range from 47% to 77%. Note that the minimum improvements obtained for all benchmarks are quite close to both the maximum and the mean. The results clearly show that an MWL-based AS approach achieves significant area reductions.

Regarding latency, the minimum latency achievable by UWL implementations is reduced in average a 22% by means of MWL AS.

3.2. Uniform Wordlength versus Multiple Wordlength Synthesis: Heterogeneous Architectures. Figures 6 and 7 contain results on the comparison of UWL versus MWL synthesis



(a) ITU, $\sigma^2 = 10^{-1}$

(b) ITU, $\sigma^2 = 10^{-1}, \lambda = 9$

(c) LAT_3, $\sigma^2 = 10^{-5}$

(d) LAT_3, $\sigma^2 = 10^{-5}, \lambda = 22$

Figure 6: UWL versus MWL: homogeneous implementations (I).

using a heterogeneous-resource architecture (i.e., both LUT-based and embedded resources present). The arrangement of figures is similar to that of the previous subsection. Now, the latency ranges from $\lambda_{\min}^{\text{MWL-HET}}$ to $\lambda_{\min}^{\text{UWL-HET}} + 10$ (HET indicates heterogeneous implementations).

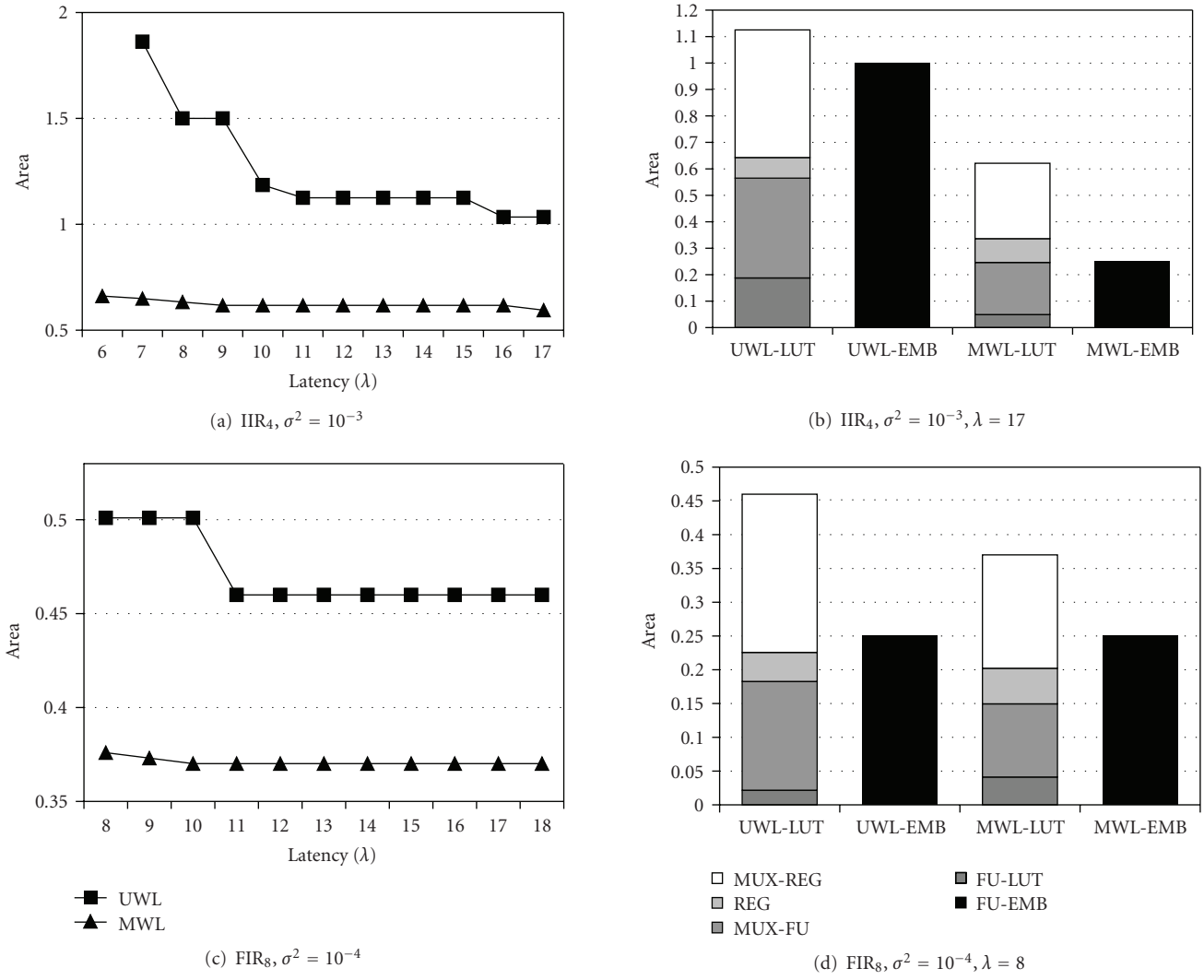
Figures 6(a), 6(c), 7(a), and 7(c) contain the implementation area versus latency curves. The graphs clearly show how the area is reduced by means of an MWL synthesis: ITU up to 79% (see Figure 6(a)), LAT_3 up to 35% (see Figure 6(c)), IIR_4 up to 40% (see Figure 7(a)), and FIR_8 up to 26% (see Figure 7(b)). The detailed resource distribution in Figures 6(b), 6(d), 7(b), and 7(d) shows how the majority of resources are decreased, and in particular the embedded multipliers and the FUs' multiplexers are clearly optimized. For instance, the ITU resource distribution for $\sigma^2 = 10^{-2}$ and $\lambda = 10$ (see Figure 6(b)) shows an overall area reduction of 72%. The LUT-based resources are reduced 59% (LUT-based FUs' area has been reduced 32%, FU's multiplexers 74%, registers 32%, and registers' multiplexers 36%); while the embedded FUs are reduced 75%.

Note that the area of embedded resources for Figures 6(d) and 7(d) is the same for both the UWL and MWL approaches, in fact a single multiplier is being used (1 out of 4). This happens because the wordlengths involved in multiplications, though not the same, are small enough for both UWL and MWL approaches as to enable the use of a single embedded multiplier. However, the LUT-based areas are quite different, and, as a result, the overall resource usage is much smaller for the MWL implementation.

In Figure 6(c) the UWL and MWL areas do not decrease as long as the latency increases. Again, this is due to the fact that the particular wordlengths obtained allow maximum resource sharing for all latencies. Different error constraints (σ^2) might change this situation.

Again, the figures show how the minimum latency can be highly improved by means of an MWL approach. Also, it can be seen that the LUT-based resources are devoted almost entirely to data multiplexing and storing.

Table 4 contains the implementation results of all the benchmarks corresponding to three different quantization



(a) IIR_4, $\sigma^2 = 10^{-3}$

(b) IIR_4, $\sigma^2 = 10^{-3}$, $\lambda = 17$

(c) FIR_8, $\sigma^2 = 10^{-4}$

(d) FIR_8, $\sigma^2 = 10^{-4}$, $\lambda = 8$

Figure 7: UWL versus MWL: homogeneous implementations (II).

noise scenarios. For each quantization scenario the latency ranges from $\lambda_{\text{min}}^{\text{UWL-HET}}$ to $\lambda_{\text{min}}^{\text{UWL-HET}} + 10$, and the minimum, maximum, and mean area improvements obtained by the MWL implementations in comparison to the UWL implementations are computed considering ∞-norm area, the LUT-based area, and the embedded FUs area. The first column in the table contains the name of the benchmark. The second, the output noise variance applied. The third column contains the minimum, maximum, and mean ∞-norm area improvement values. The fourth column contains the minimum, maximum, and mean values of the LUT-based resource area. And the last column contains the minimum, maximum, and mean values of the embedded FUs area.

The area improvements obtained are considerable; ITU obtains up to 80.77%, LAT_3 up to 48.87%, IIR_4 up to 65.13%, FIR_8 up to 38.01%. Note that the minimum improvements obtained for most of the benchmarks are again quite close to both the maximum and the mean.

The LUT-based area reductions are up to 81.07% for ITU, up to 48.87% for LAT_3, up to 65.13% for IIR_4, and up to 43.83% for FIR_8. The embedded resources are only reduced for benchmarks ITU (up to a 75.00%) and IIR_4 (up to 83.33%). Benchmarks LAT_3 and FIR_8 use the minimum possible number of embedded resources (1 embedded multiplier), hence the 0% improvement.

Area improvements up to 80.77% are achieved. The average improvement is 44.88% for the overall area, 42.76% for the LUT-based resources, and 24.03% for the embedded resources. The results clearly show that an MWL-based approach for AS leads to significant area reductions.

As a final note regarding these area results, the authors would like to emphasized that the *plus-norm* has been used during the optimization process, but it is not used to present the results as it cannot be directly related to the percentage of occupation of the FPGA. Thus, the ∞-norm is used instead.

The latency analysis throws that the minimum UWL latency is reduced an average 19% by means of MWL AS.

3.3. MWL Synthesis: Heterogeneous versus Homogeneous. Table 5 contains the implementation results of all the

Table 4: UWL versus MWL for heterogeneous architectures.

Bench.	σ^2	$\|\hat{\mathbf{A}}\|_\infty$ %			A_{LUT} %			A_{EMB} %		
		Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
ITU	10^{-1}	54.85	80.77	73.69	55.56	81.07	63.19	50.00	75.00	72.73
	10^{-2}	52.37	79.71	71.37	52.37	79.71	60.41	50.00	75.00	72.73
	10^{-3}	47.80	77.76	66.71	47.80	77.76	56.33	50.00	75.00	72.73
LAT_3	10^{-3}	48.87	48.87	48.87	48.87	48.87	48.87	0.00	0.00	0.00
	10^{-4}	35.84	35.84	35.84	35.84	35.84	35.84	0.00	0.00	0.00
	10^{-5}	31.70	31.70	31.70	31.70	31.70	31.70	0.00	0.00	0.00
IIR_4	10^{-3}	37.47	41.27	38.33	37.47	45.10	39.05	0.00	0.00	0.00
	10^{-4}	32.97	40.70	34.74	32.97	40.70	34.74	0.00	0.00	0.00
	10^{-5}	40.32	65.13	49.57	40.32	65.13	48.55	50.00	83.33	71.67
FIR_8	10^{-3}	32.45	38.01	33.97	38.79	43.83	39.68	0.00	0.00	0.00
	10^{-4}	27.53	32.83	28.62	27.53	32.83	28.62	0.00	0.00	0.00
	10^{-5}	19.53	26.12	21.17	19.53	26.12	21.17	0.00	0.00	0.00
All		19.53	80.77	44.88	19.53	81.07	42.76	0.00	83.33	24.03

benchmarks corresponding to three different quantization noise scenarios. For each quantization scenario the latency ranges from $\lambda_{\text{min}}^{\text{MWL-HOM}}$ to $\lambda_{\text{min}}^{\text{MWL-HOM}}+10$, and the minimum, maximum, and mean values of the area improvements, in terms of ∞-norm, obtained by the MWL implementations in comparison to the UWL implementations are computed. The first column of the table contains the name of the benchmark. The second, the output noise variance applied. And, the third column contains the minimum, maximum and mean area improvement values.

The area improvements obtained are remarkable; ITU obtains up to 54.76%, LAT_3 up to 43.09%, IIR_4 up to 48.79%, FIR_8 up to 44.68%. Note that, again, the minimum improvements obtained for all benchmarks are quite close to both the maximum and the mean. Area improvements up to 54.76% are achieved, being the average improvement 40.23%. The results clearly show that the inclusion of embedded resources within AS leads to higly optimized DSP implementations.

Regarding latencies, the minimum latency achievable by both homogeneous and heterogeneous implementations is the same for the experiments performed. This is due to the fact that the latency of resources is very similar in the particular conditions used for the tests. The same experiments presented in this section were repeated increasing the constant wordlength to 16 bits, obtaining that heterogeneous implementations reduced 7% the minimum latency in constrast to homogeous implementations.

3.4. Effect of Registers and Multiplexers. In this subsection the effect of including the cost of registers and multiplexers within the optimization loop is investigated. As in the previous experiments, the analysis is performed implementing the benchmarks using different noise and latency constraints. Before AS is applied a gradient-descent quantization [28] is applied according to the given noise constraint. The comparison is done by using Algorithm 1 to perform the AS using two different area cost estimation solutions: (i) Algorithm 2, which is referred to as the *complete* area estimation algorithm, and (ii) a simplified version of Algorithm 2 (*simplified* area estimation algorithm) where the cost of registers and multiplexers is neglected. When the simplified area estimation is used, the cost of registers and multiplexers is included after the optimization loop has finished its execution, using the complete area estimation (Algorithm 2).

Table 5: MWL synthesis: homogeneous versus heterogeneous architectures.

Bench.	σ^2	$\|\hat{\mathbf{A}}\|_\infty$ %		
		Min	Max	Mean
ITU	10^{-1}	40.73	52.69	51.60
	10^{-2}	43.29	53.98	53.01
	10^{-3}	50.45	54.76	54.32
LAT_3	10^{-3}	42.68	43.09	42.72
	10^{-4}	39.36	39.36	39.36
	10^{-5}	38.73	39.74	38.82
IIR_4	10^{-3}	34.83	44.77	36.04
	10^{-4}	32.92	48.23	35.96
	10^{-5}	33.21	48.79	35.79
FIR_8	10^{-3}	21.42	41.46	28.37
	10^{-4}	27.02	44.68	33.24
	10^{-5}	27.46	44.62	33.52
All		21.42	54.76	40.23

Table 6 contains the results of this analysis. The latencies range from $\lambda_{\text{min}}^{\text{ARCH}}$ to $\lambda_{\text{min}}^{\text{ARCH}}$ + 10, where ARCH refers to the type of FPGA architecture used (homogeneous or heterogeneous). The noise constraints are the same used in the previous subsection (three σ^2 for each benchmark), though the results have been combined into a single row. The first column contains the type of FPGA architecture.

Table 6: Complete versus simplified cost estimation: area improvement (%).

Arch.	Bench.	Area improvement		
		Min	Max	Mean
HOM	ITU	0.00	0.95	0.30
	LAT_3	0.71	3.50	1.53
	IIR_4	0.00	5.35	1.26
	FIR_8	0.00	1.77	0.31
HET	ITU	0.00	25.85	1.89
	LAT_3	1.15	5.77	2.52
	IIR_4	0.00	35.57	8.09
	FIR_8	0.00	8.11	0.83
All		0.00	35.57	2.09

The second column indicates the benchmark used. And the fourth column contains the minimum, maximum, and average area improvements obtained by the complete area estimation synthesis in contrast to the simplified area estimation synthesis. The last row includes the minimum, maximum, and mean improvements for all benchmarks.

The average improvements for the different benchmarks range from 0.00% to 8.09%, being the overall average improvement of 2.09%. The maximum improvement found is 35.57%. These results clearly show that failing to include the cost of registers and multiplexer during the optimization procedure can lead to unwanted area penalties.

4. Conclusions

In this paper an architectural synthesis procedure able to produce optimized fixed-point implementations using modern FPGA devices is presented. The key to success is provided by the use of highly accurate models of the datapath resources, a complete datapath resource set that includes multiplexer and registers, a novel method to handle fixed-point data alignment and multiplexing, and also the introduction of a novel resource usage metric that can cope with LUT-based and embedded FGPGA resources.

The AS procedure produces area improvements of up to 80% when compared to uniform-wordlength implementations, and latency improvement of up to 22%. The efficient use of embedded resources achieves area improvements of up to 54% when compared to homogeneous implementations. Also, the inefficiency of current FPGA architectures to implement data steering was exposed.

These results are intented to be further improved by means of tightly combining the fixed-point refinement process within the architectural synthesis [4, 31]. Also, the inclusion of the control logic in the resource model is regarded as a future research line.

Acknowledgment

This work was supported by the Spanish Ministry of Education and Science under Research Project TEC2006-13067-C03-03.

References

[1] K.-I. Kum and W. Sung, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, 2001.

[2] G. Constantinides, P. Cheung, and W. Luk, "Heuristic datapath allocation for multiple wordlength systems," in *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE '01)*, pp. 791–796, Munich, Germany, 2001.

[3] J. Cong, Y. Fan, G. Han, et al., "Bitwidth-aware scheduling and binding in high-level synthesis," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, pp. 856–861, Shanghai, China, 2005.

[4] G. Caffarena, G. A. Constantinides, P. Y. K. Cheung, C. Carreras, and O. Nieto-Taladriz, "Optimal combined word-length allocation and architectural synthesis of digital signal processing circuits," *IEEE Transactions on Circuits and Systems II*, vol. 53, no. 5, pp. 339–343, 2006.

[5] S. A. Wadekar and A. C. Parker, "Accuracy sensitive word-length selection for algorithm optimization," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '98)*, pp. 54–61, San Jose, Calif, USA, 1998.

[6] G. Caffarena, J. A. López, C. Carreras, and O. Nieto-Taladriz, "High-level synthesis of multiple word-length DSP algorithms using heterogeneous-resource FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 675–678, Madrid, Spain, 2006.

[7] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for FPGAs," in *Proceedings of the 39th Design Automation Conference (DAC '02)*, pp. 862–869, New Orleans, La, USA, June 2002.

[8] C.-S. Bouganis, G. A. Constantinides, and P. Y. K. Cheung, "A novel 2D filter design methodology for heterogeneous devices," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 13–22, Napa, Calif, USA, April 2005.

[9] X. Liang, J. S. Vetter, M. C. Smith, and A. S. Bland, "Balancing FPGA resource utilities," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 156–162, Las Vegas, Nev, USA, June 2005.

[10] A. M. Smith, G. A. Constantinides, and P. Y. K. Cheung, "Fused-arithmetic unit generation for reconfigurable devices using common subgraph extraction," in *Proceedings of the International Conference on Field Programmable Technology (FPT '07)*, pp. 105–112, Kitakyushu, Japan, December 2007.

[11] R. Rocher, D. Menard, N. Herve, and O. Sentieys, "Fixed-point configurable hardware components," *EURASIP Journal of Embedded Systems*, vol. 2006, Article ID 23197, 13 pages, 2006.

[12] N. Hervé, D. Ménard, and O. Sentieys, "About the importance of operation grouping procedures for multiple word-length architecture optimizations," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '07)*, pp. 191–200, March 2007.

[13] G. De Michelli, *Synthesis and Optimization of Digital Circuits*, Electrical and Computer Engineering series, McGraw-Hill, New York, NY, USA, 1994.

[14] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie, "An automatic word length determination method," in *Proceedings*

of the IEEE International Symposium on Circuits and Systems (ISCAS '01), vol. 5, pp. 53–56, Sydney, Australia, May 2001.

[15] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432–1442, 2003.

[16] M. Holzer, B. Knerr, P. Belanoviç, and M. Rupp, "Efficient design methods for embedded communication systems," *EURASIP Journal of Embedded Systems*, vol. 2006, Article ID 64913, 2006.

[17] G. Caffarena, J. A. López, C. Carreras, and O. Nieto-Taladriz, "Optimized implementation of DSP cores on FPGAs using logic-based and embedded resources," in *Proceedings of the International Symposium on System-On-Chip (SoC '06)*, pp. 103–106, Tampere, Finland, November 2006.

[18] R. Enzler, T. Jeger, D. Cottet, and G. Tröster, "High-level area and performance estimation of hardware building blocks on FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FBL '00)*, pp. 525–534, Villach, Austria, August 2000.

[19] K. Schoofs, G. Goossens, and H. De Man, "Bit-alignment in hardware allocation for multiplexed DSP architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '93)*, pp. 289–293, October 1993.

[20] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[21] N. Benvenuto, M. Marchesi, and A. Uncini, "Applications of simulated annealing for the design of special digital filters," *IEEE Transactions on Signal Processing*, vol. 40, no. 2, pp. 323–332, 1992.

[22] H. Orsila, T. Kangas, E. Salminen, and T. D. Hämäläinen, "Parameterizing simulated annealing for distributing task graphs on multiprocessor SoCs," in *Proceedings of the International Symposium on System-On-Chip (SoC '06)*, pp. 1–4, Tampere, Finland, November 2006.

[23] M. Lopez-Vallejo, J. Grajal, and J. Lopez, "Constraintdriven system partitioning," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '00)*, pp. 411–416, Paris, France, March 2000.

[24] S. Y. Ohm, F. J. Kurdahi, and N. D. Dutt, "A unified lower bound estimation technique for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 5, pp. 458–472, 1997.

[25] G. Caffarena, J. A. López, G. Leyva, C. Carreras, and O. Nieto-Taladriz, "Optmized architectural synthesis of fixed-point datapaths," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 85–90, Cancun, Mexico, 2008.

[26] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 269–297, 2003.

[27] J. A. López, G. Caffarena, C. Carreras, and O. Nieto-Taladriz, "Fast and accurate computation of the round-off noise of linear time-invariant systems," *IET Circuits, Devices and Systems*, vol. 2, no. 4, pp. 393–408, 2008.

[28] M.-A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 2, pp. 612–615, May 2002.

[29] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, New York, NY, USA, 1999.

[30] K.-I. Kum, J. Kang, and W. Sung, "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Systems II*, vol. 47, no. 9, pp. 840–848, 2000.

[31] G. Caffarena, *Combined word-length allocation and high-level synthesis of digital signal processing circuits*, Ph.D. dissertation, Universidad Politécnica de Madrid, Madrid, Spain, 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 219140, 9 pages
doi:10.1155/2009/219140

Research Article

Pipeline FFT Architectures Optimized for FPGAs

Bin Zhou,[1,2] Yingning Peng,[1] and David Hwang[2]

[1] *Department of Electronic Engineering, Tsinghua University, Beijing 100084, China*
[2] *Department of Electrical and Computer Engineering, George Mason University, 4400 University Drive, Fairfax, VA 22030, USA*

Correspondence should be addressed to David Hwang, dhwang@gmu.edu

Received 28 February 2009; Accepted 23 June 2009

Recommended by Cesar Torres

This paper presents optimized implementations of two different pipeline FFT processors on Xilinx Spartan-3 and Virtex-4 FPGAs. Different optimization techniques and rounding schemes were explored. The implementation results achieved better performance with lower resource usage than prior art. The 16-bit 1024-point FFT with the R2^2SDF architecture had a maximum clock frequency of 95.2 MHz and used 2802 slices on the Spartan-3, a throughput per area ratio of 0.034 Msamples/s/slice. The R4SDC architecture ran at 123.8 MHz and used 4409 slices on the Spartan-3, a throughput per area ratio of 0.028 Msamples/s/slice. On Virtex-4, the 16-bit 1024-point R2^2SDF architecture ran at 235.6 MHz and used 2256 slice, giving a 0.104 Msamples/s/slice ratio; the 16-bit 1024-point R4SDC architecture ran at 219.2 MHz and used 3064 slices, giving a 0.072 Msamples/s/slice ratio. The R2^2SDF was more efficient than the R4SDC in terms of throughput per area due to a simpler controller and an easier balanced rounding scheme. This paper also shows that balanced stage rounding is an appropriate rounding scheme for pipeline FFT processors.

1. Introduction

The Fast Fourier Transform (FFT), as an efficient algorithm to compute the Discrete Fourier Transform (DFT), is one of the most important operations in modern digital signal processing and communication systems. The pipeline FFT is a special class of FFT algorithms which can compute the FFT in a sequential manner; it achieves real-time behavior with nonstop processing when data is continually fed through the processor. Pipeline FFT architectures have been studied since the 1970's when real-time large scale signal processing requirements became prevalent. Several different architectures have been proposed, based on different decomposition methods, such as the Radix-2 Multipath Delay Commutator (R2MDC) [1], Radix-2 Single-Path Delay Feedback (R2SDF) [2], Radix-4 Single-Path Delay Commutator (R4SDC) [3], and Radix-2^2 Single-Path Delay Feedback (R2^2SDF) [4]. More recently, Radix-2^2 to Radix-2^4 SDF FFTs were studied and compared in [5]; in [6] an R2^3SDF was implemented and shown to be area efficient for 2 or 3 multipath channels. Each of these architectures can be classified as multipath or single-path. Multipath approaches can process M data inputs simultaneously, though they have limitations on the number of parallel data-paths, FFT points, and radix. This paper focuses on single-path architectures.

From the hardware perspective, Field Programmable Gate Array (FPGA) devices are increasingly being used for hardware implementations in communications applications. FPGAs at advanced technology nodes can achieve high performance, while having more flexibility, faster design time, and lower cost. As such, FPGAs are becoming more attractive for FFT processing applications and are the target platform of this paper.

The primary goal of this research is to optimize pipeline FFT processors to achieve better performance and lower cost than prior art implementations. In this paper, two comparative implementations (R4SDC and R2^2SDF) of pipeline FFT processors targeted towards Xilinx Spartan-3 and Virtex-4 FPGAs are presented. Different parameters such as throughput, area, and SQNR are compared.

The rest of the paper is organized as follows. Section 2 discusses the methodology used to select the two architectures. Section 3 describes the implementation tools and optimization methods used to improve performance and reduce resource utilization. Section 4 explains the balanced rounding schemes that were implemented and their impact on the signal-to-quantization noise ratio (SQNR). Section 5 presents the results, and Section 6 presents some brief conclusions.

Table 1: Hardware resource requirements comparison of pipeline FFT architectures (based on [4]).

	Complex multipliers	Complex adders	Memory size	Control logic	Comp. Utilization	
					add/sub	Multiplier
R2SDF	$\log_2 N - 2$	$2\log_2 N$	$N - 1$	simple	50%	50%
R4SDF	$\log_4 N - 1$	$8\log_4 N$	$N - 1$	medium	25%	75%
R4SDC	$\log_4 N - 1$	$3\log_4 N$	$2N - 2$	complex	100%	75%
R2^{2}SDF	$\log_4 N - 1$	$4\log_4 N$	$N - 1$	simple	75%	75%
R2MDC	$\log_2 N - 2$	$2\log_2 N$	$3N/2 - 2$	simple	50%	50%
R4MDC	$3(\log_4 N - 1)$	$8\log_4 N$	$5N/2 - 4$	medium	25%	25%

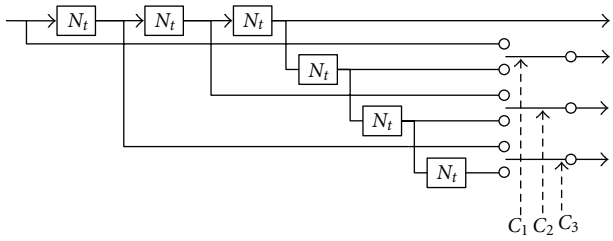


Figure 1: R4SDC commutator of stage t.

Table 2: Implementation tools.

Design step	Tool
VHDL simulation	ModelSim SE 6.2b
FPGA synthesis	Synplicity Synplify Pro; Xilinx XST
FPGA implementation	Xilinx ISE 9.1
Target FPGA	Spartan-3 Family; Virtex-E Family; Virtex-4 Family
Verification	Matlab R2006a

2. Pipeline FFT Architectures

2.1. Architecture Selection. The major characteristics and resource requirements of several pipeline FFT architectures are listed in Table 1. Computational efficiency is measured by resource utilization percentage—how often the resources are in an active state versus an idle state. As shown in the table, the radix-4 Single-Path Delay-Commutator (R4SDC) and radix-2^2 Single Path Delay Feedback (R2^{2}SDF) architectures provide the highest computational efficiency and were selected for implementation. The R4SDC architecture is appealing due to the computational efficiency of its addition; however the controller design is complex. The R2^{2}SDF architecture has a simple controller but a less efficient addition scheme. These designs are both radix-4 and scalable to an arbitrary FFT size N (N is a power of 4).

2.2. R4SDC Architecture.

R4SDC Algorithms. The R4SDC was proposed by Bi and Jones [3] and uses an iterative architecture to calculate the radix-4 FFT. The key to the algorithm is splitting the FFT into different stages by using different radices. In this paper, the radix is always 4.

The derivation starts from the fundamental DFT equation for an N-point FFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N{}^{nk} \quad \left(k = 0, 1 \cdots N-1; W_N = e^{-j(2\pi/N)}\right). \tag{1}$$

N can be represented as composite of v numbers $N = r_1 r_2 \cdots r_v$ and defined as

$$N_t = \frac{N}{r_1 r_2 \cdots r_t}, \quad 1 \le t \le v-1, \tag{2}$$

where t is the stage, and r_t is the stage radix. After putting (2) into (1) and applying the relationship $W_{N_i N_j}^{N_j k} = W_{N_i}{}^{k}$, (1) becomes

$$X(k) = \sum_{q_1=0}^{N_1-1} W_N{}^{q_1 k} \sum_{p=0}^{r_1-1} x(N_1 p + q_1) W_{r_1}{}^{pk}. \tag{3}$$

Indeces k_1 and m_1 can be defined by $k = r_1 k_1 + m_1$, where $0 \le k_1 \le N_1 - 1,\ 0 \le m_1 \le r_1 - 1$. Equation (3) becomes

$$\begin{aligned} X(r_1 k_1 + m_1) &= \sum_{q_1=0}^{N_1-1} x_1(q_1, m_1) W_{N_1}{}^{q_1 k_1}, \\ x_1(q_1, m_1) &= W_N{}^{q_1 m_1} \sum_{p=0}^{r_1-1} x(N_1 p + q_1) W_{r_1}{}^{p m_1}. \end{aligned} \tag{4}$$

Therefore the complete N-point DFT can be written as $v - 1$ different stages with intermediate stages in a recursive equation:

$$x_t(q_t, m_t) = W_{N_{t-1}}^{q_t m_t} \sum_{p=0}^{r_t-1} x_{t-1}(N_t p + q_t, m_{t-1}) W_{r_t}^{p m_t}. \tag{5}$$

$W_{N_{t-1}}^{q_t m_t}$ is the twiddle factor. For radix-4, the equations become

$$X(4k_1 + m_1) = \sum_{q_1=0}^{N/4-1} x_1(q_1, m_1) W_{N/4}{}^{q_1 k_1},$$
$$x_t(q_t, m_t) = W_{N_{t-1}}^{q_t m_t} \sum_{p=0}^{3} x_{t-1}(N_t p + q_t, m_{t-1}) W_4{}^{p m_t} \quad (6)$$

The R4SDC architecture is presented in Figures 1–3. An N-point radix-4 pipeline FFT is decomposed to $\log_4 N$ stages. Each stage consists of a commutator, a butterfly, and a complex multiplier. Figure 1 outlines the commutator for the R4SDC. Its six shift registers provide N_t delays. The control signals are generated by logic functions. The butterfly element, shown in Figure 2, performs the summation, where trivial multiplication is replaced by add/sub and imaginary/real part swapping. Figure 3 shows the overall architecture.

2.3. R2²SDF Architecture. The R2²SDF architecture was proposed by He and Torkelson [4] and also begins from (1). He applies a 3-dimensional index map:

$$n = \left\langle \frac{N}{2} n_1 + \frac{N}{4} n_2 + n_3 \right\rangle_N; \qquad k = \langle k_1 + 2k_2 + 4k_3 \rangle_N. \quad (7)$$

Using the Common Factor Algorithm (CFA) to decompose the twiddle factor, the FFT can be reconstructed as a set of 4 DFTs of length $N/4$:

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{N/4-1} \left[H(k_1, k_2, n_3) W_N^{n_3(k_1+2k_2)} \right] W_{N/4}^{n_3 k_3} \quad (8)$$

$H(k_1, k_2, n_3)$ can be expressed as

$$H(k_1, k_2, n_3) = \overbrace{[x(n_3) + (-1)^{k_1} x(n_3 + (N/2))]}^{BF\,2I} + (-j)^{(k_1+2k_2)} \underbrace{\overbrace{[x(n_3 + (N/4)) + (-1)^{k_1} x(n_3 + (3/4)N)]}^{BF\,2I}}_{BF\,2II}, \quad (9)$$

The R2²SDF algorithm can be mapped to the architecture shown in Figures 4–6. The number of stages is $\log_4 N$. Every stage contains two butterfly elements, each associated by an N_t feedback shift register. A simple counter creates the control signals. Pipeline registers can be added between butterfly elements and between stages. Registers are also added inside the complex multipliers to reduce the critical path through the summation to the multiplier. The total latency is approximately $N + 4(\log_4 N - 1)$ cycles.

3. FPGA-Based Implementations and Optimizations

3.1. Specifications, Tool Flow, and Verification. Both of these FFT architectures were implemented with generic synthesizable VHDL code and verified with simulation against Matlab scripts using Modelsim. Synplify or XST was used to perform the synthesis, and ISE was used for place and route and implementation. The architectures were optimized to achieve maximum throughput with minimal area (slices). The tools and development environment used are shown in Table 2.

3.2. General Optimization Methods. Some general optimization measures were performed, including FSM encoding, retiming, and CAD-related optimizations. Since the FFT processors were targeted to Xilinx Spartan-3 and Virtex-4 FPGAs (as well as synthesized for Virtex-E FPGAs), the SRL16 component, which can implement a 16-bit shift register within a single LUT, was inferred as much as possible to preserve LUTs. This particularly helped the R2²SDF architecture because of the large number of shift registers. R4SDC also benefited from SRL16 components in its commutator registers. Block RAMs were used to store twiddle factors, which dramatically reduced the combinational logic utilization.

3.3. Architecture-Specific Optimization. A number of architecture-specific optimizations were used. For both architectures, a complex multiplication technique was used. Usually, a complex multiplication is computed as:

$$(a + bi) \times (c + di) = a \times c - b \times d + (a \times d + b \times c)i. \quad (10)$$

This requires 4 multiplications and 2 add/suboperations. As is well known, the equation is simplified to save one multiplier:

$$(a + bi) \times (c + di) = [a \times (c + d) - (a + b) \times d] + [a \times (c + d) + (a - b) \times c]i. \quad (11)$$

This requires only 3 multiplications and 5 add/suboperations. Pipeline registers were also added in order to avoid the long critical path brought by the connection of real adders and multipliers. Figure 7 shows the pipeline stages inserted which were effective to reduce the critical path (REG means pipeline register).

3.3.1. R4SDC Optimization. The R4SDC has a complex controller, which creates a long critical path. By observing that all stages have the same control bits but have different sequences, using a ROM with an incremental address was

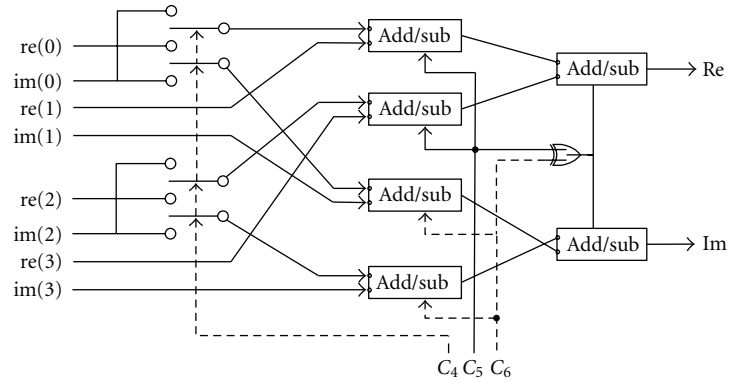


Figure 2: R4SDC butterfly element of stage t.

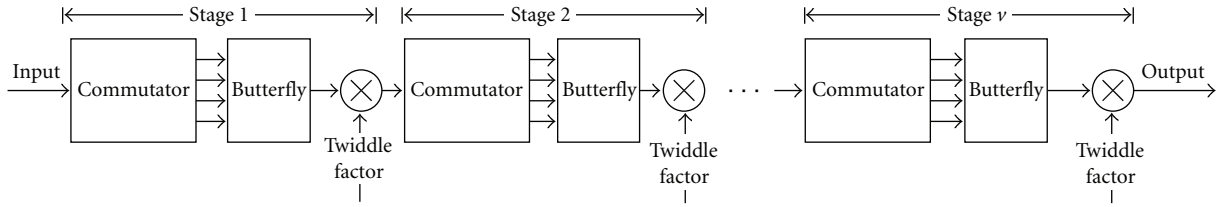


Figure 3: N-point R4SDC pipeline FFT processor architecture.

a simpler solution than using a complex FSM. Pipeline registers were also added to the butterfly elements, multipliers, and between stages. Figure 8 illustrates the addition of pipeline registers to cut the critical path efficiently within the butterfly element. Since two continuous add/subelements bring about a long propagation path, they were split using pipelining. Figure 9 shows the addition of pipeline registers between majority elements and between stages. For timing purposes, the applicable control signals were also buffered.

There were some special measures taken into account within controller in order to keep proper timing of signals. Twiddle factors should also be delayed to cope with the delayed sequence.

3.3.2. R2²SDF Optimization. Due to its simple control requirements, a simple counter was sufficient as the entire controller for the R2²SDF. To speed up the controller, a fast adder could potentially be faster than a simple ripple-carry adder. However, due to the small number of stages ($\log_4 N$), no substantial savings were found for a fast adder. Pipeline registers were added between major elements and also between stages. Note that the R2²SDF is not suited for adding pipeline registers within individual butterfly elements, because this would break the timing for the data feedback path. Figure 10 presents the pipelined stages. Note that registers were only added between element units; in addition, registers were added as necessary to keep the control signals properly timed.

4. Rounding Scheme and SQNR

Due to finite wordlength effects, the implemented FFTs always scaled by $1/N$ at the output of the design. This scaling factor was distributed as divide-by-two operations throughout each stage to reduce error propagation. As is well known, truncation or conventional rounding (which is denoted as round-half-up) will bring a notable quantization error bias in divide-by-two operations, and this bias will accumulate throughout the processing chain [5]. To alleviate the bias, three unbiased rounding methods are investigated for division by two.

Sign Bit-Based Rounding. In this scenario, if the MSB of the number to be divided is 0 (i.e., positive number) it is rounded-half-up. This will have a positive bias. On the other hand, if the MSB is 1 (i.e., negative number) it is truncated, leaving a negative bias. Assuming that the positive and negative numbers are uniformly distributed, this approach will lead to an unbiased rounding scheme. However, selecting the bias based on the MSB implies that these two rounding methods coexist in a single rounding position, which requires extra hardware. This increases the critical path, harming the performance. So it is not chosen.

Randomized [7]. In this scenario, if the bit to be rounded is 1, a random up or down rounding is performed. If it is 0, the same rounding scheme as done previously is performed. From the statistical point of view, no bias exists. But this method requires a random bit generator and a long accumulation time, requiring big extra hardware resources and significantly affects the performance. So it is not implemented.

Balanced Stages Rounding [11]. This rounding method explores balancing between stages. Round-half-up and truncation are used in an interlaced fashion, as shown in Figure 11.

Table 3: SQNR with different FFT sizes.

	FFT size	Input data width	Twiddle factor width	Stage number	SQNR (dB)
R4SDC	16	16	16	2	82.29
	64	16	16	3	73.49
	256	16	16	4	67.47
	1024	16	16	5	61.25
$R2^2SDF$	16	16	16	2	81.82
	64	16	16	3	74.47
	256	16	16	4	68.22
	1024	16	16	5	62.68

Table 4: Implementation results on Spartan-3 devices.

	Point size	Input data width	Twiddle factor width	Slices	Block RAM	Max. speed (MHz)	Latency (cycles)	Transform time Cycles	Transform time Time (μs)	Throughput (MS/s)	Throughput/area (MS/s/slice)
R4SDC	16	16	16	468	2	108.20	21	16	0.15	108.20	0.231
	64	16	16	952	2	107.23	73	64	0.60	107.23	0.113
	256	16	16	1990	3	111.98	269	256	2.76	111.98	0.056
	1024	16	16	4409	8	123.84	1041	1024	8.27	123.84	0.028
$R2^2SDF$	16	16	16	427	2	121.24	22	16	0.13	121.24	0.284
	64	16	16	810	2	98.14	74	64	0.65	98.14	0.121
	256	16	16	1303	3	98.73	270	256	2.59	98.73	0.076
	1024	16	16	2802	8	95.25	1042	1024	10.75	95.25	0.034

Table 5: Implementation results on Virtex-4 devices.

	Point size	Input data width	DSP48	Slices	Block RAM	Max. speed (MHz)	Latency (cycles)	Transform time Cycles	Transform time Time (μs)	Throughput (MS/s)	Throughput/area (MS/s/slice)
R4SDC	16	16	4	530	1	236.7	21	16	0.07	236.7	0.447
	64	16	8	803	2	236.4	73	64	0.27	236.4	0.294
	256	16	12	1370	3	218.9	269	256	1.17	218.9	0.160
	1024	16	16	3064	8	219.2	1041	1024	4.67	219.2	0.072
$R2^2SDF$	16	16	4	517	1	237.9	22	16	0.07	237.9	0.460
	64	16	8	779	2	236.7	74	64	0.27	236.7	0.304
	256	16	12	1234	3	236.7	270	256	1.08	236.7	0.192
	1024	16	16	2256	8	235.6	1042	1024	4.35	235.6	0.104

Table 6: Performance comparison versus prior art on Virtex-E devices.

FFT Design	Point size	Input data width	Twiddle factor width	Slices	Block RAM	Max. speed (MH)	Latency (Cycle)	Transform time Cycles	Transform time Time (μs)	Throughput (MS/s)	Throughput/area (MS/s/slice)
Amphion [8]	1024	13	13	1639	9	57	5097	4096	71.86	14.25	0.009
Xilinx [8, 9]	1024	16	16	1968	24	83	4096	4096	49.35	20.75	0.011
Sundance [10]	1024	16	10	8031	20	49	1320	1320	27.00	49.00	0.006
Suksawas $R2^2SDF$ [8]	1024	16	16	7365	28	82	1099	1024	12.49	82.00	0.011
Our $R2^2SDF$	1024	16	16	5008	32	95.0	1042	1024	10.78	95.00	0.019
Our R4SDC	1024	16	16	7052	32	94.2	1041	1024	10.87	94.20	0.013

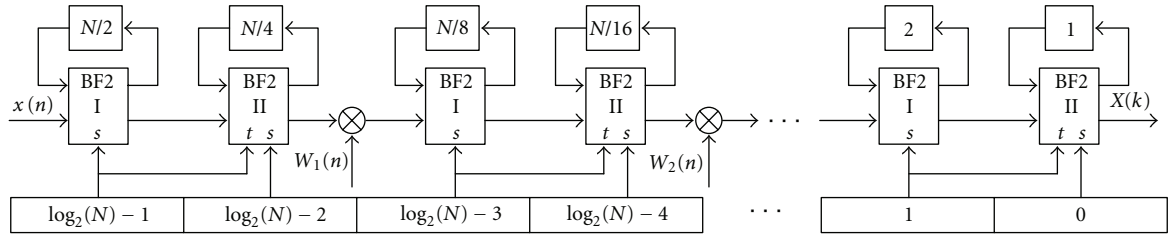


Figure 4: N-point R2^{2}SDF pipeline FFT processor architecture.

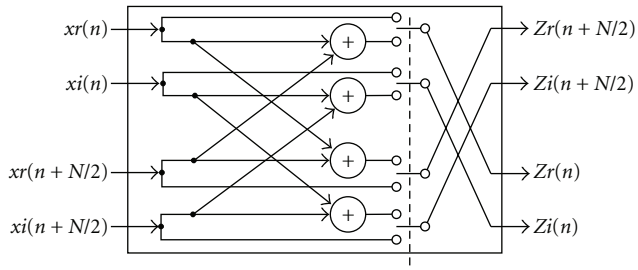


Figure 5: R2^{2}SDF BF2 I structure.

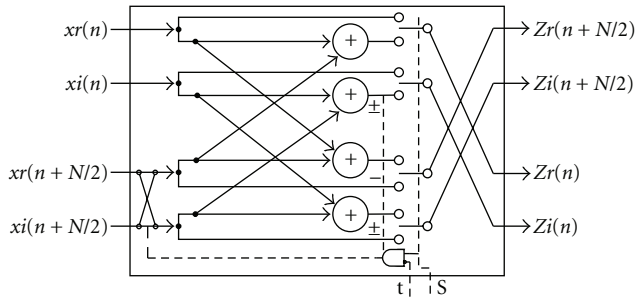


Figure 6: R2^{2}SDF BF2 II structure.

For an even number of stages, this will achieve the same result as the randomized approach, while having a smaller resource usage and simpler control. This scheme fits the R2^{2}SDF architecture particularly well, because the two butterfly elements within same stage of R2^{2}SDF can be naturally balanced. This method was chosen for the designs presented in the paper.

In order to compute the signal-to-quantization noise ratio (SQNR), random generated noise was used as the input to the pipeline FFT. A Matlab script generated double precision floating point FFT results, which were used as the true values. Figure 12 shows how they are compared with the fixed-point implementations. Random experiments were run several times and averaged to get a better error approximation.

5. Results and Analysis

5.1. SQNR Results. Figure 13 shows the SQNR results with different rounding schemes (balanced stages, truncation, and round-half-up), for R4SDC and R2^{2}SDF, respectively, for a 16-bit data width (input data, twiddle factors, and output data are 16 bits). The balanced stage rounding typically improved the SQNR by 1-2 dB. The balanced stages scheme gives better SQNR, because it leverages the randomness between stages. The truncation and round-half-up only reserve half of the information.

Table 3 presents the SQNR results as they vary with FFT size. The larger the FFT, the worse the SQNR due to the longer processing chain. Both architectures gave comparable results in terms of SQNR. It is clear that larger data widths will also give better SQNR but will increase area and critical path. A 16-bit wordlength is a sufficient choice for many signal processing applications.

The FFT architectures with smaller wordlengths than 16 bits are also implemented. The example in the Figure 14 shows the R4SDC architecture for an $N = 1024$ point FFT. Every bit of word length increment brings about 6 dB of SQNR gain.

5.2. Implementation Results. Table 4 gives the performance results of both architectures with different FFT sizes on Spartan-3 FPGAs (90 nm) [12]. The R2^{2}SDF achieved a smaller area and better throughput per area than the R4SDC. Due to the pipeline design, the maximum clock frequency did not change drastically with FFT size for either design. As expected the throughput per area decreases for larger FFT sizes, which require more stages and area.

Table 5 presents the results on Virtex-4 FPGAs (90 nm) using a 16-bit wordlength. Virtex-4 FPGAs have hardwired DSP modules called DSP48 blocks, which are high-speed modules optimized for signal processing operations such as multiply-accumulate, and FIR filtering. By utilizing these DSP48 blocks, the maximum clock frequency increased substantially over the Spartan-3 devices.

Comparisons with prior art are shown in Table 6, which shows publicly available pipeline FFT implementations on FPGAs from literature. For fair comparison, since many of the prior art implementations were implemented on Virtex-E FPGAs (180 nm), the designs are also implemented on Virtex-E. The best performance for the R2^{2}SDF method for a 16-bit 1024-point FFT was published by Sukhsawas and Benkrid in [8]. They used Handel-C as a rapid prototype language and implemented the design on Virtex-E FPGAs. They achieved 82 MHz maximum clock frequency and 7365 slices, giving a throughput per area ratio of 0.011 Msamples/s/slice.

On the Virtex-E, our R2^{2}SDF achieved better performance of 95 MHz and a smaller area of 5008 slices, giving a superior throughput per area ratio of 0.019 Msamples/s/slice.

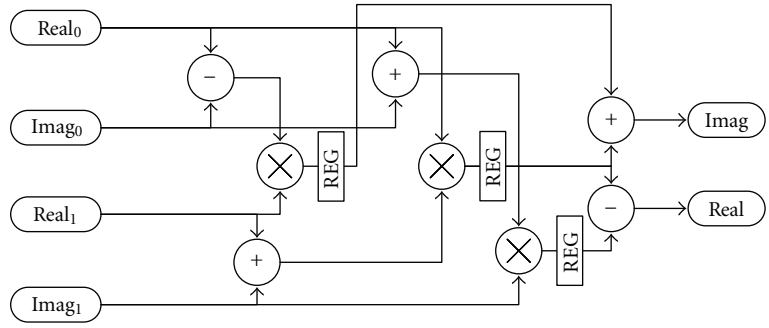


Figure 7: The pipelined complex multiplier.

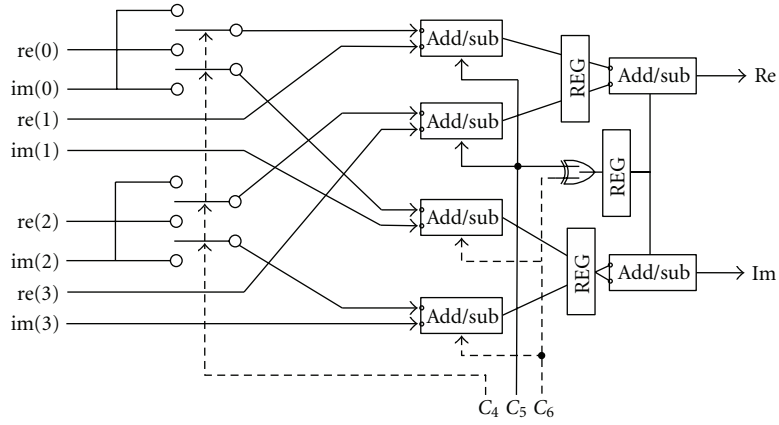


Figure 8: Adding pipeline registers to R4SDC butterfly element.

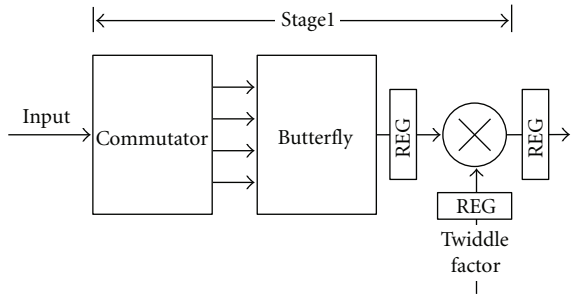


Figure 9: Adding pipeline registers between elements and stages.

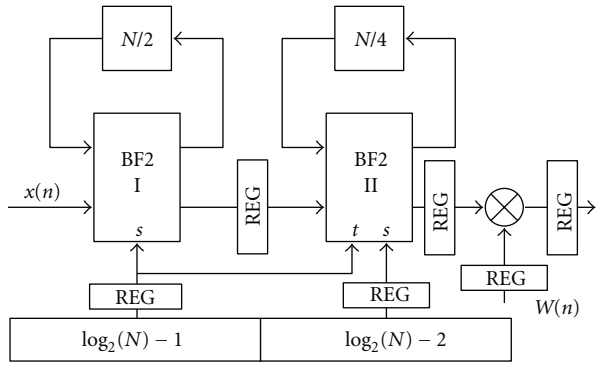


Figure 10: Adding pipeline registers for R2²SDF.

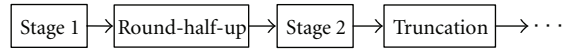


Figure 11: Balanced stages rounding.

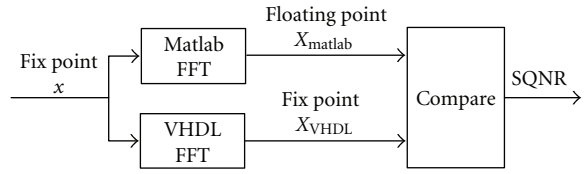
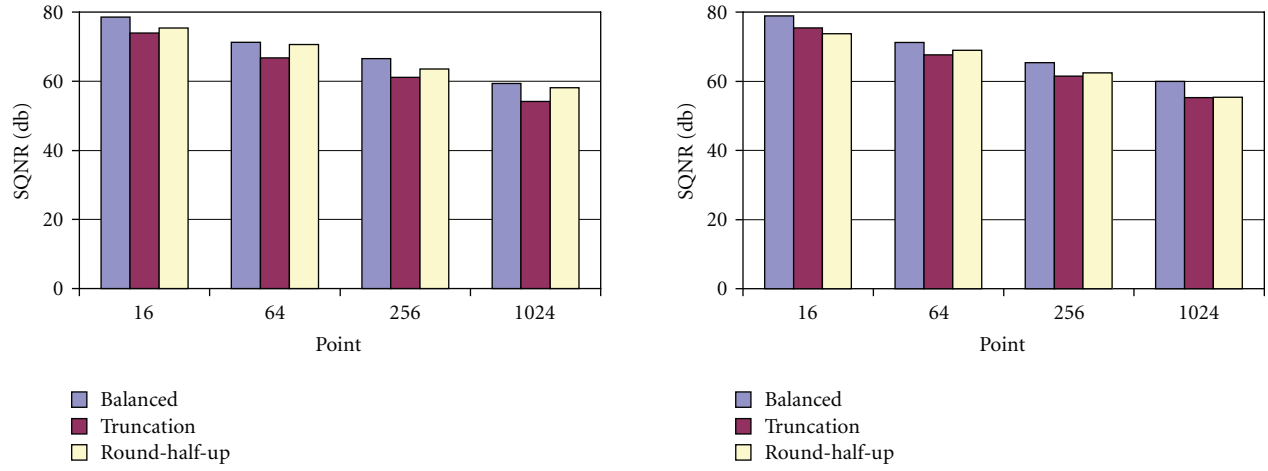


Figure 12: SQNR calculation.

Our R4SDC architecture was also superior to prior art, running at 94.2 MHz and using 7052 slices, a throughput per area ratio of 0.013 Msamples/s/slice.

Another point of reference is the Xilinx FFT IP core. For comparison sake, the IP core for Virtex-E is shown in the table. The Virtex-E core shows four times the latency (4096) in cycles due to its internal architecture. Its throughput per area ratio is also only 0.011 Msamples/s/slice. Note that all comparisons for throughput per area do not take into account block RAMs, though each of the designs had a similar number of required block RAMs. However, the Xilinx FFT DSP core could perform better with new Xtreme technology: on Virtex 4 device 4vsx25-10, 1024-point FFT



(a) R4SDC (b) R2^{2}SDF

Figure 13: Rounding effects on SQNR.

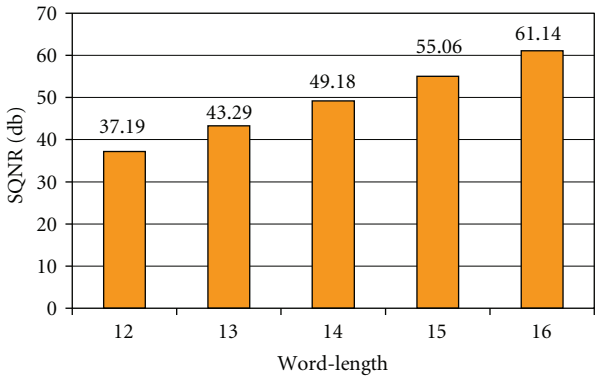


Figure 14: SQNR variation with different word lengths.

could be finished within 2.85 nanoseconds in best case, while cost 2141 Slices, 7 block RAMs, and 46 Xtreme DSP slices [13].

6. Conclusions

In this paper, optimized implementations of R4SDC and R2^{2}SDF pipeline FFT processors on Spartan-3, Virtex-4, and Virtex-E FPGAs are presented. The 16-bit 1024-point FFT with the R2^{2}SDF architecture had a maximum clock frequency of 95.2 MHz and used 2802 slices on the Spartan-3. The R4SDC ran at 123.8 MHz and used 4409 slices on the Spartan-3. On Virtex-4 device, the numbers became 235.6 MHz and 2256 slices for R2^{2}SDF and 219.2 MHz and 3064 slices for R2^{2}SDF, respectively. Different rounding schemes were analyzed and compared. SQNR analysis showed the balanced stages rounding scheme gave high SQNR with small overhead. The SQNR will gain around 6 dB with every bit increment of word length.

The R2^{2}SDF architecture outperformed the R4SDC architecture in terms of throughput per area, a measure of efficiency, for the 1024-point FFT. This is due to its simpler controller and compatibility with pipelining insertion. Both architectures have comparable maximum clock frequency and SQNR with the balanced stages rounding scheme.

References

[1] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Upper Saddle River, NJ, USA, 1975.

[2] E. H. Wold and A. M. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI implementation," *IEEE Transactions on Computers*, vol. 33, no. 5, pp. 414–426, 1984.

[3] G. Bi and E. V. Jones, "A pipelined FFT processor for word-sequential data," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 1982–1985, 1989.

[4] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pp. 766–770, Honolulu, Hawaii, USA, April 1996.

[5] J.-Y. Oh and M.-S. Lim, "New radix-2 to the 4th power pipeline FFT processor," *IEICE Transactions on Electronics*, vol. E88-C, no. 8, pp. 1740–1746, 2005.

[6] T. Sansaloni, A. Pérez-Pascual, V. Torres, and J. Valls, "Efficient pipeline FFT processors for WLAN MIMO-OFDM systems," *Electronics Letters*, vol. 41, no. 19, pp. 1043–1044, 2005.

[7] S. Johansson, S. He, and P. Nilsson, "Wordlength optimization of a pipelined FFT processor," in *Proceedings of the 42nd Midwest Symposium on Circuits and Systems*, vol. 1, pp. 501–503, 1999.

[8] S. Sukhsawas and K. Benkrid, "A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '04)*, pp. 229–232, February 2004.

[9] Xilinx, Inc., "High-Performance 1024-Point Complex FFT/IFFT V2.0," San Jose, Calif, USA, July 2000, http://www.xilinx.com/ipcenter.

[10] Sundance Multiprocessor Technology Ltd., 1024-Point Fixed Point FFT Processor, July 2008, http://www.sundance.com/web/files/productpage.asp?STRFilter=FC200.

[11] P. Kabal and B. Sayar, "Performance of fixed-point FFT's: rounding and scaling considerations," in *Proceedings of the*

IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '86), pp. 221–224, 1986.

[12] B. Zhou and D. Hwang, "Implementations and optimizations of pipeline FFTs on Xilinx FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 325–330, 2008.

[13] Xilinx, Inc., "Xilinx Fast Fourier Transform V3.2 Product Specification," San Jose, Calif, USA, January 2006.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 863630, 11 pages
doi:10.1155/2009/863630

Research Article

Answer Set versus Integer Linear Programming for Automatic Synthesis of Multiprocessor Systems from Real-Time Parallel Programs

Harold Ishebabi,[1] Philipp Mahr,[1] Christophe Bobda,[1] Martin Gebser,[2] and Torsten Schaub[2]

[1] *Professorship for Computer Engineering, Institute for Computer Science, University of Potsdam, August-Bebel-Strasse 89, D-14482 Potsdam, Germany*
[2] *Professorship for Knowledge Processing and Information Systems, Institute for Computer Science, University of Potsdam, August-Bebel-Strasse 89, D-14482 Potsdam, Germany*

Correspondence should be addressed to Harold Ishebabi, ishebabi@cs.uni-potsdam.de

Received 12 March 2009; Accepted 24 August 2009

Recommended by Lionel Torres

An automated design approach for multiprocessor systems on FPGAs is presented which customizes architectures for parallel programs by simultaneously solving the problems of task mapping, resource allocation, and scheduling. The latter considers effects of fixed-priority preemptive scheduling in order to guarantee real-time requirements, hence covering a broad spectrum of embedded applications. Being inherently a combinatorial optimization problem, the design space is modeled using linear equations that capture high-level design parameters. A comparison of two methods for solving resulting problem instances is then given. The intent is to study how well recent advances in propositional satisfiability (SAT) and thus Answer Set Programming (ASP) can be exploited to automate the design of flexible multiprocessor systems. Integer Linear Programming (ILP) is taken as a baseline, where architectures for IEEE 802.11 g and WCDMA baseband signal processing are synthesized. ASP-based synthesis used a few seconds in the solver, faster by three orders of magnitude compared to ILP-based synthesis, thereby showing a great potential for solving difficult instances of the automated synthesis problem.

1. Introduction

In order to build flexible systems that can be adapted to applications, researchers have explored FPGA-based multiprocessor systems in an attempt to exploit both high-level parallelism in applications and the flexibility of reconfigurable devices, targeting both single [1, 2] and multiFPGA platforms [3, 4].

The process of implementing such systems is a very complex undertaking, consisting of phases such as the design of constituent IP-blocks (e.g., processors, memories, and buses), task mapping and architecture determination (high-level synthesis), low-level system integration, and finally, FPGA synthesis and placement and routing. The focus of this work is on task-mapping and high-level synthesis, and builds up on a design platform that targets system integration and synthesis [1].

An automated architecture synthesis methodology based on combinatorial optimization is used to simplify the design process. This methodology addresses the problem of determining application-specific optimum system architectures as well as mapping and scheduling corresponding parallel programs.

Often, an optimum solution requires the sharing of processor resources between tasks, necessitating the use of a task scheduler, whose impact on the overall solution must be considered. In cases where cooperative schedules suffice, that is, for applications without strict timing requirements,

the resulting analysis during optimization is straightforward. This is because the overhead is easy to compute. In that case, the overall optimization objective is simply to minimize the overall execution time of the application, the makespan, or alternately, to explore area-throughput-power tradeoff.

However, applications which impose deadline guarantees for periodic tasks may require preemptive schedulers. In such situations, one must determine how often task switching actually takes place, depending on task priorities and the schedule. The reason is that the schedule has a direct impact on the overall execution time, and hence on the optimum task mapping and resource allocation. Moreover, the optimization process must take into account schedulability constraints, because some task mappings may not guarantee deadlines.

The method presented in this paper considers the effect of fixed-priority scheduling during architecture synthesis. This covers a broad spectrum of embedded applications, with and without real-time requirements. Experimental results for parallel implementations of IEEE 802.11g and WCDMA signal processing algorithms provide a proof of concept.

Because the structure of targeted parallel programs as well as task deadlines is known a priori, it is possible to synthesize multiprocessor architectures that are optimum for the programs. It can be experimentally shown that such customized architectures are superior compared to domain-specific ones. That fact is important because embedded applications exhibit a wide diversity with respect to the complexity of their algorithms, the rate at which the algorithms need to operate, as well as their memory and intertask communication patterns. The consequence is that it is virtually impossible to find a good architecture that can meet the requirements of a wide range of algorithms. Customized architectures are therefore vital.

Since it is desirable to customize a system for a target embedded parallel application, automated tools for design space exploration are required to cope with the complexity. Whereas a skilled engineer can effectively utilize workbench-based tools [5, 6] to design a feasible architecture, the sheer number of design parameters renders a disciplined exploration infeasible. Further, it has been shown in [7] that often no consistent trend with respect to design objectives can be observed when design parameters are systematically changed, where the value of the objective function can increase or decrease by more than two orders of magnitude in an apparently random manner when moving between adjacent sets of parameters in the design space. Moreover, results obtained can be counter-intuitive. Consequently, even experienced designers cannot effectively execute a guided exploration based on their expertise because it is not easy to predict the outcome of parameter variation so that superior design points can be easily missed. Those results underline the need for an automated approach.

To enable an automatic exploration, design parameters and the objective must be mathematically modeled. Since this is inherently a combinatorial optimization problem, it is natural to model the problem as such and solve it using Integer Linear Programming (ILP). However, the large number of design parameters that needs to be considered at the system level leads to a huge number of variables and constraints, thereby posing a serious challenge for ILP solvers that manifests itself in very long synthesis time. But to be useful, this automated synthesis approach must be fast in order to facilitate a systematic exploration.

On the other hand, recent advances in propositional satisfiability (SAT) methods [8] have spurred significant improvements in methods for Answer Set Programming (ASP) [9, 10]. ASP is a form of declarative programming oriented towards difficult search problems. Given the success that has been reported for solving such problems, it is interesting to study the effectiveness of these methods for speeding up the automated synthesis problem. Therefore, this paper compares ILP versus ASP-based high-level synthesis both in terms of synthesis runtime and the quality of synthesized architectures. The study uses parallel implementations of baseband signal processing chains for IEEE 802.11g and WCDMA wireless standards.

The rest of this paper is organized as follows. Summaries on related work and on our design flow are given in Sections 2 and 3, respectively. The ILP model for optimization is presented in Section 4, followed by a model of the problem in ASP semantics in Section 5. Finally, a comparison of the two methods and concluding remarks are given in Sections 6 and 7, respectively.

2. Related Work

Mathematical modeling and tool automation for synthesizing multiprocessor systems are an area that has been extensively discussed in the literature using techniques ranging from combinatorial optimization, through dynamic programming, simulated annealing, evolutionary algorithms to application-specific heuristics.

The vast majority of related work in this area have the drawback that the design space is preconstrained by fixing the architecture first, followed by task mapping and scheduling [11–13]. Since no customization is possible, resulting solutions are not optimum because the optimum application-specific system is imposed by the nature of the application, as dictated by operations performed within its tasks, as well as by intertask communication pattern. Often preconstraining is employed to overcome the complexity of the design space.

Some approaches attempt to reduce the design complexity by eliminating design dimensions [14, 15] thereby limiting the optimality of architectures. Advanced related works [16–18] recognize and address this aspect and attempt to solve the subproblems simultaneously. However, these approaches separately consider subproblems, effectively preconstraining the design space, albeit to a much smaller degree. Other approaches such as [19] randomly search for a feasible solution and may not lead to an optimum solution.

Scheduling during or after mapping too has been extensively treated in the literature [19, 20]. There also

have been efforts to map tasks in a way that optimizes for power [21], reduces chip temperature at run time [22], or minimizes interprocessor communications [23]. Dynamic mapping techniques have also been introduced with objectives such as temperature management [24] and performance optimization through adaptive mapping [25]. These approaches however consider scheduling on fixed architectures.

In contrast, in order to synthesize application-specific optimum architectures, it is important to simultaneously (i) select processors, (ii) map and schedule tasks to them, and (iii) select one or several networks for communications, such that design constraints and objectives are met. This avoids the problem of preconstraining the design space, leading to globally optimum architectures.

The contribution of the method presented in this paper is a comprehensive mathematical model that can be used for automated design space exploration without limiting the design space as well as a comparison of candidate approaches to tackle the problem.

3. The Design Flow

Figure 1 depicts the flow. The input to the flow is a parallel program, and optionally information on task periods. The application is simulated and analyzed to obtain intertask data traffic and task precedence information. This information is used to specify an instance of an ILP problem or an ASP program. Similarly to other related work in this area, the other input to the design flow is information on available processing elements and communication networks, as well as their costs and constraints. In our approach, the design space is not pre-constrained, and the problem dimensions are not ranked. This ensures the optimality of found solutions. For real-time systems, it is often sufficient to meet timing constraints so that the interest is not to find the fastest solution. In such situations, the flow can be used to find the smallest system instead.

The solution generated by the ILP/ASP solver is used to generate an abstract description of the system, which is passed to further tool-chains described in [1] to generate the configuration bit stream. Because postsynthesis results could deviate from initial cost models used, new cost models can optionally be extracted after placement and routing to start a new iteration.

4. ILP Model

The ILP model used for automated synthesis in this work consists of two major parts. The first part covers constraints that establish the system functionality, without any regards to deadlines [7]. The second part covers scheduling and the optimization objective and is the focus of this paper.

The following notation is used. $I_i \in \{I_0, \ldots, I_n\}$ is a task, $J_j \in \{J_0, \ldots, J_m\}$ is a processor, and x_{ij} is a Binary Decision Variable (BDV). $x_{ij} = 1$ means that task I_i is mapped on processor J_j, $x_{ij} = 0$ otherwise.

The objective function for a terminating parallel program, or for one period of a non-terminating program, is expressed as

$$\min\left(\sum_{i=0}^{n}\sum_{j=0}^{m} x_{ij} \cdot T_{ij} + T_{\text{net}} + T_{\text{switch}}\right), \tag{1}$$

where T_{ij} is the execution time of task I_i on processor J_j, and T_{net} is the cost in time of using communication resources as described in [7]. T_{switch} is the cost in time of task switching which depends on several factors as described in subsequent sections

$$T_{\text{switch}} = \sum_{l=0}^{2^n-2}\sum_{j=0}^{m} \gamma_{lj} \cdot F_{lj} \cdot \left(T_{s_{lj}} \cdot t_j + O_j\right). \tag{2}$$

4.1. The Processor Architecture. This is the actual cost of switching context, which depends on the memory and on the microarchitecture, as well as on the mechanism for context switching (i.e., under software or with hardware support). The coefficient t_j captures this cost and can be reliably precomputed for processors of interest. Whether this cost is actually incurred depends on task mapping as discussed in Section 4.4.

4.2. The Kernel/OS. The kernel or real-time OS introduces a control overhead due to scheduling (polling, moving tasks between run and delay queues, etc.). In this formulation, it is assumed that kernel/OS is already selected and is fixed, for each of the processors in the design space (i.e., OS selection is not a part of the optimization problem, so that the associated cost is coupled to the selected processor). This is however not a limitation because, when desired, instances of the same processor running different operating systems or microkernels can be specified in the ILP problem to extend the design space.

The overhead is caused by the clock interrupt handler interfering the execution of application tasks because of its higher priority. This increases the number of task switching and the response time of application tasks. The coefficient O_j in (2) captures the latter cost, the clock-handler time. The analysis in [26] describes how the clock-handler time can be estimated for fixed-priority schedulers. Because the overhead is kernel/OS-specific, and may depend on task mapping, the computation/estimation of O_j in the problem formulator (Figure 1) is implemented in an extensible way to support new kernel/OS models.

4.3. The Schedule/Task Switching. The schedule determines how often task switching takes place as captured by the coefficient $T_{s_{lj}}$ in (2). $T_{s_{lj}}$ is the number of task switching that is incurred for the duration of the application, or for one period, when a particular group of tasks with the index l is mapped on a processor J. One can distinguish between three major scheduler categories: cooperative, fixed-priority preemptive, and deadline-driven schedulers.

In simple cooperative schedulers (e.g., cyclic executives), there is no preemption, so that $T_{s_{lj}} = 0$. The overhead

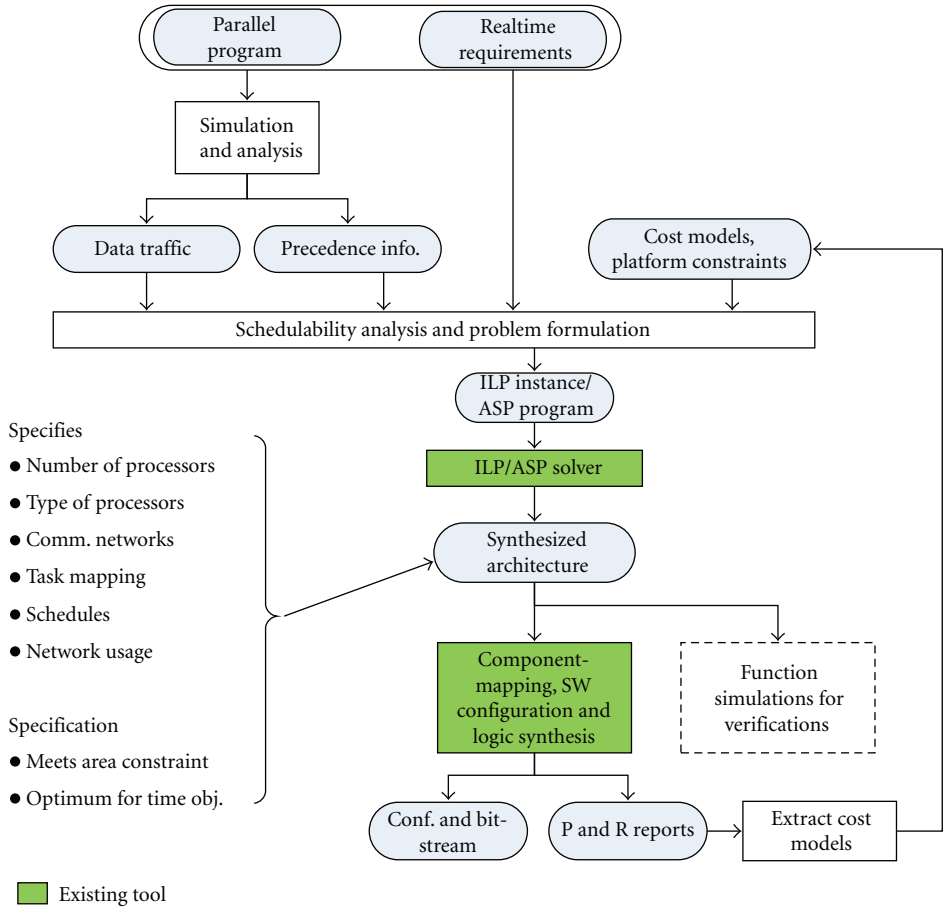


Figure 1: Architecture synthesis flow.

incurred when a task begins and ends is already captured by T_{ij} in (1) as part of function-call overhead.

Preemptive schedulers often require task priorities to make decisions by letting higher-priority tasks run first. This can improve performance if critical tasks are assigned higher priorities. Priorities can be fixed or dynamic depending on whether priorities can change at runtime. The exception to this distinction are measures against priority inversion. Even though such measures do change priorities dynamically, the changes are temporary to otherwise fixed priorities.

Deadline-driven schedulers change task priorities dynamically and have the advantage that deadlines can be guaranteed at higher CPU utilization compared to fixed-priority scheduling. In either case, $T_{s_{ij}}$ is a function of the number of context switching, and its usage in the ILP model is the same. This section discusses how the worst-case number of context switching can be estimated for fixed-priority schedulers, which are more typical in real-time embedded systems.

For these schedulers, $T_{s_{ij}}$ is equal to the number of interferences due to higher priority tasks and is obtained from Rate Monotonic Analysis (RMA) [27] within the problem formulator. The RMA in the formulator currently supports tasks with single deadlines, and which have fixed durations and nonvarying periods. However, the implementation is easily extensible to support flexible RMA models. Such models can be adapted to applications with arbitrary, multiple, or internal deadlines [28]. Future extensions will affect the computation of $T_{s_{ij}}$ only. The ILP model for synthesis remains unaffected.

RMA is conducted for all possible task groups and mappings. The output of RMA, the response r, is used to estimate $T_{s_{ij}}$. Algorithm 1 shows how the parameter is computed.

The first line computes a scheduling table for a group G_l of tasks, if the group would be mapped on a processor J_j. The rows in the scheduling table contain the priority of a tasks in the group, together with their deadlines, periods and execution times on the processor. The priorities are computed according to [27].

The table is initially filled in arbitrary order with task information, and the priorities are initially zero. The rows are then sorted in two passes according to periods and deadlines.

Prior to sorting, deadlines are relaxed according to Algorithm 2 in order to avoid pessimistic schedulability analysis. The analysis assumes that all tasks are released at the same time. If the response of a task is then greater than

```
(1)  Table T = createSchedulingTable(G_l, J_j,
     task time T_ij, task deadlines D_i, task periods T_i)
(2)  for all I_i ∈ G_l do
(3)     Response r_i = computeResponse(I_i,T)
(4)     F_lj = 1
(5)     if r > D_i then
(6)        F_lj = 0
(7)     end if
(8)  end for
(9)  r = computeLargestResponse(G_l,T)
(10) T_slj = ⌈r/period of highest priority task in G_l⌉
```

Algorithm 1: Determining context switching cost.

```
(1)  for each I_i ∈ G_l, set parent deadline to zero
(2)  while G_l is not empty do
(3)     if G_l is circular then
(4)        select I_i ∈ G_l such that T_i ≤ T_i2 for all I_i2 ∈ G_l
(5)     else
(6)        select I_i ∈ G_l with no parent in G_l
(7)     end if
(8)     for each child I_i2 of I_i in G_l do
(9)        if parent deadline < D_i then
(10)          set parent deadline to D_i
(11)       end if
(11)    end for
(12)    if I_i had a parent in G_l then
(13)       D_i+ = parent deadline
(14)    end if
(15)    remove I_i from G_l
(17) end while
```

Algorithm 2: Deadline relaxation during RMA.

the deadline as shown in Algorithm 1, the schedule is declared infeasible. However, if there is a precedence relationship, not all tasks are released at the same time. In particular, if there is an edge between I_{i_1} and I_{i_2}, then I_{i_2} cannot start until I_{i_1} has finished. Therefore, the deadline D_{i_2} needs to be relaxed to $D_{i_2} = D_{i_2} + D_{i_1}$ to reflect the fact that there is an offset from the release time of its parent task.

Relaxation proceeds by selecting the most critical task. If the subgraph G_l of the application graph G is circular, it is not immediately obvious which task is most critical because of circular producer-consumer relationships. Therefore, the algorithm selects the task in G_l with the shortest deadline. Because a critical task is eliminated from G_l at the end of each iteration, this selection has the effect of introducing cuts in G_l which removes circular paths. Otherwise, if G_l has no cycles, the most critical task in G_l is the one that does not consume data from other tasks in the subset. Before a task is removed from G_l, its deadline is relaxed by adding the deadline of its already removed parent, if the task *had* one. If the task had multiple parents, then the largest of its parents' deadlines is selected according to lines 8–12. This relaxation does not impose any limitation to the type of application graph that can be handled by the synthesis flow.

After relaxation, sorting begins. The first pass sorts tasks according to periods in ascending order. If the tasks have different periods, and G_l is at least partially connected, then a critical assumption is made that *if there is a node in G_l with a period less than that of any of its parent, then the edge between the node and the parent represents a weak precedence meaning that the corresponding task can execute without receiving data from its parent*. An example would be a task that infrequently obtains new parameters from another task for its internal computations. Otherwise, the application graph is faulty, and the resulting schedule is meaningless. Partial connectedness in this context means that G_l contains at least one nontrivial connected subgraph.

The second pass sorts the table again according to deadlines, but the sorting is done only within rows containing the same period. Since deadlines have been previously relaxed, no distinction with respect to precedence relationship between tasks needs to be made; if tasks I_{i_1} and I_{i_2} have no direct or indirect precedence, then I_{i_1} must finish before I_{i_2} if $D_{i_1} < D_{i_2}$, because I_{i_1} needs to finish earlier; if there is a precedence relationship, then $D_{i_1} < D_{i_2}$ because of the relaxation step, and I_{i_1} must appear before I_{i_2}. Indirect precedence in this context means that there is a path from I_{i_1} to I_{i_2} via one or more intermediate tasks. Therefore, because second sorting is only done within rows, tasks with shorter periods appear before those with longer periods regardless of whether or not latter tasks have shorter deadlines.

The sorting is topological and is thus not unique. Moreover, if the group represents a nonconnected graph, then the result after sorting is a partial order. The final order after sorting reflects the priorities in descending order, which are assigned by a simple enumeration.

With the table in place, the algorithm proceeds to compute the response time of each task in the group G_l according to the scheduling table. The response is computed recursively according to [27] as

$$r_i = T_{ij} + \sum_{\forall I_{i_h} \in G_l \mid \text{priority}(I_{i_h}) > \text{priority}(I_i)} \left\lceil \frac{r_i}{\mathcal{T}_{i_h}} \right\rceil \cdot T_{i_h j}. \quad (3)$$

This model of response time differs slightly from that of Liu and Layland [27] in that no bound in task blocking time due to safeguarding against priority inversion is included. This is because the programming model used here is message passing so that tasks do not share protected data such that semaphore-based synchronization for variables or memory locations is not required.

The analysis then concludes by comparing the response time against the execution time in lines 4–7 of Algorithm 1. The scheduling feasibility parameter F_{lj} in (2) is set to 0 if the response time is larger. Finally, the number of task switching is estimated in line (10) from the response time of the lowest-priority task and the period of the highest-priority task. This worst-case estimate is conservative by making the assumption that all tasks in the group are always ready when released so that the lowest-priority task experiences

maximum interference. The use of the parameter F_{lj} is explained in the following subsection.

4.4. Task Mapping. Task mapping influences the switching costs in two ways: (i) by selecting the processor, the switching mechanism, and thus the cost, is determined and (ii) by grouping tasks on one processor, the optimum schedule that can be applied, and thus the number of task switching, is determined. Consequently, scheduling and task-mapping influence each other during optimization.

To include this cross-effect during optimization, two strategies can be followed: (i) integrate scheduling in an ILP solver so that a schedule is computed prior to cost calculation for a candidate mapping or (ii) precompute optimum schedules for all possible mappings, and integrate the schedules in the ILP formulation. In this work, we opted for the latter strategy as discussed in the previous subsection. This is because, by pre-computing the schedules, infeasible mappings can be eliminated to reduce the size of the ILP instance. For this purpose, a coefficient F_{lj} is used in the formulation in (2). This coefficient is computed in the ILP formulator during RMA. Its value is 1 if there is a feasible schedule for a group of tasks with the index l on processor J, and 0 otherwise. We next describe how F_{lj} is used to enforce feasibility constraints in the formulation.

Let $\mathcal{P}(I)$ be the power set of the task set $I = \{I_0, \ldots, I_n\}$. Let G_l be an element in the power set excluding the empty set, with $l = \{0, 1, \ldots, 2^n - 2\}$. Let $I_{\text{mapped}} \subset \mathcal{P}(I)$ be a set, so that each element contains one or more tasks that will be mapped on the same processor. The solution to the combinatorial optimization problem consists of the set I_{mapped}. Each element in I_{mapped} is associated with a task switching overhead as dictated by its schedule.

Now since I_{mapped} is not known at formulation time, a variable $\mathcal{M}_{lj}$ is introduced for each element G_l in the power set $\mathcal{P}(I)$. If $\mathcal{M}_{lj} = 1$, then $G_l \in I_{\text{mapped}}$. If $\mathcal{M}_{lj} = 0$, then G_l is not an element of I_{mapped}. Therefore, we insist that

$$\mathcal{M}_{lj} \leq F_{lj} \quad \forall\, \mathcal{M}_{lj}, \tag{4}$$

so that if and only if the mapping is feasible, then $\mathcal{M}_{lj}$ constitutes a degree of freedom during synthesis. We next describe how the decision variables $\mathcal{M}_{lj}$ and x_{ij} are linked through ILP constraints.

Recalling that $x_{ij} = 1$ implies that a task I_i is mapped on a processor J_j, it follows for any group G_l, $\mathcal{M}_{lj} = 1$ if and only if $x_{ij} = 1$ for all $I_i \in G_l$. This results into a logical constraint

$$\begin{aligned} &\mathcal{M}_{lj} = \left(x_{i_{l0}j} \wedge x_{i_{l1}j} \wedge \cdots \wedge x_{i_{lg}j}\right) \quad \forall\, \mathcal{M}_{lj} \\ &\text{with } G_l = \left\{I_{i_{l0}}, I_{i_{l1}}, \ldots, I_{i_{lg}}\right\}, \quad lg = |G_l| - 1. \end{aligned} \tag{5}$$

To transform the logical constraint into a linear form, two steps were applied. First, we specified that

$$\begin{aligned} &\mathcal{M}_{lj} = 0 \longrightarrow \left(x_{i_{l0}j} + x_{i_{l1}j} + \cdots + x_{i_{lg}j}\right) < |G_l|, \\ &\mathcal{M}_{lj} = 1 \longrightarrow \left(x_{i_{l0}j} + x_{i_{l1}j} + \cdots + x_{i_{lg}j}\right) = |G_l|. \end{aligned} \tag{6}$$

These two constraints insure that when a schedule is not feasible, then at least one $I_i \in G_l$ is not mapped on J_j. This implies that other groups which are either proper subsets of G_l, or which are not super sets of G_l, can be mapped on J_j, provided that they have a feasible schedule. The second step then is a set of inequalities that satisfy the specification in(6)

$$x_{i_{l0}j} + x_{i_{l1}j} + \cdots + x_{i_{lg}j} \leq |G_l| - 1 + \mathcal{M}_{lj}, \tag{7}$$

$$x_{i_{l0}j} + x_{i_{l1}j} + \cdots + x_{i_{lg}j} \geq |G_l| \cdot \mathcal{M}_{lj}. \tag{8}$$

With (4), (7), and (8), feasible mappings are guaranteed. The last step is to capture the switching cost of the groups in the objective. A contribution of a group to the switching cost is given by $\mathcal{M}_{lj} \cdot T_{s_{lj}}$. However, this contribution cannot be directly used in the objective function by taking the sum of all contributions from all groups. This is because, as it can be observed from (7) and (8), if a group G_l is mapped on a processor J_j, then the value of $\mathcal{M}_{lj}$ for all groups which are subsets of G_l is also one. Consequently, taking the sum of contributions directly would erroneously include the switching cost of subgroups.

To prevent this incorrect inclusion of switching costs, we need to specify that the switching contribution of a group should be counted only when $\mathcal{M}_{lj} = 0$ for all of its supersets. This leads to nonlinear terms in the objective function of the form $\mathcal{M}_{lj}(\mathcal{M}_{l_{s1}j} - 1)(\mathcal{M}_{l_{s2}j} - 1)\ldots$, where l_{s1} and l_{s2} are indices of supersets for a group with an index l.

To break the non-linearity, a BDV γ_{lj} is introduced for each $\mathcal{M}_{lj}$. Its value is 1 only when the group is mapped, and $\mathcal{M}_{lj} = 0$ for all supersets. To model this property, we note from (8) that the left-hand side of the inequality for a group G_{l_1} with $\mathcal{M}_{l_1 j} = 1$ is greater than the left-hand side of the same inequality for a group G_{l_2}, if $G_{l_2} \subset G_{l_1}$. Therefore, the following relationship holds:

$$0 \leq \frac{1}{|I|}\left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij}\right) < 1 \quad \forall\, G_l, J_j. \tag{9}$$

The first sum in the above relationship is the total number of tasks that have been mapped on J_j. The second sum is the size of a group, which is the same as the left-hand side of (8). The difference of the two sums is zero in two cases: (i) if nothing is mapped on J_j and (ii) if a mapped group has no superset G_{l_s} for which $\mathcal{M}_{l_s j} = 1$ for that specific mapping. The sum is greater than zero, if for G_l, there is a superset G_{l_s} with $\mathcal{M}_{l_s j} = 1$. This is because there is at least one decision variable x_{ij} with value 1 in the first sum, which is not present in the second. The largest value that the difference of the two sums can have is $|I| - 1$, so that the upper limit in (9) is 1.

We next exploit this relationship by specifying that

$$\begin{aligned} &\gamma_{lj} + \frac{1}{|I|}\left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij}\right) \leq 1, \\ &\gamma_{lj} + \frac{1}{|I|}\left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij}\right) \geq \mathcal{M}_{lj} - 1 + \frac{1}{|I|}. \end{aligned} \tag{10}$$

4.5. Processor-External Factors. Processor-external factors such as interrupts and data availability have a direct runtime impact on the schedule. The foregoing formulation has the limitation that it is based on worst-case assumptions in rate monotonic analysis. In particular, it is assumed that tasks in a group G_l with no precedence relationship can become ready at the same time.

With respect to data availability, the worst-case assumptions can be relaxed by taking into account in RMA when data can actually arrive depending on source-task-destination-task mapping, and on the selected communication network.

A possible relaxing solution is to compute offsets between release times of tasks with indirect precedence. For example, if there are three tasks such that the first sends data to the second, and the second to the third, and the mapping is such that the first and third are mapped on one processor, and the second on another, then there is an offset between release times of the first and third tasks. This offset is equal to the time needed for data to be sent to the second task, plus the response time of the second task, plus the time for the resulting data to be sent from the second task to the third task. A suitable ILP formulation that will not significantly increase the problem size needs to be found.

5. Answer Set Programming

Answer Set Programming (ASP) is different from procedural programming in that a problem is described using a formal language, and a solver finds a solution. A problem is presented as a logic program consisting of a set of atoms and rules [29]. An atom is a Boolean proposition about the problem universe; whereas rules specify relationships between the atoms. A solution to a program is called a stable model and tells which atoms are true [29]. This is similar to SAT problems if rules and stable models are perceived to be clauses and satisfying assignments, respectively. ILP models can be encoded into equivalent ASP programs. We opted to use the ASP solver clasp [9] whose grounding tool natively supports the encoding of linear constraints [30]. The native support eliminates the need of having to translate constraints into clauses, a procedure that can lead to a huge number of clauses [31]. The rest of this section describes how the ILP model from Section 4 is coded into an equivalent ASP program. The same notation is used so that variables from Section 4 now stand for atoms.

Linear inequalities are coded into rules whose general form is

$$b[v_0 = a_0, v_1 = a_1, \ldots, v_n = a_n]c, \tag{11}$$

where the syntax $v_i = a_i$, $a_i \in \mathbb{N}$ denotes the weight a_i of a variable v_i in a linear (in)equality, and the syntax $b[\cdots]c$, $b, c \in \mathbb{N}$ is a general form for constraining such that a coded constraint represents an equality if $b = c$, a less-than inequality if b is not specified (is absent), and a greater-than inequality if c is not specified [30]. Since weights in the ILP model are generally out of $\mathbb{R}$ whereas $a_i, b, c \in \mathbb{N}$, rounding is required. We therefore round a_i and b up, and c is rounded down. Consequently, more restrictive constraints result, which can theoretically exclude a solution that otherwise does not violate the original problem constraints. This is the reason that we compare the quality of generated solutions in Section 6.

The general form (11) is used throughout for constraints, with a few exceptions in which constraints can be directly expressed as clauses, and thus directly as ASP rules. This has the advantage of eliminating auxiliary variables and associated constraints such that the overall problem size becomes smaller. An example is (5); the link between the auxiliary variable $\mathcal{M}_{lj}$ for a group of tasks and mapping decision variables x_{ij} is already in conjunctive form so that the constraint can be specified directly as a rule

$$\mathcal{M}_{lj} \longleftarrow x_{i_{l0}j}, x_{i_{l1}j}, \ldots, x_{i_{lg}j}, \tag{12}$$

thereby dropping (7) and (8). However, (12) represents a logical implication, whereas (5) is a logical equality. Without further measures, a stable model can potentially have $\mathcal{M}_{lj}$ as true when one or several of the atoms $x_{i_{lg}j}$ are false. We therefore additionally add the rule

$$\longleftarrow \mathcal{M}_{lj}, 1\left[\text{not } x_{i_{l0}j}, \text{not } x_{i_{l1}j}, \ldots, \text{not } x_{i_{lg}j}\right], \tag{13}$$

as an integrity constraint [30] such that $\mathcal{M}_{lj}$ is not derived if any of its associated atoms is not derived.

Similarly, while not directly obvious, (10) stand for logical conjunctions that can be represented by the rule

$$\gamma_{lj} \longleftarrow \mathcal{M}_{lj}, \text{not } \mathcal{M}_{l_{s_0}j}, \text{not } \mathcal{M}_{l_{s_1}j}, \ldots, \tag{14}$$

where $\mathcal{M}_{l_{s_i}j}$ is the ith superset of the group G_l. The implication is that γ_{lj} is derived when the corresponding group $\mathcal{M}_{lj}$ is derived, but none of the atoms for associated supersets is derived.

Using the same syntax for specifying weights, the objective function has the form [30]

$$\text{minimize } [v_0 = a_0, v_1 = a_1, \ldots, v_n = a_n], \tag{15}$$

but in this case the weights a_i are not directly rounded, rather, the weights which represent costs in time are converted into processor cycles so that (15) matches (1) as close as possible. In order to avoid large numbers which can overflow the computation of the objective function, these weights are expressed in terms of cycles that would have been spent on the slowest processor, normalized by the number of cycles on the same processor that would have been consumed for the smallest weight in the objective.

6. Comparison of Synthesis Results

This section compares synthesis runtime as well as quality of results for ASP-based synthesis against the ILP-based flow. For this purpose, two parallel programs have been implemented using the Message Passing Interface (MPI) standard [32]. Only a subset of the standard that can be efficiently implemented in embedded systems has been used.

Table 1: WLAN tasks.

Function	Index	Deadline	P1	P2	P3
Master	I0	19	6032	1034	11
Phase	I1	988	3504	601	601
Channel	I2	988	909	156	156
Timing	I3	19	33781	5791	17
Demap	I4	1216	598	102	102
FFT	I5	1216	534	534	534
Fine	I6	19	11	11	11
Coarse	I7	19	13	13	13

Table 2: WCDMA tasks.

Function	Index	Deadline	P4	P5
Master	I0	65	4376	43
FFT	I1	4160	3788	3788
Vector mul	I2	4160	3082	3082
FFT	I3	4160	3788	3788
ov. add	I4	4160	3665	3665
Long Code	I5	1040	3784	147
DLL	I6	1040	8746	1000
Short Code	I7	1040	835	143
Rake	I8	1040	16204	686
Combiner	I9	16640	383	383

The first application implements a signal processing chain for IEEE 802.11g WLAN standard between the antenna and the channel decoder. The algorithms are described in [33–36]. The second application implements the functionality for the same portion for WCDMA. The algorithms are described in [37–39]. Tables 1 and 2 show the parallel tasks and their deadlines in nanoseconds. In this implementation, the deadlines are equal to the periods. The latter were obtained from the standards. The three last columns in Table 1 show approximated execution time in nanoseconds of the tasks on three different processors with loosely coupled accelerators. The execution times were obtained by executing the tasks on given processors. Similarly, the two last columns in Table 2 show the approximated time for WCDMA tasks on two other processors.

The differences in execution times in the two tables are caused by different type of accelerators attached to the processors. For example, all cores have a butterfly accelerator based on the architecture described in [40]; therefore, the execution time for FFT tasks is the same for all processors (the difference for FFT tasks between WLAN and WCDMA is due to the different number of FFT points). On the other hand, only P2 and P3 have a CORDIC accelerator; therefore, the execution time for the carrier-phase offset estimation and compensation task is only 156 nanoseconds for these cores, versus 909 nanoseconds for P1.

Tables 3 and 4 show the communication traffic between tasks for the two applications. These were obtained from MPI simulations for 27 OFDM symbols for WLAN, and for one slot for WCDMA. The third and fourth columns show the amount and number of data transfers, respectively.

Table 3: Traffic pattern for WLAN.

Src (i_1)	Dst (i_2)	$D_{i_1 i_2}$ (bytes)	$B_{i_1 i_2}$
I0	I2	220	55
I0	I7	2339608	2242
I0	I1	220	55
I4	I2	5824	28
I4	I1	5824	28
I6	I5	7168	28
I2	I1	5824	28
I7	I6	2913300	2241
I5	I4	7168	28
I1	I3	5824	56

In order to additionally compare the synthesis runtime with and without fixed-priority preemptive scheduling, configurations with a basic cyclic executive and with a preemptive kernel were used for all processors with each run.

For ILP, the same solver settings were used for all cases (node auto-ordering, most feasible basis crash, automatic branch and bound branching, and presolving of rows and columns) [41].

No options were specified for ASP. However, we determined that splitting the objective into its three constituent parts for execution, communication, and scheduling time significantly speeds up the ASP solver time by up to two orders of magnitude. This circumstance was exploited subsequently since the speed up is not accompanied by any penalty in quality of the solution. Splitting the objective function is a feature in clasp/clingo that was conceived to avoid possible overflows when computing the value of the objective function because of integrality of weights [30].

Tables 5 and 6 show the parameters of used processors and networks, respectively. The number of processors and networks used was 6 in each case. Whereas the number of resources could be selected to reflect the number of nodes and edges in the application graph, it is advantageous to start with a smaller number to speedup the synthesis. If resources are exhausted, the number should be increased in another run to avoid preconstraining the design space.

Table 7 summarizes the results, which were obtained on a machine with a T5500 processor and 2 GB of memory. The columns "No. cons." show the number of constraints and the number of rules for ILP and ASP modes, respectively. Similarly, the columns "No. var." show the number of decision variables and atoms for the two modes. These numbers give a measure for the complexity of the problem instances. The number of variables for ILP mode is much less than the number of atoms for ASP mode because the ILP solver, lp_solve, has a presolve option that can reduce the size of the problem by eliminating redundant constraints. This option was exploited because presolving tends to reduce the solver time.

The columns "Form." and "Solver" show synthesis runtime spent formulating and solving the problem, respectively. Formulator time is rather large; most of this time is spent reading text files generated from MPI simulations.

Table 4: Traffic pattern for WCDMA.

Src (i_1)	Dst (i_2)	$D_{i_1 i_2}$ (bytes)	$B_{i_1 i_2}$
I8	I9	40948	2560
I1	I2	81924	161
I0	I1	81412	160
I5	I8	10236	2559
I5	I6	40960	10240
I4	I6	40964	10241
I3	I4	81924	161
I2	I3	81924	161
I6	I8	81892	5119
I7	I8	10236	2559

Table 5: Parameters of used processors.

Name	T_j (μs)	A_j (slices)	No. instances
P1	0.70	450	2
P2	0.93	526	2
P3	1.20	605	2
P4	1.20	553	3
P5	1.19	724	3

Table 6: Parameters of used networks.

Type	M_k	L_k (μs)	p_k	τ_k (μs)	No. instances
Link	2	6.25	0	0	3
Bus	16	8.00	0.0625	16	3

The sizes of the files in these experiments were 4.8 GB and 6.4 GB for WCDMA and WLAN, respectively. They contain, among others, time stamps for each data packet transmitted between tasks. This large time is not a limitation for automated exploration; much faster time can be achieved by using compressed binary files and/or usage of cache files to capture relevant information only during automated explorations.

The solver times for ASP mode are dramatically shorter by up to three orders of magnitude. Given that ASP solver time is in the order of few seconds (versus up to 8 hours for ILP mode), synthesizing using ASP is a promising approach for automatically exploring a large number of design alternatives as it is the requirement for this flexible multiprocessor synthesis problem.

As previously mentioned, we additionally need to compare the quality of results because of a potential for over-constraining in ASP mode, particularly in light of the fact that ASP mode is much faster. The columns "Obj." show the value of the objective function after optimization. Since both methods are heuristic, it is interesting to know how far integer solutions are away from corresponding relaxed solutions, which is a measure of how good a solution is in case the solver times out. A timeout occurred once for WCDMA for ILP synthesis mode under cyclic scheduling. The columns "Gap" give this measure, which indicates that the solution was quite good even in the timeout case.

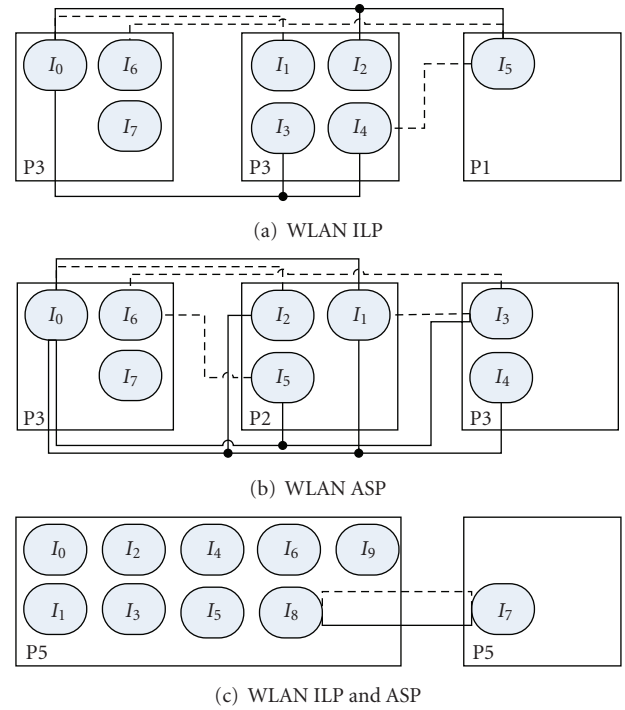


Figure 2: Synthesized architectures under preemptive scheduling. Dashed lines: links, full lines: buses.

Comparing the value of the objectives for the two modes, the differences before rounding are insignificant with the exception of the timeout case where ASP mode found a better solution. The impact of more restrictive constraints for ASP mode due to rounding as discussed in Section 5 was not apparent in these experiments. While these particular results are suggestive, experiments with a much larger set of parallel programs are still required to characterize the potential impact.

The impact of scheduling constraints can be seen by comparing the values of the objective functions under the two scheduling modes. Better values were obtained under cyclic scheduling because no deadlines were imposed when mapping tasks so that only the execution times needed to be considered. Consequently, the solvers attempt to group tasks such that expensive intertask communications are minimized. However, these apparently faster architectures are practically not usable because of no deadline guarantees.

Figure 2 shows synthesized architectures under the two modes for preemptive scheduling. These architectures are very similar for WLAN, and the same architecture was obtained for WCDMA. This result emphasizes the potential for ASP-based synthesis, since the quality of results was not traded against solver runtime. Synthesized architectures are not necessarily unique because there may exist several optimum solutions to a combinatorial problem. Thus, for WLAN case, where the two architectures are similar but not the same, it is quite possible that either architecture could have been obtained through either of ILP or ASP synthesis mode. This is because all resources are allocated

Table 7: Synthesis results. Nonshaded rows for ILP, shaded rows for ASP.

	Cyclic						Preemptive					
	Problem size		Run time (sec)		Obj.	gap	Problem size		Run time (sec)		Obj.	Gap
Appl.	No. cons.	No. var.	form.	solver	(sec)		No. cons.	No. var.	form.	solver	(sec)	
WLAN	7089	3638	497	636	0.0001	0.0	5901	2538	552	1010	0.10	9.6
WCDMA	32261	13320	701	28811	18.25	7.8	25394	12792	704	19567	16.90	0.2
WLAN	7033	207412	1467	2.421	0.0002	—	7033	207412	1477	16.797	0.08	—
WCDMA	25631	21806	977	6.781	16.898	—	25631	21806	1125	10.219	16.90	—

through a non-constraining combinatorial optimization process according to (1) based on resource parameters and characteristics of the parallel program, and not on the synthesis method used.

Finally, two discussion points are in order for these architectures. First, as previously mentioned, our proposed design flow does not pre-constrain the design space. As a result, multiple communication resources are allocated between any pair of processors in this experiment. This allocation minimizes expensive intertask communications which are necessary under preemptive scheduling because of schedulability constraints leading to conditions such that certain tasks cannot be mapped on the same processor. Thus, the generated architecture description (Figure 1) includes not only the netlist, but also information on which communication libraries a task should use to communicate with any other task. This information is used by configuration tools to automatically bind appropriate low-level communication libraries for different networks, links and buses.

Second, the physical implementations of messaging passing interfaces make use of FIFO to queue messages. This means that, once a task has initiated a data transfer by calling the appropriate function, the task is free to do further processing and to initiate or wait for data via another communication resource. The result is that communication latencies can be hidden through overlapping. However, this effect is not accounted for in the objective function because temporal information is not used. Consequently, the actual cost in total computation time can be smaller than what the objective function indicates.

Accounting for temporal information is not a feasible prospect because a far greater number of variables would need to be considered to model and capture all possible moments in which communications can be initiated.

7. Summary and Conclusion

In this paper, a method for automated architecture synthesis for FPGA multiprocessor systems has been presented. The method takes into account fixed-priority preemptive scheduling to cover a broad spectrum of embedded application requirements. Combinatorial optimization is used during synthesis. A case study, in which architectures for IEEE 802.11g and WCDMA baseband signal processing algorithms are synthesized, demonstrates the feasibility of the automated synthesis by showing that problems with sizes that can be encountered in the embedded domain can be solved. Synthesis based on ILP and ASP methods has been compared. ASP mode has a far greater potential for solving difficult synthesis problems. Solver times in this mode were in the order of a few seconds, which is up to three orders of magnitude faster compared to ILP-based synthesis without sacrificing the quality of results.

References

[1] C. Bobda, T. Haller, F. Muehlbauer, D. Rech, and S. Jung, "Design of adaptive multiprocessor on chip systems," in *Proceedings of the 20th Symposium on Integrated Circuits and System Design (SBCCI '07)*, pp. 177–183, Rio de Janeiro, Brazil, September 2007.

[2] W.-T. Zhang, L.-F. Geng, D.-L. Zhang, et al., "Design of heterogeneous MPSoC on FPGA," in *Proceedings of the 7th International Conference on ASIC (ASICON '07)*, pp. 102–105, Guilin, China, October 2007.

[3] M. Saldana, D. Nunes, E. Ramalho, and P. Chow, "Configuration and programming of heterogeneous multiprocessors on a multi-FPGA system using TMD-MPI," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig '06)*, pp. 1–10, San Luis Potosi, Mexico, September 2006.

[4] N. Njoroge, J. Casper, S. Wee, et al., "ATLAS: a chip-multiprocessor with transactional memory support," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '07)*, pp. 3–8, San Jose, Calif, USA, 2007.

[5] T. Kangas, P. Kukkala, H. Orsila, et al., "UML-based multiprocessor SoC design framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[6] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proceedings of Design Automation Conference (DAC '04)*, pp. 113–118, 2004.

[7] H. Ishebabi and C. Bobda, "Automated architecture synthesis for parallel programs on FPGA multiprocessor systems," *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 63–71, 2009.

[8] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[9] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "clasp: a conflict-driven answer set solver," in *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '07)*, vol. 4483 of *Lecture Notes in Computer Science*, pp. 260–265, Tempe, Ariz, USA, May 2007.

[10] E. Giunchiglia, Y. Lierler, and M. Maratea, "Answer set programming based on propositional satisfiability," *Journal of Automated Reasoning*, vol. 36, no. 4, pp. 345–377, 2006.

[11] D. Bertozzi, A. Jalabert, S. Murali, et al., "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 113–129, 2005.

[12] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '05)*, pp. 75–80, Jersey City, NJ, USA, September 2005.

[13] B. H. Meyer and D. E. Thomas, "Rethinking automated synthesis of MPSoC architectures," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–6, March 2007.

[14] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (CODES+ISSS '04)*, pp. 60–65, Stockholm, Sweden, 2004.

[15] 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 System Synthesis (CODES+ISSS '05)*, pp. 273–278, Jersey City, NJ, USA, September 2005.

[16] C. Zhu, Z. Gu, R. P. Dick, and L. Shang, "Reliable multiprocessor system-on-chip synthesis," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pp. 239–244, Salzburg, Austria, 2007.

[17] C. Lee and S. Ha, "Hardware-software cosynthesis of multitask MPSoCs with real-time constraints," in *Proceedings of the 6th International Conference on ASIC (ASICON '05)*, vol. 2, pp. 919–924, Shanghai, China, 2005.

[18] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Hardware-software codesign of multimedia embedded systems: the PeaCE approach," in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, pp. 207–214, Sydney, Australia, August 2006.

[19] D. L. Rhodes and W. Wolf, "Co-synthesis of heterogeneous multiprocessor systems using arbitrated communication," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '99)*, pp. 339–342, San Jose, Calif, USA, November 1999.

[20] P. Pop, *Analysis and synthesis of communication-intensive heterogeneous real-time systems*, Ph.D. thesis, Linkoping University, Linkoping, Sweden, 2003.

[21] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 358–374, 2006.

[22] M. Bao, A. Andrei, P. Eles, and Z. Peng, "Temperature-aware task mapping for energy optimization with dynamic voltage scaling," in *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS '08)*, pp. 1–6, April 2008.

[23] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano, "Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 3–8, Munich, Germany, 2006.

[24] S. Carta, F. Mereu, A. Acquaviva, and G. De Micheli, "MiGra: a task migration algorithm for reducing temperature gradient in multiprocessor systems on chip," in *Proceedings of International Symposium on System-on-Chip Proceedings (SOC '07)*, Tampere, Finland, November 2007.

[25] P. K. Holzenspies, J. L. Hurink, J. Kuper, and G. J. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multiprocessor system-on-chip (MPSOC)," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 212–217, Munich, Germany, March 2008.

[26] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, pp. 475–480, 1995.

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[28] A. Burns, "Preemptive priority based scheduling: an appropriate engineering approach," Tech. Rep. 214, University of York, 1994.

[29] P. Simons, I. Niemelá, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 181–234, 2002.

[30] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "A user's guide to gringo, clasp, clingo, and iclingo," November 2008.

[31] N. Een and N. Sorensson, "Translating pseudo-boolean constraints into sat," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006.

[32] http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[33] T. M. Schmidl and D. C. Cox, "Robust frequency and timing synchronization for OFDM," *IEEE Transactions on Communications*, vol. 45, no. 12, pp. 1613–1621, 1997.

[34] J. Heiskala and J. Terry, *OFDM Wireless LANs: A Theoretical and Practical Guide*, SAMS Publishing, Indianapolis, Ind, USA, 2002.

[35] R. V. Nee and R. Prasad, *OFDM for Wireless Multimedia Communications*, Artech House, Boston, Mass, USA, 2000.

[36] C.-F. Hsu, Y.-H. Huang, and T.-D. Chiueh, "Design of an OFDM receiver for high-speed wireless LAN," in *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 558–561, Sydney, Australia, May 2001.

[37] F. Adachi, M. Sawahashi, and H. Suda, "Wideband DS-CDMA for next-generation mobile communications systems," *IEEE Communications Magazine*, vol. 36, no. 9, pp. 56–69, 1998.

[38] H. Schulze and C. Lüders, *Theory and Application of OFDM and CDMA*, John Wiley & Sons, New York, NY, USA, 2005.

[39] K. Kettunen, "Enhanced maximal ratio combining scheme for RAKE receivers in WCDMA mobile terminals," *Electronics Letters*, vol. 37, no. 8, pp. 522–524, 2001.

[40] O. Atak, A. Atalar, E. Arikan, et al., "Design of application specific processors for the cached FFT algorithm," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '06)*, vol. 3, pp. 1028–1031, Toulouse, France, May 2006.

[41] http://lpsolve.sourceforge.net/5.5.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 541067, 12 pages
doi:10.1155/2009/541067

Research Article

An ILP Formulation for the Task Graph Scheduling Problem Tailored to Bi-Dimensional Reconfigurable Architectures

F. Redaelli,[1] M. D. Santambrogio,[1,2] and S. Ogrenci Memik[3]

[1] *Politecnico di Milano, 20133 Milano, Italy*
[2] *Massachusetts Institute of Technology, Cambridge, MA 02139-4307, USA*
[3] *Northwestern University, Evanston, IL 60208, USA*

Correspondence should be addressed to M. D. Santambrogio, santambr@mit.edu

Received 10 March 2009; Revised 25 June 2009; Accepted 30 September 2009

Recommended by Lionel Torres

This work proposes an exact ILP formulation for the task scheduling problem on a 2D dynamically and partially reconfigurable architecture. Our approach takes physical constraints of the target device that is relevant for reconfiguration into account. Specifically, we consider the limited number of reconfigurators, which are used to reconfigure the device. This work also proposes a reconfiguration-aware heuristic scheduler, which exploits *configuration prefetching, module reuse*, and *antifragmentation* techniques. We experimented with a system employing two reconfigurators. This work also extends the ILP formulation for a HW/SW Codesign scenario. A heuristic scheduler for this extension has been developed too. These systems can be easily implemented using standard FPGAs. Our approach is able to improve the schedule quality by 8.76% on average (22.22% in the best case). Furthermore, our heuristic scheduler obtains the optimal schedule length in 60% of the considered cases. Our extended analysis demonstrated that HW/SW codesign can indeed lead to significantly better results. Our experiments show that by using our proposed HW/SW codesign method, the schedule length of applications can be reduced by a factor of 2 in the best case.

1. Introduction

Systems on a Chip (SoCs) have been evolving in complexity and composition in order to meet increasing performance demands and serve new application domains. Changing user requirements, new protocol and data-coding standards, and demands for support of a variety of different user applications require flexible hardware and software functionality long after the system has been manufactured. Inclusion of hardware reconfigurability addresses this need and allows a deeper exploration of the design space.

Nowadays, reconfigurable hardware systems, FPGAs in particular, are receiving significant attention. At first, they have been employed as a cheap means of prototyping and testing hardware solutions, while nowadays it is not uncommon to even directly *deploy* FPGA-based solutions. In this scenario, that can be termed *Compile Time Reconfiguration* [1], the configuration of the FPGA is loaded at the end of the design phase, and it remains the same throughout the whole application runtime. With the evolution of technology, it became possible to reconfigure the FPGA *between* different stages of its computation, since the induced time overhead could be considered acceptable. This process is called *Run Time Reconfiguration* (RTR) [1]. RTR is exploited by creating what has been termed *virtual hardware* [2, 3] following the concept of *virtual memory* in general computers. When an application bigger than the available FPGA area has to be executed, it can be partitioned in m partitions that fit in that area and these will be executed into numerical order, from 1 to m, to obtain the correct result. This idea is called *time partitioning*, and has been studied extensively in literature (see [4, 5]). A further improvement in FPGA technology allows novel devices to reconfigure only a portion of its own area, leaving the rest unchanged. This can be done using partial reconfiguration bitstreams. The *partial reconfiguration* time depends on the FPGA logic that needs to be changed. When both these features are available, the FPGA is called *partially dynamically reconfigurable*.

However, this scenario turns the conventional embedded design problem into a more complex one, where

the reconfiguration of hardware is an additional explicit dimension in the design of the system. Therefore, in order to harvest the true benefit from a system which employs dynamically reconfigurable hardware, existing approaches pursue the best trade-off between hardware acceleration, communication cost, dynamic reconfiguration overhead, and system flexibility. In these existing approaches the emphasis is placed on identifying computationally intensive tasks, also called kernels, and then maximizing performance by carrying over most of these tasks onto reconfigurable hardware. In this scenario, software mostly takes over the control dominated tasks. The performance model of the reconfigurable hardware is mainly defined by the degree of parallelism available in a given task and the amount of reconfiguration and communication cost that will be incurred. The performance model for software execution is on the other hand static and does not become affected by external factors. Starting from [6], HW/SW codesign researchers try to provide both analysis and synthesis methods specific for new architectures. Classical HW/SW Codesign techniques need to be improved to design reconfigurable architectures, because of a new degree of freedom. This new freedom resides in the design flow: the system can now dynamically modify the functionalities performed on the reconfigurable device. The second aim of this work is to present a model of the problem of scheduling a task graph onto a partially dynamically reconfigurable FPGA, taking into account the possibility of having both software and configurable hardware executions. The novelty of this work resides in the considered architectural model: Figure 1 shows the model. There is a processor and a reconfigurable part, each one with its own memory. The architecture is absolutely general and can be used also for a non-FPGA scenario. Furthermore, in an FPGA scenario, the processor can be within the FPGA or outside the device. What is really effective is that it has to be connected to the reconfigurable part with a channel. The channel is modeled as a bidirectional bus. Once this structure is ensured, the developed model works. With this architecture, when a hardware task needs data from the processor memory there is a latency due to this transfer.

This work provides the following contributions:

(i) an ILP formulation for the problem of minimizing the schedule length of a task graph on a 2D partially dynamically reconfigurable architecture to obtain optimal performance results,

(ii) a heuristic scheduler which takes into consideration *antifragmentation techniques* for general task graphs, a mix between classical deconfiguration policies and *antifragmentation* ones, and the use of out-of-order scheduling to better exploit *module reuse*,

(iii) an ILP formulation and a heuristic scheduler for the extended problems raised by introducing HW/SW Codesign in the initial problem.

This paper will focus on the scheduling of tasks on partially dynamically reconfigurable FPGAs in order to minimize the overall latency of the application. Section 2 proposes a description of the target architecture, describing the architectural solution on which the proposed model has been based. Section 3 describes the proposed ILP formulations and Section 4 the heuristic schedulers. Section 6 presents a set of experimental results comparing the ILP results and the heuristic ones to the results of the model presented in [7]. Finally, the conclusions regarding the proposed approach will be addressed in Section 7.

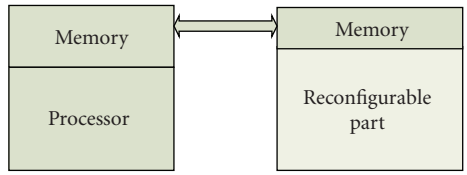
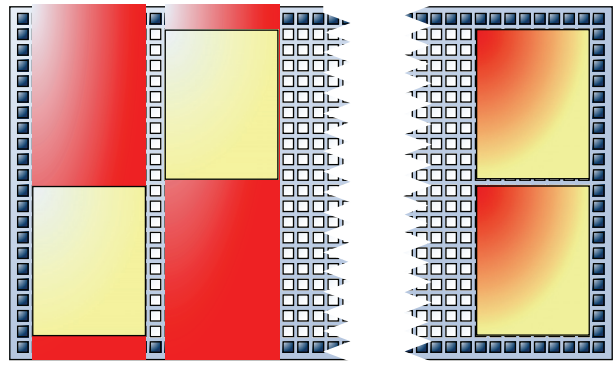


Figure 1: Considered architecture.



(a) (b)

Figure 2: 1D and 2D placement constraints versus 1D and 2D reconfiguration.

2. Target Device and Context Description

2.1. Architecture Description. The modern FPGA devices exploit a technology that allows powerful reconfiguration features. First, it is possible to perform dynamic reconfiguration. Second, emerging technologies allow 2D reconfiguration increasing the designer's degree of freedom. The payback for this increasing freedom is the necessity of new tools capable of exploiting these features in an effective way. The possibility of having a 2D partial dynamic reconfiguration may lead to both better solutions for well-known problems and feasible solutions for new problems.

Figure 2 shows the differences between 1D and 2D reconfigurations. In a 1D scenario, a module occupying only a column portion needs the reconfiguration of the entire column, while the same module in a 2D scenario can be reconfigured in much less area. In the case of Figure 2, in a 1D scenario the two modules would occupy 16 columns, while by exploiting 2D reconfiguration only 8 columns can be used. When a portion of the FPGA has to be reconfigured, a specific file called *bitstream* is needed: this file contains the information concerning the next behavior of that portion of FPGA.

The main characteristic of bitstreams is that they have a correlation with the operation they implement: once the

bitstream is defined, the operation is defined too, while given an operation, there could exist more than one bitstream implementing it. Therefore, it is possible to assign to each bitstream an attribute called *type* used to identify the operation implemented, the area occupied on the target architecture, and the time needed to be configured and to be executed by that bitstream. The latest FPGA technology, such as the Xilinx V4 [8, 9] and V5 [10, 11] families, allows 2D partial dynamic reconfiguration. At the same time, the complexity of the problem of minimizing the schedule length of an application by exploiting reconfiguration increases. Furthermore, thanks to multiple reconfigurator devices, concurrent reconfigurations can be performed, different modules can be configured simultaneously onto the FPGA.

Let us define a set of reconfiguration features that have to be taken into account to define the schedule. *Module reuse* means that two tasks of the same type have the possibility to be executed exactly on the same module on board, with a single configuration at the beginning. The *deconfiguration policy* is a set of rules used to decide when and how to remove a module from the FPGA. *Antifragmentation techniques* avoid the fragmentation of the available space on board trying to maximize the dimension of free connected areas. *Configuration prefetching* means that a module is loaded onto the FPGA as soon as possible in order to hide its reconfiguration time as much as possible.

2.2. Formal Problem Description. The 2D reconfigurable device is modeled as a grid of *reconfigurable units* (RU) by representing rows and columns as two sets $R = \{r_1, r_2, \ldots, r_{|R|}\}$ and $C = \{c_1, c_2, \ldots, c_{|C|}\}$: each cell represented by a pair (r, c), with $r \in R$ and $c \in C$, is made up of ρ_u CLBs. Columns and rows are *linearly ordered*, by which we mean that r_k is adjacent to $r_{k\pm1}$ on the FPGA, for every $1 < k < |R|$; the same property holds also for columns. The application is provided as a task graph $\langle \mathcal{S}, \mathcal{P} \rangle$, which is a Directed Acyclic Graph (DAG). $\mathcal{S}$ is the set of tasks in the graph, while $\mathcal{P}$ is the set of precedences among them. The tasks can be physically implemented on the target device using a set E of *execution units* (EUs), which correspond to different configurations of the resources (RUs) available on the device, therefore different bitstreams. In such a scenario, the reconfigurable scheduling problem amounts to scheduling both the reconfiguration and the execution of each task according to a number of precedence and resource constraints. Resource sharing occurs at EU level: different tasks may exploit the same RUs if they are executed in disjoint time intervals. Moreover, when they also share the same EU, they can be executed consecutively with a single reconfiguration at the beginning. Given any task s and any of its feasible EU implementations i, we assume that suitable algorithms exist to readily compute the latency $l_{i,s}$, the size r_i and the reconfiguration time d_i. Therefore, it is possible to define a function that specifies for each task:

(i) the EU on which it has to be executed,

(ii) the position on the FPGA where to place the selected EU,

(iii) the reconfiguration start time for the selected EU, or the possibility of reuse if possible,

(iv) the execution start time for the task.

In this work the general problem has been simplified: for each task type there is only one available EU. In this way the problem does not lose much in generality, but becomes easier to solve. Since each EU is associated with a bitstream and due to the former simplification, the model works with a number of bitstream types equal to the number of task types.

When HW/SW Codesign is considered, there is the necessity of mapping each task on either the processor or the FPGA. This introduces complexity in the problem solution. In this case the task type concept needs to be extended: each task has both a hardware implementation, that is, a bitstream, and a software one. These two different implementations share the same task type. Moreover, when HW/SW Co-design is condidered, multiple hardware implementations are considered: each task may have more than one bitstream to be implemented on, but the task type remains just one. Another issue, with HW/SW Codesign, is that there is the necessity of moving data between the memory of the processor and the memory of the reconfigurable device. This introduces latency that must be taken into account in the scheduling process. This latency depends on the amount of data needed to be transferred from a task to one of its children.

3. The ILP Formulation for the 2D Reconfiguration and Software Executions

We consider a 2D reconfiguration scenario, as presented in [12]: the sets C and R of RUs are respectively the set of columns and the set of rows of the FPGA. Therefore, all RUs have the same ρ_u (conventionally, $\rho_u = 1$). Each task must be assigned to a rectangular set of RUs and due to the possibility of having multiple reconfigurator devices, concurrent multiple reconfiguration may be exploited. We consider the following model. The starting scenario [12] has been extended to include the possibility of having a task executed also in software, we have to extend the classical *pure* reconfigurable architecture, considering also the presence of the processor, not only to take care of the reconfiguration itself, but also as processing element. Within this scenario, we can work with a processor host in the static area and a reconfigurable area, each one with its own memory. The architecture is absolutely general and can be used also for a non-FPGA scenario. Furthermore, in an FPGA scenario, the processor can be within the FPGA or outside the device. What is really effective is that it has to be connected to the reconfigurable part with a communication channel. Such a communication channel is modeled as a bidirectional bus. Once this structure is ensured, the developed model works. With this architecture, when a hardware task needs data from the processor memory there is a latency due to this transfer. The model considers also multiple hardware implementations for each task, to explore a bigger solution space. Since there are two separated memories, one for the processor and one for the FPGA, it is needed to transfer data

between them when a task requires it. This consideration introduces the concept of communication in the model. In the following are presented only those parts of the model that need to be added to the former one.

3.1. Constants.

(i) $a_{ij} := 1$ if tasks i and $j \in \mathcal{S}$ can be executed on the same bitstreams (by convention, $a_{ii} = 1$), if 0 they use different bitstreams;

(ii) $aM_{ij} := 1$ if task $i \in \mathcal{S}$ can be executed on bitstream $j \in M$, 0 otherwise;

(iii) $l_{ij} :=$ latency of task $i \in \mathcal{S}$ executed on an instance of bitstream j;

(iv) $lp_i :=$ latency of task $i \in \mathcal{S}$ executed on the processor;

(v) $d_i :=$ time needed to reconfigure an instance of bitstream i;

(vi) $c_i :=$ number of RU columns required by an instance of bitstream i,

(vii) $r_i :=$ number of RU rows required by an instance of bitstream i,

(viii) $cdl_{ij} :=$ time needed to transfer the data needed by task j from task i between the FPGA and the processor memories;

(ix) $NREC :=$ number of reconfigurator devices.

The scheduling time horizon $T = \{0, \ldots, |T|\}$ is large enough to reconfigure and execute all tasks. A good estimate of $|T|$ may be obtained via a heuristic.

3.2. Variables. Binary variables:

(i) $m_{ihkm} := 1$ if one instance of bitstream i is present on the FPGA starting from time h until time $h + d_i$ and cell (k, m) are the leftmost and bottommost used by i, 0 otherwise;

(ii) $b_{ihkm} := 1$ if task i is present on the FPGA at time h and cell (k, m) is the leftmost and bottommost used by i, 0 otherwise;

(iii) $p_{ih} := 1$ if task i is present on the processor starting from time h until time $h + lp_i$, 0 otherwise;

(iv) $tm_{ij} := 1$ if task i is executed on one instance of bitstream j, 0 otherwise;

(v) $cd_{ij} := 1$ if task j follows task i and there is the necessity to transfer data through the channel, 0 otherwise;

(vi) $\bar{t}_{ih} := 1$ if the reconfiguration of task i starts at time h, 0 otherwise;

(vii) $m_i := 1$ if task i exploits module reuse, otherwise 0;

(viii) $S_i^{\text{on}} :=$ arrival time of task i on the FPGA;

(ix) $S_i^{\text{off}} :=$ last time instant when task i is on the FPGA;

(x) $t_e :=$ overall execution time for the whole task graph.

3.3. Objective Function. The objective is to minimize the overall completion time of the task graph,

$$\min t_e. \tag{1}$$

3.4. Constraints. We used the *if-then* transformation (see [13]) to model the constraints marked with a*.

3.4.1. Task-to-Bitstream Assignment Constraints. Each task executed onto the FPGA must be executed on a particular bitstream:

$$tm_{ij} \geq m_{jhkm} + b_{i(h+d_j)km} - 1, \quad i \in \mathcal{S},\ j \in M, \quad k \in C, \quad m \in R, \quad h \geq 1 \wedge h \leq T - d_j, \tag{2}$$

$$tm_{jn} - tm_{in} \leq 2 - b_{ihkm} - b_{j(h-1)km}, \quad i, j \in \mathcal{S}, \quad n \in M, \quad k \in C, \quad m \in R, \quad h \in T. \tag{3}$$

When a task is executed in software, it does not need any bitstream, otherwise it needs exactly one bitstream:

$$\sum_{j \in M} tm_{ij} = 1 - \sum_{h \in T} p_{ih}, \quad i \in \mathcal{S}. \tag{4}$$

3.4.2. No-Board Constraints. When a task is executed on the processor, it cannot be placed also on the board:

$$\sum_{h \in T} \sum_{k \in C} \sum_{m \in R} b_{ihkm} \leq T \sum_{j \in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{5}$$

3.4.3. Non-Overlap Constraints. A task cannot be present on the FPGA with different leftmost and bottommost cells:

$$p_{ihkm} + \sum_{l \in C \setminus \{k\}} \sum_{j \in R \setminus \{m\}} b_{inlj} \leq 1, \quad i \in \mathcal{S}, \quad h, n \in T, \quad k \in C, \quad m \in R. \tag{6}$$

3.4.4. Single-Cell Constraints. A task cannot be present on the FPGA with different leftmost and bottommost cells:

$$b_{ihkm} + \sum_{l \in C \setminus \{k\}} \sum_{j \in R \setminus \{m\}} b_{inlj} \leq 1, \quad i \in \mathcal{S}, \quad h, n \in T, \quad k \in C, \quad m \in R. \tag{7}$$

3.4.5. Cell-on-the-Right-and-Top Constraints. The leftmost column of task i cannot be one of the last $c_i - 1$ columns; the same constraint has to be assumed for the last $r_i - 1$ rows:

$$b_{ihkm} = 0 \quad i \in \mathcal{S},\ h \in T,\ k \geq |C| - c_i + 2 \vee m \geq |R| - r_i + 2. \tag{8}$$

3.4.6. Arrival Time Constraints. The arrival time is the time in which a task comes on the FPGA. Since this time is the

first time step in which the associated p variables are set to 1, it must not exceed that time step:*

$$S_i^{\text{on}} \le h \sum_{k\in C}\sum_{m\in R} b_{ihkm} + h \cdot p_{ih} + |T|\left(1 - \sum_{k\in C}\sum_{m\in R} b_{ihkm} - p_{ih}\right), \quad i \in \mathcal{S}, h \in T. \tag{9}$$

3.4.7. Leaving Time Constraints. For each task i, the leaving time must not precede either the last instant for which b is 1 or the time p is 1 plus $lp_i - 1$:

$$S_i^{\text{off}} \ge h \sum_{k\in C}\sum_{m\in R} b_{ihkm} + (h + lp_i - 1)\, p_{ih}, \quad i \in \mathcal{S}, h \in T. \tag{10}$$

3.4.8. No-Preemption Constraints. A task is present on the FPGA in all time steps between the arrival and leaving time: this constraint works thanks to (9), (10), and (7) (No-preemption means that *once the configuration of a task begins* the configured task lasts on the FPGA until the end of its own execution). Equation (7) ensures that all the 1s of a particular task need to be on the same position of the FPGA for all the time that a task exists. This is because a task can perform its work, either be reconfigured or be executed, only if it is on the FPGA, and in specific only when its p variables are set to 1. Equation (9) ensures that the arrival time is lesser or equal to the first, in terms of time, 1 of a task. Equation (10) ensures that the leaving time is greater or equal to the last, in terms of time, 1 of a task. To ensure a task to exist on the FPGA in a single portion of time, the difference between the leaving time and the arrival time needs to be equal to the sum of all the 1s of that task. Since (7) ensures a single position, this constraint ensures that a task cannot be placed and removed and then placed again:

$$S_i^{\text{off}} - S_i^{\text{on}} + 1 = lp_i \sum_{h\in T} p_{ih} + \sum_{h\in T}\sum_{k\in C}\sum_{m\in R} b_{ihkm}, \quad i \in \mathcal{S}. \tag{11}$$

3.4.9. Precedence Constraints. Precedences must be respected:

$$S_i^{\text{off}} - l_j \ge S_i^{\text{off}}, \quad (i,j) \in \mathcal{P}. \tag{12}$$

3.4.10. Task Length Constraints. A task must be present on the FPGA at least for its execution time:

$$\sum_{h\in T}\sum_{k\in C}\sum_{m\in R} b_{ihkm} \ge \sum_{r\in M} (l_{ir} \cdot tm_{ir}), \quad i \in \mathcal{S}. \tag{13}$$

3.4.11. Reconfiguration Start Constraints. Each task has a single reconfiguration start time or none (if it exploits module reuse):

$$\sum_{h\in T} \bar{t}_{ih} = 1 - \sum_{j\in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{14}$$

Reconfiguration starts as soon as the task is on the FPGA, therefore, if a task needs to be configured on the FPGA, its reconfiguration will start at the first time step in which its p variables are set to 1, that is, its arrival time:*

$$-|T| \sum_{j\in M} tm_{ij} \le S_i^{\text{on}} - \sum_{h\in T} h\bar{t}_{ih} \le |T| \sum_{j\in M} tm_{ij}, \quad i \in \mathcal{S}. \tag{15}$$

3.4.12. Reconfiguration Overlap Constraints. At most $NREC$ reconfigurations can take place simultaneously:

$$\sum_{i\in\mathcal{S}} \sum_{m=\max(1,h-d_i+1)}^{h} \bar{t}_{im} \le NREC, \quad h \in T. \tag{16}$$

3.4.13. Starting Time Constraints. This is the general formulation for this constraint, since multiple bitstreams must be considered, as for the nonoverlap constraints. This does not change the constraint formulation itself, but several instances of the constraint have to be created, one for each bitstream. Thus, their number increases, but they are written in the same way, with the obvious variable replacement. The starting instant is reserved, so that the FPGA is initially empty:

$$p_{i0km} = 0, \quad i \in \mathcal{S},\ k \in C,\ m \in R. \tag{17}$$

3.4.14. Single Processor Constraints. Only one processor is available:

$$\sum_{i\in\mathcal{S}} \sum_{l=h-lp_i+1}^{h} p_{il} \le 1, \quad h \in T. \tag{18}$$

3.4.15. Communication constraints. When two tasks i and j are linked by a precedence relation, the results of task i must be transferred to j. Due to the architectural model presented in Section 1, when two tasks are executed on the same device, either both on the FPGA or both on the processor, there is no need to transfer any data because the memories are local to the devices. When one tasks is executed on the FPGA and the other one on the processor, there is the necessity of moving the data from one to the other. For this reason a specific set of constraints have been developed:

$$\begin{aligned} -cd_{ij} &\le \sum_{l\in M} \left(tm_{il} - tm_{jl}\right) \ge cd_{ij}, \quad h(i,j) \in \mathcal{P}, \\ cd_{ij} &\le \sum_{l\in M} \left(tm_{il} + tm_{jl}\right), \quad h(i,j) \in \mathcal{P}, \\ cd_{ij} &\le 2 - \sum_{l\in M} \left(tm_{il} + tm_{jl}\right), \quad h(i,j) \in \mathcal{P} \end{aligned} \tag{19}$$

3.4.16. Definition of the Overall Latency.

$$t_e \ge S_i^{\text{off}}, \quad i \in \mathcal{S}. \tag{20}$$

3.5. Heterogeneous Case. So far, the proposed model describes the problem of scheduling a task graph onto a partially dynamically FPGA with homogeneous columns: all the FPGA cells have the same type. In the latest FPGAs devices it is possible to have columns of different types: CLBs, multiplexer, multiplier, BRAM, and so on. For this reason, a task can be implemented in different ways, due to which columns are involved in the synthesis process. Different implementations for the same tasks are now available, and the design space exploration must be more accurate: a bitstream for each one of these implementations must be created. Using the bitstream concept is very useful, because it is possible to use exactly the same ILP formulation for the extended problem. A preprocessing phase is needed, in order to avoid a bitstream to be placed on not compatible columns, and in general cells.

For each bitstream j, for each time h, for each column k, and for each row m, the variable m_{ihkm} is set to 0 if: given a couple (i,l) such that $i \geq k \wedge i \leq k + c_j - 1$ and $l \geq m \wedge l \leq m + r_j - 1$, cell (i,l) of the FPGA has a different type from cell $(i-k+1, l-m+1)$ of the bitstream. Once all these variables have been set, the proposal ILP can be applied.

4. Napoleon: A Heuristic Approach

From the results obtained through ILP solvers applied over the previous model, see Section 6.1, it is impossible to rely on it because of the huge amount of time needed. It is necessary to develop a fast technique that still obtains good results in terms of schedule length. A greedy heuristic scheduler has been selected as the best choice, and we developed it taking into account the experience achieved by writing the ILP model.

Napoleon is a reconfiguration-aware scheduler for 2D dynamically partially reconfigurable architectures. It is characterized by the exploitation of *configuration prefetching*, *module reuse* and *antifragmentation* techniques. Algorithm 1 shows the pseudocode of Napoleon. First, it performs an infinite-resource scheduling in order to sort the task set $\mathcal{S}$ by increasing ALAP values. Then, it builds subset RN with all tasks having no predecessors. In the following, RN will be updated to include all tasks whose predecessors have all been already scheduled (*available tasks*). SN, instead, is the set of scheduled tasks. As long as the dummy end task S_e is unscheduled, the algorithm performs the following operations. First, it scans the available tasks in increasing ALAP order to determine those which can reuse the modules currently placed on the FPGA. Each time this occurs, task S is placed in the position (k,m) which hosts a compatible module and is the farthest from the center of the FPGA. Unused modules can be present on the FPGA because Napoleon adopts *limited deconfiguration* as an antifragmentation technique: all modules are left on the FPGA until other tasks require their space, in order to increase the probability of reuse. The *farthest placement* criterium is also an antifragmentation technique, that aims at favoring future placements, as it is usually easier to place large modules in the center of the FPGA [14]. The execution

```
t ← 1
𝒮 ← computeALAPandSort(𝒮,𝒫)
RN ←findAvailableTasks(𝒮)
while S_e is unscheduled do
   SN ← ∅
   Reuse ← true
   for all S ∈ RN do
   (k, m) ← findFarthestCompatibleModule(S, t)
      if ∃(k, m) then
         schedule(S, t, k, m,Reuse)
         RN ← RN \ {S}
         SN ← SN ∪ {S}
      end if
   end for
   Reuse ← false
   for all S ∈ RN do
      (k, m) ← findFarthestAvailableSpace(S, t)
      if ∃(k, m) and ∃ free reconfigurators then
         schedule(S, t, k, m, Reuse)
         RN ← RN \ {S}
         SN ← SN ∪ {S}
      end if
   end for
   RN ← RN∪ newAvailableNodes(SN)
   t ← nextControlStep(t)
end while
return t_{S_e} + l_{S_e}
```

Algorithm 1: Algorithm Napoleon($\mathcal{S}$,$\mathcal{P}$).

starting time is tentatively set to the current time step t, but it is postponed if any predecessor has not yet terminated (see Algorithm 2 with Reuse = true). The task is also moved from the available to the just scheduled tasks (subset SN). When no further reuse is possible, Napoleon scans the available tasks in increasing ALAP order to determine those which can be placed on the FPGA in the current time step. The placement is feasible when a sufficient space is currently free or it can be freed by removing an unused module, and when a reconfigurator is available. If this occurs, the position for task S is chosen once again by the *farthest placement* criterium. The reconfiguration starting time is set to the current time step t and the execution starting time is first set to $t + d_S$ and then possibly postponed to guarantee that all the predecessors of S have terminated (see Algorithm 2 with Reuse = false). Thus, there might be an interval between the end of the reconfiguration and the beginning of the execution of a task (*configuration prefetching*). When all possible tasks have been scheduled, the set of available tasks RN is updated: Algorithm 3 does that by scanning the successors of the tasks in SN, which have just been scheduled, and determining the ones which must be added to RN. Finally, the current time step is updated by replacing it with the first time step in which a reconfigurator is available. Algorithm 1 shows the basic scheduling algorithm used, but for the sake of simplicity it does not report two optimizations to increase efficiency: if in the current time step all configured modules are in use, reuse is not possible and the first **for**

```
place(S, k, m)
if Reuse = true then
    t_S ← t
else
    t̄_S ← t
    t_S ← t + d_S
end if
for all S′ ∈ predecessors(S) do
    t_S ← max(t_S, t_S′ + l_S′)
end for
```

Algorithm 2: Procedure schedule($S, t, k, m,$ Reuse).

```
RN′ ← ∅
for all S ∈ SN do
    for all S′ ∈ successors(S) do
        if predecessors(S′) are all scheduled then
            RN ← RN ∪ {S′}
        end if
    end for
end for
return RN′
```

Algorithm 3: Function newAvailableNodes(SN).

loop can be skipped; if there is not enough available area to place any task, because no new placement is possible and the second **for** loop can be skipped.

4.1. HW/SW Extension. The scheduling algorithm schedule the task at the best possible time with respect to the schedules metric. It is simple to add the concept of HW/SW Codesign in this algorithm. Each time a task is considered to be scheduled, the algorithm computes the earliest time it can finish its execution on the processor, considering both precedences and communication delay. Then, if this time is lower than the minimum found on the FPGA device, the algorithm schedules the task on the processor, otherwise on the FPGA.

Figure 3 shows the schedule result obtained by using Napoleon in its HW/SW Codesign version. The characteristics for each task are listed in Table 1.

The number shown underneath the name of the tasks in Figure 3 represents ALAP values. Is it possible to see that Napoleon exploits the processor in an intensive way: it tries to schedule on the FPGA those task types that have more occurrences. This is done accordingly with the ALAP values and the available reconfigurable area: task B and task C share the same type, but their execution on the FPGA is not performed because it will lead to local delay in the schedule.

Table 1: Characteristics of tasks in Figure 3.

	Area	rec. time	HW time	SW time
A	2	2	1	3
B	3	3	1	2
C	3	3	1	2
D	2	2	3	4
E	4	4	2	4
F	2	2	1	3
G	2	2	3	4
H	2	2	1	3

5. Related Works

5.1. Reconfigurable Systems and Codesign Techniques. The VULCAN system [15] has been one of the first frameworks to implement a complete codesign flow. The basic principle of this framework is to start from a design specification based on a hardware description language, HardwareC, and then move some parts of the design into software. Another early approach to the partitioning problem is the COSYMA framework [16]. Unlike most partitioning frameworks, COSYMA starts with all the operations in software, and moves those that do not satisfy performance constraints from the CPU to dedicated hardware. More recent work [17] proposes a partitioning solution using Genetic Algorithms. This approach starts with an all software description of the system in a high level language like C or C++.

Camposano and Brayton [18] have been the first to introduce a new methodology for defining the Hardware (HW) and the Software (SW) sides of a system. They proposed a partitioner driven by the closeness metrics, which provides the designer with a measure on how efficient a solution could be, one that implements two different components on the same side, HW or SW. This technique was further improved with a procedural partitioning [19, 20]. Vahid and Gajski [19] proposed a set of closeness metrics for a functional partitioning at the system level.

In the context of reconfigurable SoCs, most approaches focused on effective utilization of the dynamically reconfigurable hardware resources. Related works in this domain focus on various aspects of partitioning and context scheduling. A system called NIMBLE was proposed for this task [21]. As an alternative to conventional ASICs, a reconfigurable datapath has been used in this system. The partitioning problem for architectures containing reconfigurable devices has different requirements. It demands a two dimensional partitioning strategy, in both spatial and temporal domains, while conventional architectures only involve spatial partitioning. The partitioning engine has to perform temporal partitioning as the FPGA can be reconfigured at various stages of the program execution in order to implement different functionalities. Dick and Jha [22] proposed a real-time scheduler to be embedded into the cosynthesis flow of reconfigurable distributed embedded systems. Noguera and Badia [23] proposed a design framework for dynamically reconfigurable systems, introducing a dynamic context scheduler and hw/sw partitioner. Banerjee et al. [24] introduced a partitioning scheme that is aware of the placement constraints during the context scheduling of the partially reconfigurable datapath of the SoC.

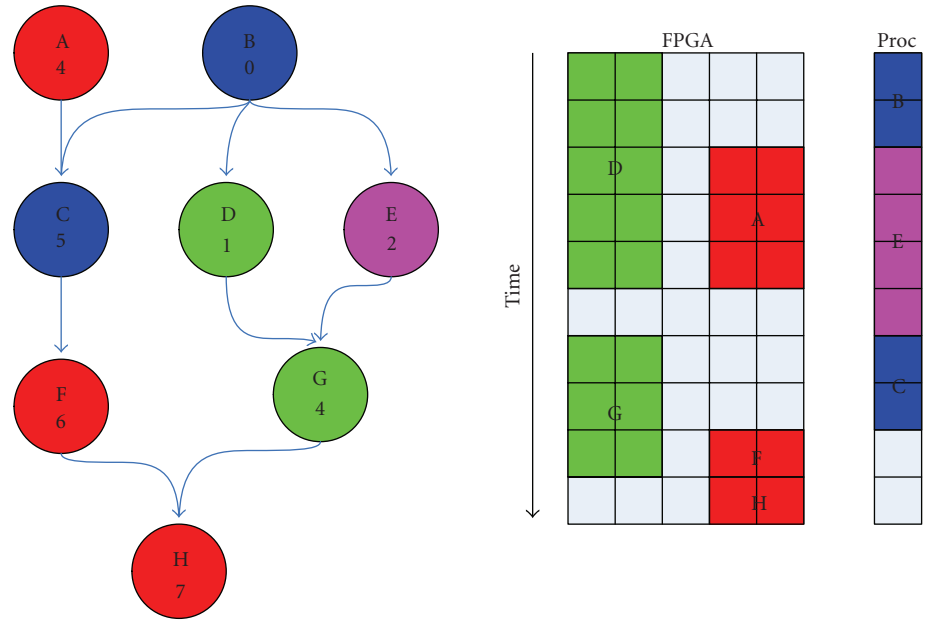


Figure 3: A scheduling example from Napoleon.

5.2. Scheduling Solution. The literature solutions for the considered scheduling problem do not exploit all the possible features of partial dynamic devices where HW/SW partitioning has been taken into consideration. Actual *hardware/software codesign* approaches find a partitioning of the original specification and then schedule the partitioned application on an architecture similar to the one described in Figure 4, where both the processor unit and the FPGA can execute one or more partitions. This kind of approach can be found in [25–28] and the scheduler used is almost always based on *list-based* scheduling algorithms with the priority function given by the mobility range of nodes. All these schedulers are static schedulers: the schedule of the task graph is done only one time before the real execution starts. An existing solution to the problem of partitioning and scheduling a task graph onto an architecture containing a processor and a partially dynamically reconfigurable FPGA [28] is shown in Figure 5.

In this architecture both the processor and the FPGA have their own memory. The FPGA memory is called *shared* just because it can be accessed by all the hardware modules eventually deployed on the FPGA. The authors present an exact approach based on an ILP formulation in order to show how the partial reconfiguration and the tasks placement issue have to be considered in the solution of this problem. This ILP formulation gives as a solution the complete schedule of the task graph and for each task state if it has to be executed in SW or in HW; moreover for the HW task it gives the time in which the reconfiguration has to start and in which is position of the FPGA and the execution starting time. The formulation takes into account dependences between tasks, *configuration prefetching* for the HW tasks, communication overhead between tasks placed in different partitions, and as an improvement also multiple task implementations and

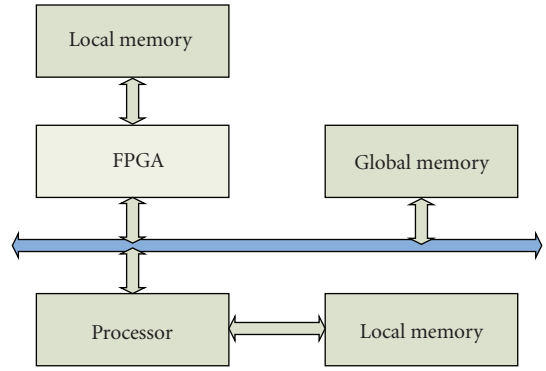


Figure 4: Architecture model for HW/SW Codesigned reconfigurable devices.

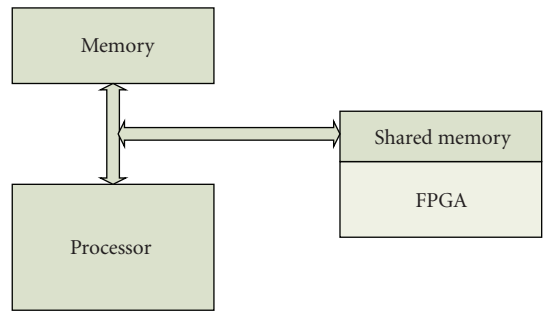


Figure 5: Two separated memories architectural model.

heterogeneity in the conformation of the columns of the FPGA. This is a model for the problem considered in this work, but *module reuse* is never considered.

A refinement of the *module reuse* concept is described in [29], where a solution to the problem of HW/SW Codesign of a task graph onto a partially dynamically reconfigurable architecture is given by an ILP formulation. The formulation is based on two concepts:

(i) early partial reconfiguration (EPR), which is similar to the concept of *configuration prefetching* and simply tries to reconfigure a hardware module as soon as possible onto the FPGA; the aim of this technique is to hide as much as possible the reconfiguration overhead;

(ii) incremental reconfiguration (IR), which is based on the concept of module reuse and states that if a new hardware module has to be placed over another already onto the FPGA, the configuration data that have to be configured are only the percentages that are not in common between the two modules.

The problem with this approach is that, in order to obtain a good IR, it requires the computation of all the possible differences between two task bitstreams and this takes a very long time. Since the developed model is required to be useful to realize a baseline scheduler, the model proposed in [29] is not suitable for the same aim: in an online scenario, all the difference bitstreams need to be in memory, thus, the total memory requirement is very large.

There is a set of works made by Teich et al., [30–32], where the authors present online heuristics for the scheduling/placement policy, taking into account the routing needed by the hardware modules to communicate among themselves. In these works, modules can communicate among themselves without sending data to the processor. This solution is good when tasks have to remain on the device for a long time and they need to frequently send data to other modules, while in a general case when the first issue is to free as soon as possible the reconfigurable area, this approach is not so interesting. Angermeier and Teich [33], present a heuristic scheduler for reconfigurable devices that minimize reconfiguration overheads. The problem with this algorithm is that it works for an architecture where tasks can be placed in a set of identical reconfigurable regions that communicate among themselves through a crossbar. Here the shape of the hardware modules must be well defined and it is impossible to place tasks bigger than a reconfigurable region. Furthermore, the complexity of the problem is reduced, and the scheduler works bad when tasks with great differences in size need to be scheduled on the device.

Some work has been done in the field of processing pipelines, where the whole application is a succession of large tasks that communicate with each other: the first task with the second one, the second with the third, and so on. Specific algorithms have been developed for managing this kind of an application, very important in image processing. In these works, such as [34, 35], the scheduling algorithm handles the HW/SW Codesign in a way that tries to minimize the overall execution time, the communication time among tasks, and to improve the throughput of the system.

6. Experimental Results

6.1. Pure Hardware Reconfiguration: ILP Results. This section compares the optimal results obtained by the models proposed in [7, 36] to the results of the model described in Section 3 considering input specifications characterized by only hardware and reconfigurable hardware elements. The evaluation has been performed by scheduling ten task graphs of ten nodes on an FPGA with 5 columns and 5 rows. The instances considered have small task graphs and few columns and rows because the problem is *NP-complete* and the computation time grows rapidly. Both task graphs and tasks have been generated by hand in order to verify different behaviors of the models: task graphs with tasks of different types, high *module reuse*, and high reconfiguration time. Task occupancy spreads among one column and one row, to four columns and 4 rows. The execution time is of the same order as the reconfiguration time. Furthermore, the number of dependencies goes from linear graphs to almost completely connected ones. The optimal schedule lengths are shown in Table 2. The first and second columns report the results of the proposed model, with 1 or 2 reconfigurators respectively, which correspond to a realistic scenario. The third and fourth columns report the results of the model proposed in [36], once again with 2 or 1 reconfigurators. The former is marked by an* because the original model does not accommodate more than one reconfigurator, and, hence, we have extended it to support multiple reconfigurators. The fifth column reports the results of the model proposed in [7], in which the number of reconfigurators is unlimited but the reconfiguration must immediately precede the execution and follow the end of the preceding tasks. It is possible to see that increasing the number of reconfigurator devices can improve the schedule length. This improvement is not assured because it is not always possible to hide completely the reconfiguration time. The model proposed in [36] is dominated by our proposed approach because it only allows 1D reconfiguration instead of 2D reconfiguration. Dominated means that every solution the model in [36] can find, our approach can find it too. Furthermore, our model can find and explore a bigger design space thanks to the possibility of having 2D reconfiguration. The *Fekete* model, [7], can obtain worse results because it does not exploit *module reuse* and *configuration prefetching* even if it has possibility of reconfiguring as many tasks as it needs at the same time. This is an interesting aspect of our proposed model: by modeling all the physical features recently introduced in reconfigurable devices, better results can be obtained.

6.2. Reconfigurable Hardware and Software Executions: ILP Results. This section compares the results obtained by the proposed HW/SW model, Section 3, and the one proposed in [28]. These same task graphs used in Section 6.1 have been scheduled. The only difference is that, for the HW/SW model, multiple hardware solutions have been taken into account, along with a software solution. The model described in [28] does not consider model reuse, so it is reasonable to expect a worse behavior with high possibility of reuse.

Table 2: ILP results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 [36]*	NRECS = 1 [36]	Fekete [7]
Ten1	15	17	15	17	18
Ten2	22	22	22	22	33
Ten3	16	16	16	16	25
Ten4	14	15	16	17	25
Ten5	21	21	21	21	28
Ten6	19	20	21	22	23
Ten7	20	20	20	20	28
Ten8	22	24	23	24	29
Ten9	26	26	26	26	32
Ten10	23	23	23	23	34

Table 3: ILP results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 [28]*	NRECS = 1 [28]
Ten1	13	13	14	14
Ten2	22	22	24	24
Ten3	14	14	18	18
Ten4	14	14	15	16
Ten5	16	16	17	18
Ten6	16	16	16	16
Ten7	16	16	20	21
Ten8	21	21	21	21
Ten9	22	22	22	22
Ten10	19	19	23	23

The resulting schedule lengths are shown in Table 3. The second and third columns report the results of the proposed model, with 2 or 1 reconfigurator devices respectively, which correspond to a realistic scenario. The third and fourth columns report the results of the model proposed in [28], once again with 2 or 1 reconfigurator devices. The former is marked by an* because the model had to be extended to support multiple reconfigurators. The considered FPGA has two different types of columns and so the tasks used to verify the models. Each one of these tasks can be executed to at least two different bitstreams; furthermore, the reconfiguration model considered is 2D.

It is possible to see that increasing the number of reconfigurator devices does not improve significantly the schedule length. The reason is that multiple branches of a task graph are executed on the FPGA and the other on the processor; to take advantage by using multiple reconfigurator devices, the tasks have to be on an area occupation very little with respect to the FPGA area; moreover, multiple concurrent branches have to be available. The model proposed in [28] is always dominated by the one proposed here because of the impossibility of having *module reuse*. The interesting aspect of the proposed model is that it can exploit in the best possible way the reconfiguration.

Comparing these results with the ones obtained without HW/SW Codesign, see [12], shows that the possibility of having a usable processor increases the schedule effectiveness. The schedule length decreases thanks to the possibility of having more parallelism: it is possible to execute tasks on the FPGA, in parallel, and also on the processor, saving reconfiguration time and FPGA area for subsequent tasks.

6.3. Reconfigurable Hardware and Software Executions: Heuristic Results. In this section we make a comparison between the the HW/SW Codesign model and the correspondent heuristic scheduler. These schedulers have been tested and compared on the following applications (useful to extract features from a large set of data) that have been selected from a popular data mining library, the *NU-MineBench* suite [37]:

(1) *variance* application: it receives as input of a single set of data and calculates the mean and the variance among the whole data set;

(2) *distance* application: it receives as inputs of two sets of data of equal size and calculates the distance between them;

(3) *variance1* application: it receives as input of a single set of data and calculates the mean and the variance among the whole data set. The tasks graph is different than the former variance application, since it involves different task types.

These applications are massive computing applications where there are few task types and a lot of tasks available at the same time. The developed schedulers are effective in this case due to good management of the FPGA area, the *module reuse* and *configuration prefetching* techniques. These applications are characterized by large number of tasks, grouped in two or three task types. Their graphs have the shape of a reverse tree: the same operation, task, must be done over the whole input set, then a new operation over the results, and so on. Each task does not occupy more than 5% of the reconfigurable device and its execution time is really short: two or three clock cycles. Furthermore, the communication time needed by data transfer is comparable with the execution time. The reconfiguration time is two orders of magnitude bigger than the execution time. Regarding the software implemantations, their execution time is one order of magnitude bigger than the hardware execution time. Because of the comparison among heuristics and ILPs, we choose to schedule task graphs with at most 32 tasks.

In this case, increasing the number of reconfigurator devices allows better solution in most of the cases. This is due to the fact that the parallelism can be handled in a more effective way. Napoleon, in its HW/SW Codesign version, reaches the optimal solution in just a case. With respect to the model in [28] and [28]*, the heuristic scheduler obtains always better, or at least not worse, results. This is because during the reconfiguration phases the processor can handle some tasks. In this algorithm, the reconfiguration time for each task is two orders of magnitude bigger than the execution time, thus, the scheduler decides to use the processor for a lot of tasks.

Table 4: ILP/heuristic results comparison.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 Napoleon HW/SW	NRECS = 1 Napoloen HW/SW
Ten1	13	13	13	14
Ten2	22	22	24	24
Ten3	14	14	15	17
Ten4	14	14	15	16
Ten5	16	16	18	18
Ten6	16	16	16	16
Ten7	16	16	19	22
Ten8	21	21	21	21
Ten9	22	22	22	22
Ten10	19	19	21	24
distance	520	520	520	520
variance	520	520	520	520
variance1	610	610	610	610

Table 5: ILP and heuristic execution time.

	NRECS = 2 ILP model	NRECS = 1 ILP model	NRECS = 2 Napoleon	NRECS = 1 Napoleon
Ten1	27 days	26 days	546 ms	477 ms
Ten2	31 days	25 days	413 ms	398 ms
Ten3	30 days	26 days	566 ms	513 ms
Ten4	27 days	25 days	578 ms	544 ms
Ten5	27 days	28 days	456 ms	401 ms
Ten6	34 days	31 days	670 ms	555 ms
Ten7	25 days	25 days	590 ms	487 ms
Ten8	23 days	20 days	602 ms	599 ms
Ten9	39 days	29 days	489 ms	433 ms
Ten10	36 days	37 days	716 ms	673 ms
distance	41 days	40 days	1,222 ms	1,001 ms
variance	35 days	36 days	1,321 ms	1,543 ms
variance1	45 days	41 days	1,561 ms	978 ms

Execution time for the experiments shown in Table 4 is shown in Table 5.

It is possible to see that the ILP solutions are too heavy to be used in real cases, while the heuristic approaches reach good solution in a reasonable time. It is noticeble that the ILP solutions for real applications do not scale bad: this is because the solver exploits standard searching methods that lead to "fast" solutions. Furthermore, due to the reconfiguration time, the processor is used for a lot of tasks at the beginning of the schedule, and this leads to an improvement in solution time.

7. Conclusion

The main goal of this work was to introduce a formal model for the problem of scheduling in a 2D partially dynamically reconfigurable scenario. The proposed model takes into account all the features available in a partial dynamic reconfiguration scenario. The results show that a reconfiguration-aware model can strongly improve the solution. The second goal of this work was to propose a heuristic reconfiguration-aware scheduler that obtains good results, with respect to the optimal one, but in a much shorter time. In fact an ILP solver takes a very long time to solve the problem exactly, while the heuristic algorithm reaches a good solution in a very short time. The results prove that *Napoleon* can be used effectively as a baseline scheduler in an online scenario. The next step in this work is to develop an online scheduler that starting from the results obtained by *Napoleon*, finds a feasible schedule and mapping at runtime.

References

[1] B. L. Hutchings and M. J. Wirthlin, "Implementation approaches for reconfigurable logic applications," in *Field-Programmable Logic and Applications*, W. Moore and W. Luk, Eds., Lecture Notes in Computer Science, pp. 419–428, Springer, Berlin, Germany, 1995.

[2] X.-P. Ling and H. Amano, "Performance evaluation of WASMII: a data driven computer on a virtual hardware," in *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe (PARLE '93)*, vol. 694 of *Lecture Notes in Computer Science*, pp. 610–621, Munich, Germany, June 1993.

[3] W. Fornaciari and V. Piuri, "Virtual FPGAs: some steps behind the physical barriers," in *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*, vol. 1388 of *Lecture Notes in Computer Science*, pp. 7–12, Orlando, Fla, USA, March-April 1998.

[4] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99)*, pp. 616–622, IEEE Computer Society, New Orleans, La, USA, 1999.

[5] J. M. P. Cardoso, "Loop dissevering: a technique for temporally partitioning loops in dynamically reconfigurable computing platforms," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 181.2, IEEE Computer Society, Nice, France, April 2003.

[6] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, 2003.

[7] S. Fekete, E. Koler, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 658–667, Munich, Germany, 2001.

[8] Xilinx, Inc., "Virtex-4 user guide," Tech. Rep. ug70, Xilinx Inc., March 2007, http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[9] Xilinx, Inc., "Virtex-4 configuration user guide," Tech. Rep. ug71, Xilinx Inc., January 2007, http://www.xilinx.com/support/documentation/user_guides/ug071.pdf.

[10] Xilinx, Inc., "Virtex-5 user guide," Tech. Rep. ug190, Xilinx Inc., February 2007, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[11] Xilinx, Inc., "Virtex-5 configuration user guide," Tech. Rep. ug191, Xilinx Inc., February 2007, http://www.xilinx.com/support/documentation/user_guides/ug191.pdf.

[12] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *Proceedings of the International Conference on ReconFigurable Computing and FPGAs (ReConFig '08)*, pp. 97–102, December 2008.

[13] W. L. Winston, *Introduction to Mathematical Programming: Applications and Algorithms*, Duxbury Resource Center, 2003.

[14] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Considering run-time reconfiguration overhead in task graph transformations for dynamically reconfigurable architectures," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 273–274, Napa, Calif, USA, April 2005.

[15] R. K. Gupta and G. D. Micheli, "Hardware/software cosynthesis for digital systems," *IEEE Design & Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[16] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[17] Y. Zou, Z. Zhuang, and H. Chen, "HW-SW partitioning based on genetic algorithm," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '04)*, vol. 1, pp. 628–633, IEEE Press, 2004.

[18] R. Camposano and R. K. Brayton, "Partitioning before logic synthesis," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '87)*, pp. 324–326, 1987.

[19] F. Vahid and D. D. Gajski, "Closeness metrics for system-level functional partitioning," in *Proceedings of the 37th Conference on Design Automation (DAC '95)*, pp. 328–333, IEEE Computer Society, Brighton, UK, 1995.

[20] F. Vahid and D. D. Gajski, "Incremental hardware estimatin during hardware/software functional partitioning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 3, pp. 459–464, 1995.

[21] T. C. Y. Li, E. Darnell, R. E. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th IEEE/ACM Annual Design Automation Conference*, pp. 507–512, Los Angeles, Calif, USA, 2000.

[22] R. P. Dick and N. K. Jha, "CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 62–67, ACM Press, Los Angeles, Calif, USA, 1998.

[23] J. Noguera and R. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[24] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proceedings of the 42nd Annual Conference on Design Automation (DAC '05)*, pp. 335–340, ACM Press, Anaheim, Calif, USA, 2005.

[25] J. Noguera and R. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[26] J. Noguera and R. Badia, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 729–734, Munich, Germany, March 2001.

[27] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the 11th ProRisc Workshop on Circuits, Systems and Signal Processing*, Veldhoven, The Netherlands, November 2000.

[28] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, 2006.

[29] B. Jeong, *Hardware-software partitioning for reconfigurable architectures*, M.S. thesis, School of Electrical Engineering, Seoul National University, Seoul, South Korea, 1999.

[30] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 4, p. 134, April 2004.

[31] S. Fekete, T. Kamphans, N. Schweer, et al., "No-break dynamic defragmentation of reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 113–118, Heidelberg, Germany, September 2008.

[32] S. Fekete, J. C. van der Veen, J. Angermeier, D. Ghringer, M. Majer, and J. Teich, "Scheduling and communication-aware mapping of HW-SW modules for dynamically and partially reconfigurable SoC architectures," in *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS '07)*, p. 9, Zurich, Switzerland, March 2007.

[33] J. Angermeier and J. Teich, "Heuristics for scheduling reconfigurable devices with consideration of reconfiguration overheads," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, Miami, Fla, USA, April 2008.

[34] H. Quinn, M. Leeser, and L. King, "Dynamo: a runtime partitioning system for FPGA-based HW/SW image processing systems," *Journal of Real-Time Image Processing*, vol. 2, no. 4, pp. 179–190, 2007.

[35] H. Quinn, L. King, M. Leeser, and W. Meleis, "Runtime assignment of reconfigurable hardware components for image processing pipelines," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 173–182, April 2003.

[36] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 519–522, March 2008.

[37] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, J. Pisharath, G. Memik, and A. Choudhary, "MineBench: a benchmark suite for data mining workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '06)*, pp. 182–188, October 2006.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 376232, 9 pages
doi:10.1155/2009/376232

Research Article

A Message-Passing Hardware/Software Cosimulation Environment for Reconfigurable Computing Systems

Manuel Saldaña,[1] Emanuel Ramalho,[2] and Paul Chow[2]

[1] *Arches Computing Systems, 708-222 Spadina Avenue, Toronto, ON, Canada M5T 3A2*
[2] *The Edward S. Rogers, Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, ON, Canada M5S 3G4*

Correspondence should be addressed to Manuel Saldaña, ms@archescomputing.com and Paul Chow, pc@eecg.toronto.edu

Received 15 March 2009; Accepted 19 June 2009

Recommended by Lionel Torres

High-performance reconfigurable computers (HPRCs) provide a mix of standard processors and FPGAs to collectively accelerate applications. This introduces new design challenges, such as the need for portable programming models across HPRCs and system-level verification tools. To address the need for cosimulating a complete heterogeneous application using both software and hardware in an HPRC, we have created a tool called the Message-passing Simulation Framework (MSF). We have used it to simulate and develop an interface enabling an MPI-based approach to exchange data between X86 processors and hardware engines inside FPGAs. The MSF can also be used as an application development tool that enables multiple FPGAs in simulation to exchange messages amongst themselves and with X86 processors. As an example, we simulate a LINPACK benchmark hardware core using an Intel-FSB-Xilinx-FPGA platform to quickly prototype the hardware, to test the communications. and to verify the benchmark results.

1. Introduction

High-performance reconfigurable computers (HPRCs) are now at a similar stage as supercomputers were before the appearance of MPI [1]. Prior to MPI, every vendor had their own Message passing application program interface (API) to program their own supercomputers causing a lack of portable designs. Currently there is no standard API for the interaction between processors and FPGAs. Companies such as Cray [2], SGI [3], Intel [4], XtremeData [5], DRC [6], and SRC [7] provide their own software APIs and their own hardware interfaces for application hardware engines. This situation reduces the portability and productivity because vendor-specific details distract designers from focusing on the application algorithm. We believe that in the same way a standard C program runs in any X86 processor with most operating systems; a standard VHDL/Verilog design can be implemented on any FPGA, with the exceptions of using nonstandard code, for example, specific resources on a given chip or non-ANSI C functions.

In addition to the portability issue, the mix of X86 processors (X86 from now on) and FPGAs introduces new design challenges that require new design tools. For example, testing and debugging procedures for software are different from the procedure used in hardware. A typical testing procedure in software is a step-by-step execution or printing debug information to the screen. In contrast, for hardware components, such as FPGAs, a detailed behavioral or even timing simulation is required. Both software and hardware can be tested independently up to a certain extent but system-level features or dynamic interaction between them is harder to test that way. For example, a bus functional model (BFM) helps with verifying and developing low-level interactions with a given communication interface, but higher-level protocols or application-level protocols cannot be tested, especially if the behavior changes based on the data received or sent.

A multi-FPGA multicore system with heterogeneous computing elements running in parallel requires system-level tests in addition to tests to the individual components in isolation.

In this paper we extend previous work on TMD-MPI [8, 9], which implements a subset of the MPI standard targeting multiple computing elements (hardware engines and embedded processors) inside FPGAs to include X86 processors enabling a uniform and portable MPI-based communication mechanism for HPRCs. To do this, we developed the Message-passing Simulation Framework (MSF) that allows multiple X86 processes, running at full speed, to exchange messages with computing elements inside the FPGAs being simulated. With this approach we exercise the system-level interaction while having full visibility of what happens inside the FPGAs.

In this paper, we perform a functional system-level simulation of the LINPACK benchmark [10] with the purpose of testing the communications, the simulation environment, and quickly prototyping and verifying the correctness of the LINPACK hardware engine, which is in the early stages of development.

The rest of the paper is organized as follows. Section 2 provides a quick overview of previous work on TMD-MPI. Section 3 contrasts our work to other related cosimulation environments. Section 4 presents the communication infrastructure and its simulation framework. Section 5 explains how TMD-MPI and the MSF can help with system-level architecture exploration. Section 6 describes an example simulation of the LINPACK benchmark system using the MSF. Section 7 presents future work. Finally, conclusions are discussed in Section 8. At the end of the paper there is a glossary of all the acronyms used in this paper.

2. Background

As mentioned before, we use TMD-MPI to provide an abstraction layer for the communications. TMD-MPI has been developed as a result of the need for a programming model for the Toronto Molecular Dynamics (TMDs) machine being developed at the University of Toronto [11]. The TMD machine is a scalable Multi-FPGA configurable system designed to accelerate Molecular Dynamic simulations, although the machine is not limited to this particular application. In fact, the generic MPI-based programming model and the flexibility of FPGAs allow us to target a broader spectrum of computing-intensive applications; however, the TMD-MPI name still remains for historical reasons.

Previously, TMD-MPI only supported MPI-based communication between PowerPC embedded processors, MicroBlaze soft-processors, and hardware engines (collectively known as Computing Elements) across multiple FPGAs, but now with HPRC featuring tightly coupled FPGAs to X86 processors, we have extended TMD-MPI to include X86 processors using shared memory as a medium to exchange messages.

TMD-MPI does not include all the MPI functionality described in the standard because it is targeted to run on FPGAs with limited resources, such as memory (e.g., 4 MB of on-chip RAM in Virtex5 chips compared to GigaBytes for a typical X86 system). Nevertheless, functionality can be added as needed depending on the application. TMD-MPI supports blocking and nonblocking communications as well as some collective operations, which is enough to implement many parallel applications. With the appearance of HPRC machines, a new window of opportunity arises to have a more complete implementation of the standard. For example, including MPI-2 functionality such as remote memory access to implement DMA capabilities to exchange data between X86 processors and FPGAs.

The TMD-MPI programming model is based on the assumption that, from the communications perspective, Computing Elements inside FPGAs can be treated as peers rather than just coprocessing units, which is the way a typical MPI program works. Also, modern FPGAs have enough resources to host several Computing Elements interconnected using an on-chip network, which TMD-MPI also abstracts from the user. A system-level approach at the beginning of the design flow can help to conceive hardware accelerators as peers to processors rather than mere coprocessors.

In an FPGA-as-coprocessor model, an X86 usually acts as a message relay between Computing Elements located in different FPGAs introducing big latencies and limiting what FPGAs can do in terms of communication. For example, in a typical Master-Slave parallel program, FPGA coprocessors attached to the software slave processes would actually be slaves of the slave processes. In contrast, with a peer-to-peer model, Computing Elements (including X86 processors) can exchange data between themselves regardless of their physical location and without intermediaries, which reduces the latency and also simplifies the programming model.

We have used TMD-MPI in multi-FPGA machines based on Amirix [12] and BEE2 [13] boards to implement Molecular Dynamics. Currently we are porting the application to use an HPRC with Intel processors and Xilinx FPGAs attached to the FSB; it is for the latter case that we created the MSF to help us develop and verify our designs.

3. Related Work

There has been abundant research on codesign methodologies and cosimulation environments [14]. The research concludes that the lack of a system-level view of a mixed HW/SW system leads to difficulties in verifying the entire system, and hence to incompatibilities across the HW/SW boundary leading to inefficient designs. However, most of the research focuses on embedded systems with microcontrollers, DSPs, ASICs, and FPGAs, but the research does not address explicitly the High-performance Supercomputing sector. The appearance of FPGAs in Supercomputers opens opportunities to adapt and apply codesign techniques and cosimulation environments to HPRCs. Our TMD-MPI and MSF are one step towards that direction by framing cosimulation into an MPI-based paradigm.

Most of the cosimulation environments typically use hardware in the form of accelerators to speedup the simulation itself. For example, in [15], the authors provide a cosimulation environment where an X86 and an FPGA

are placed on a dual socket motherboard to accelerate a processor simulation tool called Simplescalar. In [16], the authors use an FPGA plugged into the PCI bus to accelerate ModelSim's [17] functional simulations. In contrast, we do not use the FPGA to accelerate a simulation. We use ModelSim running in an X86 to simulate and emulate the FPGA, and let it interact with other X86 processors as if the FPGAs were present. Once the design inside the FPGA has been verified in simulation it can run at full speed in the real FPGA.

Other vendor-specific simulation frameworks such as Cray's simulation framework [18] and SGI's SSP Stub [19] only allow a Bus Functional Model (BFM) testing procedure or low-level data transfer primitives. The user can provide a set of inputs to the FPGA with certain delays and expected outputs to compare the results against. This static kind of verification is adequate to test the interaction with a given interface or for independent FPGA testing, but not as a system-level multi-FPGA approach. Our simulation approach is more generic and portable, allowing the simulation of multiple FPGAs, each with multiple hardware engines (possibly heterogeneous) interacting with multiple X86 MPI software processes concurrently.

4. Simulation Environment

In this section we describe our HPRC reference architecture and the MPI-based communication system. Then we explain how the MSF enables the simulation of such architectures.

4.1. Reference Architecture. Figure 1 shows our reference architecture, which is based on an Intel 4-Processor Server System S7000FC4UR motherboard and the Xilinx ACP M2 FPGA modules distributed by Nallatech [20]. These modules can be stacked one on top of another (M2 Stack) to group a number of FPGAs and plug them into the processor socket on the motherboard, providing higher compute density. Any combination between three Intel Xeon quad-core processors and one M2 Stack, or three M2 Stacks and one Intel Xeon quad-core processor is permitted. In all cases they share the main system memory through the FSB North Bridge chip.

The M2-Stack consists of two kinds of M2 modules: the M2 Base (M2B) and the M2 Compute (M2C). The M2B module has one XC5VLX110 FPGA, which contains the Xilinx FSB interface to provide access to the main system memory. The M2C module has two XC5VLX330 FPGAs and it plugs on top of the M2B. An additional M2C can be stacked on top of the first M2C. The FPGAs in this three-layer stack are connected through parallel LVDS lines.

Currently, the system runs a 64-bit CentOS Linux SMP operating system with 8 GB of memory.

4.2. Message Passing on the ACP Platform. Figure 2 shows an example of a parallel MPI application mapped to our reference platform. For simplicity, in this case, one Quad-core Xeon processor and one M2B module are used. The application has a total of six tasks known as ranks in the MPI jargon. Each rank in the system (logically represented as ovals in Figure 2) has its own private memory space (on-chip memory). Three of the ranks (R0, R1, and R2) are software processes and run in three X86 cores inside the Quad-core Intel Xeon processor. The remaining three ranks (R3, R4, and R5) are inside the FPGA running as hardware engines (hardware ranks), although they could be Microblaze soft-processors or embedded PowerPC processors as well. The X86 processors exchange messages using shared memory, and the hardware engines exchange messages using the Network-on-Chip (NoC). To exchange messages between X86s and hardware engines the data must travel through a shared memory MPI bridge (*MPI_Bridge*), which implements in hardware the same shared memory protocol that the X86 processors use. This bridge takes data to/from the NoC and issues read or write memory requests to the vendor-specific low-level communications core (LLCC), which executes the request. The *MPI_Bridge* effectively abstracts the vendor-specific communication details from the rest of the on-chip network.

For this paper, we used Intel's FSB Bus as the communication media but the same concepts can be applied to other communication media, such as AMD's HyperTransport [21], Intel's QuickPath [22], Cray's Rapid Array Transport [18], SGI's Scalable System Port-NUMA link connection [19],

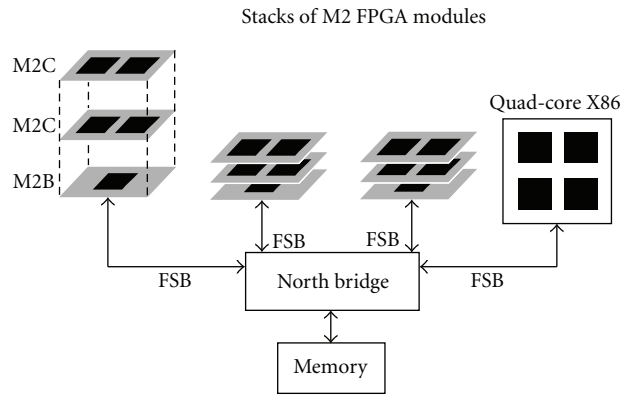


Figure 1: Stacks of Xilinx M2 FPGA modules that can be placed in standard CPU sockets on the motherboard.

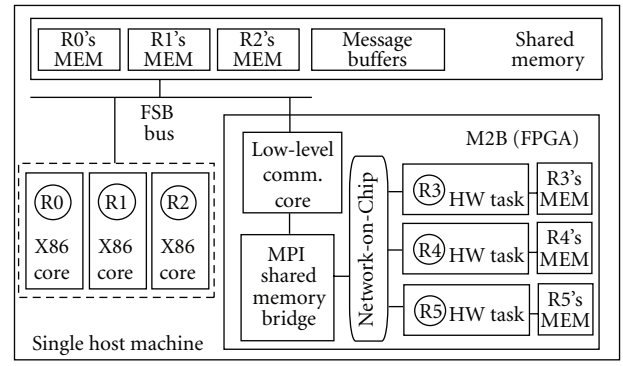


Figure 2: Example of a 6-rank MPI application, with three ranks mapped to X86 processor cores and three ranks mapped to hardware engines.

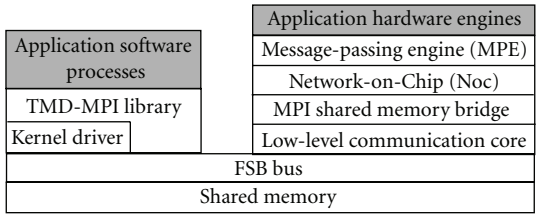


Figure 3: HW/SW abstraction layers.

or even with a standard PCI Express core because they all provide a physical connection to the main system memory. The communication media determines what LLCC to use. In this paper, we use a Xilinx-Intel FSB communication core that handles the low-level protocol to read and write to memory as well as the memory coherence control.

TMD-MPI's shared-memory, message-passing protocol should be mostly the same across HPRCs. The only change is the physical interconnection between the *MPI_Bridge* and the vendor-specific LLCC. By implementing an *MPI_Bridge* for each type of LLCC we make the system portable. For example, in this paper we use an MPI_Xilinx_FSB_Bridge, but we could also implement an MPI_Cray_Bridge to use a Cray HPRC machine.

An extension of this approach to a distributed memory machine (a Cluster) or many HPRC hosts is natural since message-passing assumes no shared memory. A distributed memory approach could use an *MPI_Ethernet_Bridge*, or any other point-to-point communication interface to allow the connection of multiple hosts through the FPGA itself; however, this remains future work for now and in this paper we focus only on a single host machine.

In Figure 2 only one FPGA is shown, but multi-FPGA systems can also be part of the MPI communication by plugging two more M2B modules or stacking M2C modules. This is further explained in Section 4.5.

4.3. Abstraction Layers. Figure 3 shows the abstraction layers for software and hardware in the TMD-MPI programming model. A software application relies on the MPI library layer to send and receive data (calls to MPI_Send(), MPI_Recv(), etc.). In turn, TMD-MPI uses a kernel driver to allocate memory for the shared memory buffers, to perform virtual-to-physical memory translations and some low-level setup for the LLCC in the FPGA. Data is then placed in memory via the FSB and the MPI shared memory bridge will read it and send it over the NoC, which will route the packets to the proper destination Message Passing Engine (MPE). Finally, the MPE will deliver the message to the application hardware engine. Data traveling in the opposite direction is also possible; the FPGA can be a master and send data without the X86 first having to request it.

The MPE encapsulates part of the MPI functionality in hardware. It is responsible for handling requests, acknowledgments, and full-duplex data transmission and reception. Also, it is in charge of packetizing/depacketizing large messages as well as handling unexpected messages. A hardware engine interacts with its MPE via FSLs, which are Xilinx unidirectional FIFOS. An example of the interface between a hardware engine and the MPE is further discussed in Section 6.

With this communications architecture, applications become more portable because the hardware accelerators should remain unchanged from one HPRC to another as well as the software code for the X86 processors since they are all behind the message-passing abstraction.

4.4. The MSF FLI Module. By using TMD-MPI, hardware engines and software processors are isolated from machine-specific communications hardware. However, it introduces a new challenge for the design and verification of applications. In a typical MPI parallel program, an MPI rank is not tested in isolation from the other MPI ranks. It has to be tested with all ranks running at once to verify the correct collective operation and synchronization between them. With FPGAs as containers of MPI ranks, they must be part of the system-level testing process. As mentioned before, FPGA testing requires a cycle-accurate simulation, so the question now becomes how to simulate such a system. Furthermore, the complexity of the testing process increases if there are multiple FPGAs to simulate, each with potentially different hardware engines or embedded processors.

The MSF provides a portable simulation environment based on ModelSim that emulates the FPGA and lets the MPI ranks inside of the FPGAs exchange messages with the ranks running in X86 processors or in other FPGAs. Figure 4 shows the simulation scheme of the architecture depicted in Figure 2. Note that the FPGA in Figure 2 is now an X86 core running ModelSim simulating the FPGA design. For ranks R0, R1, and R2 running in the X86 processors, the FPGA in simulation will be seen as a slow FPGA (simulation speed). In this sense, the FPGA simulation is actually an emulation of the FPGA. Naturally, the time it takes to send a message will be drastically reduced when the FPGA is no longer in simulation and runs in the real FPGA. However, keep in mind that a message-passing paradigm assumes a coarse grain parallelism in which tasks should have reasonable communication demands to be efficient and also should be latency tolerant. In other words, a correct MPI program does not rely on the time it takes to send or receive a message to produce the correct results, and therefore the latency introduced by the simulation should not change the results when the FPGA design runs in the actual physical FPGA.

The central part of the MSF is the use of ModelSim's Foreign Language Interface (FLI) [23], which is a typical way to perform cosimulations by allowing a C program (actually a shared library) to have access to ModelSim's simulation information, such as signal or register values, components instantiated, and simulation control parameters. The MSF FLI module replaces the vendor-specific LLCC by providing the required functionality directly to the MPI_Bridge. The MSF FLI accepts the MPI_bridge memory requests (address and data) and performs the reads and writes directly to shared memory. In the case of the distributed memory environment, the MSF FLI module would translate

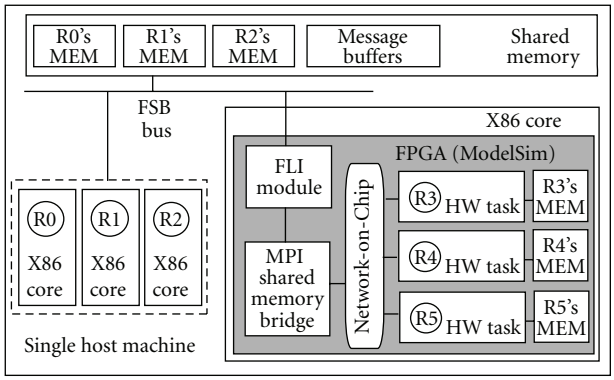


Figure 4: Simulation scheme for the sample application. One X86 core runs ModelSim with the FLI module for shared memory.

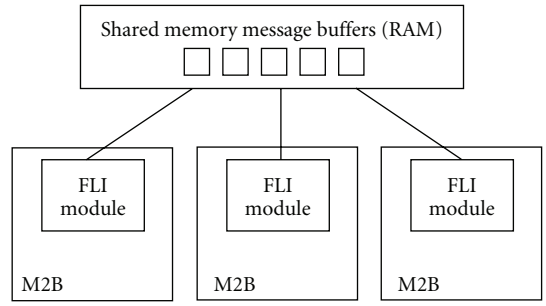


Figure 5: Multiple FSB FLI Modules can independently access the message buffers in shared memory during simulation.

the send/receive requests to socket writes/reads allowing the interaction of remote machines with the FPGA under simulation.

The MSF FLI module uses TMD-MPI's memory allocation and memory mapping subroutines to be able to access the shared memory message buffers. In a typical transaction, the MSF FLI module receives the physical addresses of a buffer from the MPI_Bridge and translates them to virtual addresses before reading or writing to main memory. This is required because the MSF FLI module is under the control of the operating system as a normal user process, which cannot access memory using a physical address.

Since there can be a variety of MPI_Bridges based on the vendor-specific LLCC, there will be a corresponding FLI module that ModelSim can load at runtime. That is, there will be MSF FLI module variations. For example, we use the *FLI_Xilinx_FSB* module, but we could implement the *FLI_Cray* to simulate the interaction with the FPGA in a Cray machine. This is convenient because the simulation itself becomes portable. TMD-MPI and the MSF FLI module absorb the platform changes and make the simulation in different HPRCs transparent to the user.

An additional advantage of the MSF is that there is no need to simulate the vendor-specific LLCC, which can be proprietary and not public, such as the Intel FSB signals. The MSF does not need to know the details of those vendor-specific internals because the FLI module provides the MPI_Bridge with the same memory access (or network device access for the distributed version) that the LLCC provides. In other words, the MSF FLI module models the functional behaviour of the LLCC in terms of memory access.

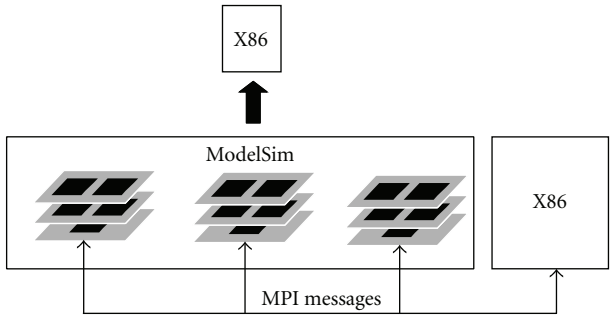
During the simulation, the user has full visibility inside the FPGA at the resolution available in ModelSim, which is useful when tracking bugs in the design, such as glitches, signal delays, or any other subcycle events with the caveat of reduced simulation speed. Black-box cosimulation environments can be faster than ModelSim but limit the design's visibility to its outputs, and only use cycle-accurate simulations, whereas Modlsim can simulate subcycle delays. Also, in the MSF, the user has full control of the simulation by using ModelSim's console or GUI to stop it, pause it, and continue it. Even breakpoints can be asserted to stop executing a particular hardware MPI rank and the software MPI ranks can continue executing because they are completely decoupled due to the implicit asynchronous nature of the message-passing programming model. Only those MPI ranks that are exchanging messages with a stopped rank will be automatically blocked if using MPI blocking calls or if there is a collective operation, such as an MPI_Barrier and MPI_Bcast. However, if nonblocking communications are used, then the ranks can overlap computation and communication.

4.5. Multistack Simulation. At this point, only one M2B plus one Xeon processor system has been presented to explain the basic concept of TMD-MPI and the MSF. However, a fully populated system with three M2-stacks (1 M2B and 2 M2C per stack) would contain 15 FPGAs and a quad-core Xeon processor per motherboard. To perform system-level tests, it should be possible to simulate all the FPGAs at once while communicating with MPI software processes running in the X86.

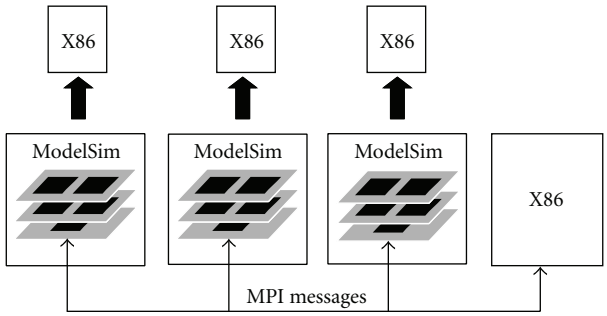
Such a multi-FPGA, multistack simulation imposes challenges on the CAD tools to generate simulation models and set up testbenches. In addition, an interstack communication mechanism for simulation is required, and it poses an increased demand on computing power to simulate such large systems.

To address the multistack communication mechanism in simulation, the shared memory protocol in the MPI_FSB_Bridge and the TMD-MPI software library were modified to support this feature. However, the MSF FLI module only required minor modifications because it is only an intermediary for performing memory reads and writes as commanded by TMD-MPI's shared-memory message-passing protocol. From the MSF perspective, interstack communication was a straightforward extension of the one-stack MSF version. Now, as shown in Figure 5, multiple FLI modules (multiple M2Bs in simulation) exchange information through shared memory without Xeon processor intervention, or having to set up or start the communication.

To generate the simulation models, the MSF uses the Xilinx *simgen* command included as part of the ISE/EDK design tools. *Simgen* generates a self-contained simulation model of an EDK project, which represents an entire FPGA



(a) One single ModelSim process is executed by a single CPU.



(b) Multiple ModelSim processes take advantage of multicore processors.

Figure 6: Two options to simulate multiple M2 Module Stacks.

design. This includes simulation models for the components instantiated inside the FPGA. The MSF then runs *simgen* for as many FPGAs as there are in the system, and finally it creates a ModelSim script to generate the clock and reset signals.

Once the simulation models for all the FPGAs are generated there are two options on how to set up the simulation using ModelSim, and this is shown in Figure 6. One option is to create a top-level testbench that instantiates the FPGAs as components in the testbench and simulates all the FPGAs in one ModelSim process. The second option is to start three ModelSim processes, one per stack, and let them simulate independently. In either case, the FLI module will communicate the same way (through shared memory) and every M2B will have its own FLI module. Also, the amount of computation required to perform the simulation is the same in both options because it is the same logic. However, the second option should be able to take advantage of the quad-core processor. By having three ModelSim processes, each process can execute concurrently in their own X86 core, leaving the fourth core for the X86 MPI software application.

To the best of our knowledge, ModelSim is not a parallel application nor takes advantage of multicore processors. But by virtue of using an implicit parallel programming model (MPI) to program FPGAs we can also split and run the simulation in parallel without ModelSim being a parallel program itself and without modifying its code.

To prove this hypothesis we simulate a simple test that consists of two hardware engines exchanging ping-pong (round trip) messages between two M2B modules, one hardware engine per module. One experiment will have both M2B modules in one ModelSim process and the second experiment will use two ModelSim processes, one per M2B. The MPI application contains three ranks, rank 0 (X86 processor) will send the test configuration parameters as messages as well as the "start test" message to ranks 1 and 2, which are the hardware engines. Then ranks 2 and 3 will exchange round trip messages of increasing size and when the test is complete they will inform rank 0 by sending a "done message." The actual messages will pass through the FLI modules and shared memory because ranks 1 and 2 are located in different stacks, and it is the same amount of logic to simulate.

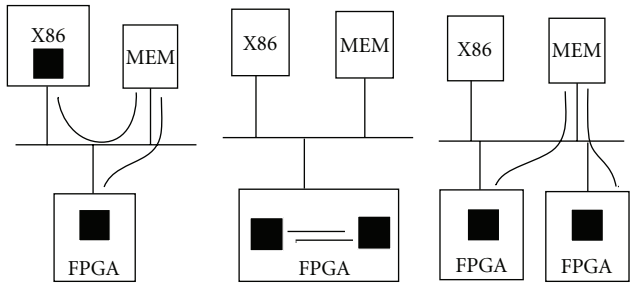
We measure the simulation time by using the MPI_Wtime() function, which returns the number of seconds elapsed since the application started to run, and obtain the time difference between the "start" and "stop" messages. The time measurement reports 20 seconds for the first option (both FPGAs in one ModelSim process) and 10 seconds for the concurrent simulation. As expected, the concurrent simulation is twofold faster than the single process option. We anticipate that for a three-stack system, the parallel simulation option would be three times faster than the all-in-one simulation option.

5. Flexibility and Architecture Exploration

One of the biggest advantages of abstracting and standardizing the communications using MPI is the flexibility to place the computing ranks in different places and using different types of computing elements. Ranks can initially be a processor (X86, PowerPC, MicroBlaze, etc.) or hardware engines, and the designer can change forth and back the type of the rank without having to change the code for the rest of the ranks. In other words, a rank can be a software process and then be replaced by a hardware engine with the same functionality.

Similarly, if a rank is implemented in an FPGA, the rank can be physically placed in one FPGA and later it can be moved to another FPGA without changing the code of any of the other ranks or the code of the rank that is being moved. From the programmer and designer perspective, the communication is being performed between two or more ranks, regardless of their location. The on-chip network will route the packets and make sure the messages arrive at their destinations.

Figure 7 shows the ping-pong example from Section 4.5 but the ranks are implemented in different ways. In Figure 7(a) the X86 processor is a software rank and it exchanges messages with a hardware rank. The ping-pong messages are exchanged through shared memory. In Figure 7(b) the X86 software rank is replaced by a hardware rank resulting in two hardware ranks implemented in the same FPGA. In this case the ping-pong messages are exchanged using only the on-chip network; there is no shared memory access. Finally, in Figure 7(c) one of the hardware ranks is moved to another FPGA and now the ping-pong messages are exchanged using shared memory. Note that in



(a) X86-M2B (b) Intra-M2B (c) Inter-M2B

Figure 7: Different ways two MPI ranks can be mapped and communicate.

this case the X86 processor is independent and does not intervene in the ping-pong messages between the hardware ranks. In all three scenarios the C and HDL code is the same, changing only configuration files that indicate the mapping of computing ranks to physical resources available.

The Message-Passing Simulation Framework allows the designer to simulate the three options in Figure 7 and perform this architectural exploration before running a potentially lengthy place and route processes for all the different combinations.

6. Case Example: LINPACK

This section presents a brief description of our LINPACK benchmark parallel implementation and some insights of the core and its communication interface. Keep in mind that we use this application to test the MPI_Bridge, the TMD-MPI software library, and the MSF FLI module, rather than to obtain peak performance in LINPACK.

6.1. LINPACK Implementation. The LINPACK Benchmark [10] is a widely used algorithm that measures floating-point computing performance by solving a system of linear equations, $\mathbf{Ax} = \mathbf{b}$. It has two main subroutines: *DGEFA* (performs an LU decomposition on the matrix **A**) and *DGESL* (solves the system of linear equations by using vector **b**). More than 97% of the time taken to compute the benchmark is spent inside the *DGEFA* subroutine. *DGEFA* comprises three BLAS [24] level 1 functions, *IDAMAX*, *DSCALI*, and *DAXPY*, where the latter, alone, is responsible for about 95% of the time spent in the *DGEFA* subroutine. The original benchmark uses double precision, but for simplicity we use single precision, which is acceptable for the purposes of this paper.

To implement a parallel version of this algorithm, we first parallelized the sequential LINPACK code using MPI with all the ranks running in X86 processors, and verify the correctness of the parallel algorithm itself purely in software. At this point, high-level application decisions can be made, such as the communication pattern or data partitioning scheme. For the LINPACK benchmark, *DAXPY* accounts for most of the time spent inside *DGEFA*, however, the *DGEFA* subroutine was chosen to be the parallelization focus to reduce the number of messages being sent across the ranks. As in Figure 4, we use six MPI ranks, all of them have the same functionality and perform the same computation, except for rank 0, which also performs the inital data distribution, stores the results back to the file system, and computes the *DGESL* subroutine.

After successfully parallelizing the algorithm in software, three of the six ranks are targeted to run in the FPGA. This decision is just to show how software processes (ranks 3, 4, and 5) can be turned into engines without changing a single line of code for ranks (0, 1, and 2), following the peer-to-peer model between X86 processors and hardware engines. The *DGEFA* subroutine is converted to hardware manually, without using any C-to-gates compiler, however, nothing in the MSF or TMD-MPI paradigm prevents that.

Since each rank contains a full *DGEFA* subroutine, columns of matrix A are cyclically distributed across the ranks, that is, 1st column goes to rank 0, 2nd column to rank 1, and so on. This reduces the data communication in each iteration. After the first data distribution, which is only done once, only two broadcasts occur in each iteration, one column of matrix A and the corresponding pivot. These broadcasts are performed after the *DSCAL* function is executed and done by the rank storing the respective column, which means that each iteration will have a different broadcast source; therefore, there is communication between all the ranks. Finally, when all the data is computed, it must be sent back to rank 0, which will run the *DGESL* subroutine and end the algorithm with the residual calculation.

6.2. The DGEFA Benchmark Hardware. The *DGEFA* computing engine, shown in Figure 8, consists of a state-machine that has encoded the DGEFA benchmark flow, and a *BLAS1* block, which is a special-purpose fully pipelined engine that calculates the *BLAS* level 1 functions. The DGEFA state-machine issues send and receive commands to the MPE, similar to the MPI calls for X86 processors. The MPE implements the TMD-MPI protocol in hardware and gives the DGEFA engine the ability to communicate with all the other ranks in the system. The MPE has independent command and data FIFOs, that allow the streaming of data directly into the datapath of the *DGEFA* computing engine. Figure 8 shows how the *DGEFA* engine connects to the MPE.

6.3. Verification of the Results. At the end of the application, rank 0 has the final matrix, which is compared against the sequential version of the code run only in software. There is a maximum error of 0.64% with an average error of 0.0011% in the results due to fact that the X86 uses 80- and 64-bit precision floating-point units compared to only 32 bits in the engine, but that can be improved. The point being made is that by using the MSF we have a way to measure further improvements to the engine's precision or mixed (X86 + engine) precision calculations.

A caveat of our approach is that the communication latency between X86s and FPGAs is not known with precision because of cache effects, system load and memory traffic bring uncertainty about the exact memory transfer

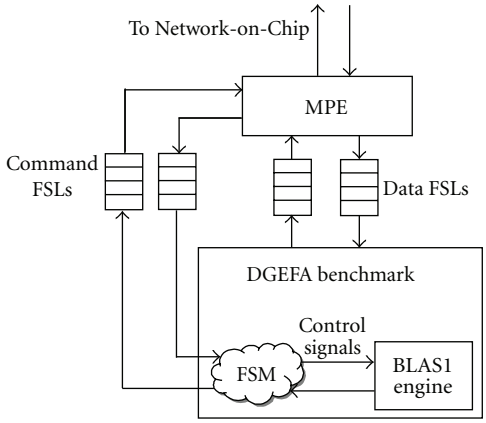


Figure 8: Interface between MPE and hardware engines.

latencies. The MSF module does not simulate the LLCC nor the FSB bus transactions; therefore, it is hard to predict the entire application's performance exactly. However, keep in mind that the focus of the MSF is correct functionality rather than accurate performance measurement. Nevertheless, we can estimate the LINPACK core performance based on measuring the most time consuming part of the application. By using the LINPACK function *second()* we know that the DAXPY loop takes 482 microseconds in the X86 processor (3.4 GHz), compared to 818 microseconds in the *DGEFA* engine (clocked at 100 MHz) measured in simulation. This is a fair comparison since there is no communication involved in that loop, just raw computation.

Due to the MPI paradigm and the DGEFA core implementation it is very easy to increase the number of ranks in the system, as long as there are enough resources in the FPGA. The code (C and VHDL) does not need to change at all to include more ranks. Based on preliminary synthesis results, we can place around 16 *DGEFA* engines (excluding the on-chip network, MPEs, MPI_Bridge, and the Xilinx FSB core) on the XC5VLX110 FPGA.

7. Future Work

Future work includes the support for external memory simulation models for those modules that have memory chips next to the FPGA; this will allow us to simulate designs with larger datasets because currently the hardware engines and embedded processors are limited to work with on-chip RAM. The MSF should be able to automatically generate top-level testbenches that include these external memory models.

Currently, the FLI module provides a one-clock read/write main memory latency, which is not realistic. By using Xilinx Chipscope in a placed and routed design we have been able to measure the latency of a memory read (MPI_FSB_Bridge requesting to read a memory location) to be between 40 and 50 clock cycles (approximately 150 nanoseconds) in an unloaded system. However, this changes in heavily loaded systems. Future releases of the MSF FLI module will include parameters or functions to introduce these latencies to further improve simulation accuracy, although changes in latency should not change the results of the application.

In this paper, only multiple M2B modules were simulated, but to simulate 15 FPGAs (including M2C modules) could be quite demanding on computing power, especially for large FPGA designs. Future work will include further performance measurements and optimizations to the MSF.

8. Conclusions

In this paper we describe a portable MPI-based approach for cosimulating multiple hardware engines implemented in FPGAs communicating with multiple X86 processes. Although, in this paper, we use an Intel-FSB-Xilinx-FPGA platform, the same concepts and ideas can be applied to other platforms.

Vendor-specific details should be hidden in a portable design and considered perhaps for further optimization. For initial stages of the design, a quick prototyping simulation environment such as the MSF can be very useful to accelerate the design flow and test the functionality as well as explore design alternatives. The MSF has demonstrated its usefulness during the development of a LINPACK system by allowing a fast compile-debug-modify-recompile cycle speeding up the design task because there is no need to run place and route to test the algorithm. The LINPACK system was cosimulated using six MPI ranks, half of them running as X86 software processes and the other half as hardware engines in simulation; all exchanging messages in a peer-to-peer fashion.

By virtue of using an implicit parallel programming model, the simulation of multiple FPGAs can also be distributed to multiple ModelSim processes and take advantage of multicore processors. We show that a simulation distributed across two ModelSim processes achieved a twofold speedup over a single ModelSim process simulation.

Similarly, latency tolerant designs are natural when the designer has an asynchronous, distributed, message-passing mind set. Therefore, communication latency between processors and FPGAs, although key for performance, does not need to be simulated to obtain a functionally correct system.

Glossary

ACP: Accelerated computing platform
FLI: Foreign language interface
FSB: Front side bus
FSL: Fast simplex link
HDL: Hardware description language
HPRC: High-performance reconfigurable computer
LLCC: Low-level communication core
LVDS: Low-voltage differential signal
M2B: M2 base
M2C: M2 compute
MPE: Message passing engine

MPI: Message passing interface
MSF: Message passing simulation framework
NoC: Network-on-chip
TMD: Toronto molecular dynamics.

Acknowledgments

The authers acknowledge the CMC/SOCRN, NSERC, and Xilinx for the tools, hardware, and funding provided for this project.

References

[1] The MPI Forum, "MPI: a message passing interface," in *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '93)*, pp. 878–883, ACM Press, 1993.
[2] CRAY, Inc., http://www.cray.com.
[3] SGI, http://www.sgi.com.
[4] Intel, Corp., http://www.intel.com.
[5] Xtreme Data Inc., http://www.xtremedatainc.com.
[6] DRC computer, http://www.drccomputer.com.
[7] "General purpose reconfigurable computing systems," Tech. Rep., SRC Computers, Inc., 2005, http://www.srccomp.com.
[8] M. Saldaña and P. Chow, "TMD-MPI: an MPI implementation for multiple processors across multiple FPGAs," in *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications*, Madrid, Spain, 2006.
[9] M. Saldaña, E. Ramalho, and P. Chow, "A message-passing hardware/software cosimulation environment to aid in reconfigurable computing design using TMD-MPI," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 265–270, December 2008.
[10] A. Petitet, J. Dongarra, and P. Luszczek, "The LINPACK Benchmark: Past, Present and Future," http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf.
[11] A. Patel, M. Saldaña, C. Comis, P. Chow, C. Madill, and R. Pomès, "A scalable FPGA-based multiprocessor," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, Calif, USA, 2006.
[12] Amirix Systems, Inc., http://www.amirix.com.
[13] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.
[14] H. Hubert, *A survey of HW/SW cosimulation techniques and tools*, M.S. thesis, Royal Institute of Technology, Stockholm, Sweeden, June 1998.
[15] T. Suh, H.-H. S. Lee, S.-L. Lu, and J. Shen, "Initial observations of hardware/software co-simulation using FPGA in architecture research," in *Proceedings of the 2nd Workshop on Architecture Research Using FPGA Platforms (WARFP '06)*, February 2006.
[16] M. N. Wageeh, A. M. Wahba, A. M. Salem, and M. A. Sheirah, "FPGA based accelerator for functional simulation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '04)*, vol. 5, pp. 317–320, May 2004.
[17] Mentor Graphics, Corp., http://www.mentor.com.
[18] Cray Inc., "CRAY XD1 FPGA Development," pp. 9–11, pp. 63–66, 2005.
[19] SGI, "Reconfigurable Application-Specific Computing Users Guide," pp. 9–12, pp. 223–244, January 2008.
[20] Nallatech, http://www.nallatech.com.
[21] HyperTransport Consortium, http://www.hypertransport.org.
[22] Intel, "Intel Quick Path Architecture (White Paper)," http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf.
[23] Mentor Graphics, "ModelSim SE Foreign Language Interface Manual," February 2008.
[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 450607, 22 pages
doi:10.1155/2009/450607

Research Article

OveRSoC: A Framework for the Exploration of RTOS for RSoC Platforms

Benoît Miramond,[1] Emmanuel Huck,[1] François Verdier,[1] Amine Benkhelifa,[1] Bertrand Granado,[1] Thomas Lefebvre,[1] Mehdi Aïchouch,[1] Jean Christophe Prevotet,[2] Yaset Oliva,[2] Daniel Chillet,[3] and Sébastien Pillement[3]

[1] *ETIS, CNRS-UMR8051, ENSEA, Université de Cergy-Pontoise, 6 avenue du Ponceau, 95000 Cergy-Pontoise, France*
[2] *IETR INSA—UMR 6164 CNRS, CS 14315, 35043 Rennes, France*
[3] *CAIRN—IRISA/ENSSAT, 6 rue de kerampont, 22300 Lannion, France*

Correspondence should be addressed to Emmanuel Huck, emmanuel.huck@ensea.fr

Received 15 March 2009; Revised 19 October 2009; Accepted 20 December 2009

Recommended by Lionel Torres

This paper presents the OveRSoC project. The objective is to develop an exploration and validation methodology of embedded Real Time Operating Systems (RTOSs) for Reconfigurable System-on-Chip-based platforms. Here, we describe the overall methodology and the corresponding design environment. The method is based on abstract and modular SystemC models that allow to explore, simulate, and validate the distribution of OS services on this kind of platform. The experimental results show that our components accurately model the dynamic and deterministic behavior of both application and RTOS.

1. Introduction

Nowadays, algorithmic complexity tends to increase in many domains such as signal, image processing or control. In parallel, embedded applications require a significant power of calculation in order to satisfy real-time constraints. This leads to the design of hardware architectures composed of heterogeneous and optimized computation units operating in parallel. Hardware components in SoC (System on Chip) may exhibit programmable computation units, reconfigurable units, or even dedicated data-paths. In particular, reconfigurable units, denoted here as Dynamically Reconfigurable Accelerators (DRA), allow an architecture to adapt to various incoming tasks at runtime.

Due to their intrinsic complexities, such heterogeneous architectures need even more complex management and control. In this context, the utilization of an RTOS (Real Time Operating System) is more and more required to support services such as communications, memory management, task scheduling, task placement, and so forth. These services have to be fulfilled in real-time according to the application constraints. Moreover, such an operating system must provide a complete design framework independent of the technology and of the hardware architecture. As for standard computers, the RTOS must also provide an abstraction and a unified programming model of the heterogeneous platforms. This abstraction permits to drastically reduce the time to market by encouraging re-usability.

Embedded RTOS for SoCs are of great interest and are subject of several significant studies. In the context of reconfigurable architectures, a study in [1] has determined and classified the different services that operating systems should provide to handle reconfigurability. Today, two different approaches have emerged for the development and the integration of these dedicated services. The first consists in utilizing an existing standard RTOS (RTAI, RTLinux, VxWorks, etc.) and in adding functionalities dedicated to the management of the reconfigurable resources [2]. The second is to develop a specific RTOS from scratch by implementing the necessary functionalities devoted to the management of the reconfigurable resources [3, 4].

The design process of such complex and heterogeneous reconfigurable systems requires method, rigor and tools. The OveRSoC framework is developed to take into account

both the RTOS, and the platform to propose an efficient exploration of the design space. The OverSoC methodology is based on 4 important design concepts: exploration, separation of concerns, incremental refinement and re-usability.

Firstly, a number of design choices have to be done prior any implementation, especially when the platform itself is designed and tailored for a specific application. We advocate the use of a high level model of Reconfigurable SoCs (RSoC) in order to explore different critical design choices. Among these important choices we distinguish two exploration issues:

(i) the exploration of the application partitioning onto the processing resources (topic already addressed in the literature [5, 6]),

(ii) the exploration of the RTOS services distribution and their algorithms.

Each design strategy belonging to these exploration levels is manually made by the designer. But the proposed method helps the designer to easily and quickly build the executable specification of the corresponding systems. The underlying tools then bring performance evaluations in order to analyze and compare design strategies. The design choices corresponding to the second exploration issue (RTOS) are the architecture of the embedded RTOS (centralized or distributed, OS services organization, software, or hardware implementation, etc.), the services algorithms (scheduling policies, etc.), the interactions between OS service functions and underlying resources (reconfigurable, memories, interconnects) and the software programming model.

Secondly, once validated the candidate design solutions are incrementally refined toward lower levels of abstraction down to the final implementation. The OveRSoC methodology permits the separation of concerns during the modeling and refinement process. It also defines modeling rules that facilitate independence and re-usability between components. For each design concern specific and related refinement steps are proposed. The resulting methodology serves as a design map for the designer of RSoC platforms.

Finally, the method imposes a functional approach at each level of abstraction which allows the validation of the application functionality besides the performance evaluation.

As a consequence, in the rest of the paper the problem of OS design is presented as a platform management problem. This paper presents the OveRSoC methodology and the related framework that consists of a set of SystemC models. The associated graphical exploration environment is also presented.

The remainder of the paper is as follows. Related work is described in Section 2. Section 3 presents the OveRSoC methodology and the corresponding tool for RTOS design. The flexible SystemC abstract RTOS model which allows RTOS service distribution and customization is presented in Section 4. Section 5 describes the RSoC architecture modeling step. Section 6 provides experimental results while Section 7 brings out our conclusions and presents the perspectives of this work.

2. Related Work

One of the main issues in reconfigurable platforms consists in determining efficient control mechanisms that may have dynamical properties in the sense that they must take on-line decisions from unpredictable system properties [7]. Several studies such as [3, 8] aimed at identifying the properties of the RTOS that can take dynamic reconfiguration into account. Specific properties such as application partitioning and tasks placement are described and placed in the context of reconfigurable computing which is often based on a farm of reconfigurable circuits. In [8], authors present one of the first attempts to develop an OS dedicated to the management of reconfigurable resources.

For the particular SoC domain, the authors in [9] list important properties to stress the usefulness of an OS to manage heterogeneous and static resources.

Adding reconfigurable units in a chip brings up many other issues from a design point of view. Introducing an OS for the management of an RSoC is of high interest in the research community [10]. Indeed, the partial reconfiguration abilities of current architectures need to be fully exploited in order to improve performance, cost, power-efficiency and time-to-market. Even if classical software approaches can be used, the OS then needs to be adapted to this new computation paradigm. More precisely, specific services are requested to manage the specific properties and resources of the dynamically reconfigurable units.

Designing a complete RSoC including an RTOS is a very complex task and requires appropriate methodologies. In this section we firstly introduce constraints on a dedicated RTOS for RSoC, we then discuss a proposal of methodologies in order to design these circuits efficiently.

2.1. Dedicated OS Services. The required specific services for RSoC can be roughly decomposed in four categories:

2.1.1. Spatiotemporal Scheduling. The task scheduling service is obviously one of the most important features of a multitasking OS. Scheduling of hardware tasks on reconfigurable areas adds a spatial dimension to the classical temporal problem [11]. This is defined as the spatiotemporal scheduling problem. The mapping of hardware tasks onto the reconfigurable unit can follow two spatial schemes according to the technology [3]: 1D or 2D schemes. While the 1D technique is simple to support, its performance in terms of computation density is low. On the other side, the 2D placement technique ensures a more efficient utilization of the reconfigurable area, but the associated algorithms are more complex.

2.1.2. Reconfiguration and Resource Usage Management. The resource management is very close to the placement service which needs to know the global state of the system. The resource table needs to be extended to store specific information necessary to manage the reconfiguration [2]. We can cite for example the area information for each reconfigurable task, the task communication needs which must be ensured

when the task is placed on the reconfigurable resource, the form factor, and so forth. The area fragmentation problem also appears when managing reconfigurable resources [12, 13]. This problem can prevent the placement of tasks while there is enough area within the reconfigurable resource. In this case, the designer can decide to implement a defragmentation service into the OS to limit the task placement rejection.

The reconfiguration latency of DRA represents a major problem that must be taken into account. Several works can be found addressing temporal partitioning for reconfiguration latency minimization [14]. Moreover, configuration prefetching techniques are used to minimize the reconfiguration overhead [15]. A prefetch and replacement unit modifies the schedule and significantly reduces the latency even for highly dynamic tasks.

2.1.3. Task Preemption and Migration. hardware task migration is an interesting property that requires the implementation of the hardware task preemption service [16]. Efficient implementation of preemption and migration requires several additional OS services, such as online communications routing and spatial placement. To limit the scheduling overhead and the number of configuration phases, which can be very time consuming, some OS prevent the preemption of hardware tasks. Non preemptive operating systems are known to be more deterministic, but do not take full advantage of platform flexibility. The conditions allowing more complex hardware task preemptions are defined in [17]. In this article, the authors describe three types of requirements allowing to perform multitasking on FPGA. First, save and restore mechanisms of current state of all registers and internal memories are required. Second, the configuration manager must obviously support fast configuration and readback of the FPGA. It must also have complete control over all the clock signals in order to freeze execution during context switching. Finally, it requires an open bitstream format in order to readback the status information bits.

As an example, preemption of hardware tasks have been studied in [18]. The authors present prospective architectural extensions of SRAM-based FPGA devices allowing a very fast and efficient context save and restoration. The proposed architecture supports the hardware defragmentation.

2.1.4. Flexible Communications. This property deals with the inter-task communication property of an OS and impacts the routing service [19]. The communication functionality is an important part of the system to ensure the data exchanges between all tasks, whatever their type or localization. Considering the localization of the tasks, communications are classically divided into two different types:

(i) the global communications: this communication level enables data exchanges between the different available resources (e.g., DSP, processors, reconfigurable units, etc.).

(ii) the local communications: this level ensures the data routing between different tasks placed simultaneously into the same reconfigurable area.

The global communication structures have to support flexible throughput and guaranteed bandwidth. In this case, OS services must provide the capacities to manage these structures. The requirements of the local communications within the reconfigurable area are quite different. Tasks implemented within this area are dedicated to intensive computation and are generally constrained by real time execution. In this case, communications do not support any delay nor excessive latency.

2.2. RSoC Dedicated Methodologies. Several studies tend to abstract the reconfiguration management by working at a system-level model. This level enables the exploration of systems while software, hardware or reconfigurable parts are not completely defined. It also enables the validation of various configurations to find the most efficient solution.

In order to introduce the reconfiguration in Symbad [20] which is a system-level codesign platform for SoC, the refinement phase has been modified to handle static and reconfigurable modules [21]. Specific simulation parameters, such as the reconfiguration time, are taken into account. Associated tools enable the evaluation of the reconfigurable contribution to the system performances. In [22], the authors propose a methodology in order to implement an application in an RSoC. This methodology is based on a UML descriptive model of the software parts and on a SystemC description of the architecture. Currently, these works do not take the dynamicity of the reconfiguration into account.

The collaborators of the Adriatic project propose an original methodology that handles dynamic reconfiguration [23]. The reconfigurable block is composed of a controller that launches or stops reconfigurable tasks, and features an input router that dispatches data among active blocks. Adriatic then proposes high level estimation of performances. Different strategies and approaches of estimation, simulation and partitioning are implemented in the Perfecto [24] and ReChannel [25] frameworks. Unfortunately, none of these works considers the development of an OS in order to dynamically manage the RSoC.

New approaches tend to provide a high-level hardware design model while managing the hardware implementation efficiently. This goal is achieved by a multi-languages approach.

In [26], the authors develop a framework based on the RTL language HIDE for implementation purpose, and on Handel-C to describe hardware at a higher level of abstraction. At present, the proposed framework does not handle dynamically reconfigurable architectures.

The multi-languages strategy is also used in the European Project Andres [27] which addresses heterogeneous systems. It is built around the HetSC methodology for the specification of the software part and the OSSS+R SystemC library for the reconfigurable part. Andres also includes a part of analog mixed design by supporting the SystemC AMS.

A special case of RTOS generation is the definition of dedicated OS services for DRA. The work presented in [28] addresses this problem by proposing a RTOS/SoC codesign framework. The customized RTOS is automatically generated from existing OS basic blocks which are available in software and/or in hardware. The 4S project [29] provides a design flow to develop RSoC platforms including an OS. In this project, algorithms are implemented into tasks which are mapped onto reconfigurable or non reconfigurable modules. The proposed tool provides information about the performances of each task for a given mapping. In an exploration step, the OS manages the implementation of tasks within reconfigurable units and generates flexible communication mechanisms. At present, these projects do not include the OS definition as part of the design process.

As a conclusion, adding reconfigurability in a platform imposes the management of hardware tasks at run-time. These tasks have to be placed into a reconfigurable unit in a dynamic and flexible manner. To ensure this management, some OS services need to be adapted (synchronization, migration, etc.), but some other services are completely new and need to be developed from scratch (spatiotemporal scheduling, fragmentation management, etc.). In the literature, to the best of our knowledge, no work proposes a complete solution, neither on real platforms nor in simulation, for the DRA management. The main contribution of this paper is to propose a unified modeling environment where all the needed services can be specified, tested and validated when distributed onto an heterogeneous multiprocessor platform. In this paper we do not provide and describe new spatiotemporal algorithms nor defragmentation methods but an open platform for the exploration of these complex algorithms where existing and upcoming methods for DRA management may be evaluated and compared. The services and the underlying platform are part of the exploration process. This objective has been reached thanks to the following contributions:

(i) a design methodology adapted to the exploration of the RSoC specific services,

(ii) a tool implementing this methodology,

(iii) a set of generic simulation models of MPSoC (Multi Processors System-on-Chip) components,

(iv) a high-level model of a DRA,

(v) a top-down refinement process.

3. The OveRSoC Methodology

In this section, we describe the methodology which is developed in the OveRSoC project and the tool that implements it. Our main goal is to provide a simulation framework for hardware/software design exploration of an RSoC including a dedicated RTOS. The framework is based on four main concepts: a methodology based on several design and analysis steps, the automation of simulation code generation from a library of basic blocks, the separation of concerns and the capability to simulate heterogeneous abstraction levels during the modeling process.

3.1. Platform Exploration Flow. The global methodology focuses on the original concepts addressed by OveRSoC, that is, the exploration of a distributed control of dynamic reconfiguration. In this way the methodology aims to explore the appropriate OS services that will be necessary to manage the RSoC platform. It relies on an iterative approach based on the refinement concepts as depicted in Figure 1.

The input of the exploration flow consists in specifying both the application and the system constraints. The RSoC platform model requires parametrization. The application is described as a set of tasks implemented whether in hardware or software. Their communications and synchronizations are also described as a graph of connections and dependencies. These dependencies can represent either pure data streams or synchronization mechanisms. Since version 2.0, SystemC supports a very powerful generic model of computation [30] but at the present time we only consider Communicating Sequential Process, Data-flow, and Kahn Process Networks [31]. These models satisfy the set of properties of the digital and signal processing domains that we address in this work. As imposed by the methodology, the functional behavior of each task must be defined as a pure C specification whether the tasks are executed in software or in hardware. During the early modeling steps, we use a common specification for the software or hardware implementation of a task. But for all the tasks, information about the execution time, periodicity, deadline are taken into account and considered as implementation specific attributes. This type of information may be either first estimated and refined afterwards or directly obtained by other tools that are capable of delivering accurate timing in the case of reused software or hardware IPs.

The basic RSoC platform considered is composed by three main types of components: the OS that manages the entire structure, the Processing Elements (PEs): the processors and the DRA, and the Communication Elements (CEs) composed of a communication media and a memory hierarchy. The OS may be distributed on the PEs of the RSoC platform (at least one processor and one DRA). The framework provides a set of models stored into the system library for each type of component. The library can be extended by adding new models to take into account new architectures. All the components feature their own list of design attributes. These attributes are used to customize each block within the RSoC platform. For example, the scheduler algorithm of a specific instance constitutes an attribute for an operating system, the latency of a specific task corresponds to an attribute for the application, the numbers and types of available resources within the reconfigurable area constitute one of the attributes that describe the DRA.

Once the platform architecture is defined and customized, the central work for the designer is to specify the different services that are required by the operating system in order to manage the global platform. Some services are available in a service library, but it is also possible to create new ones by specifying their behavior.

The validation of the design is based on the notion of metrics. Metrics are component specific measurements that can be reported to the designer during the simulation. They

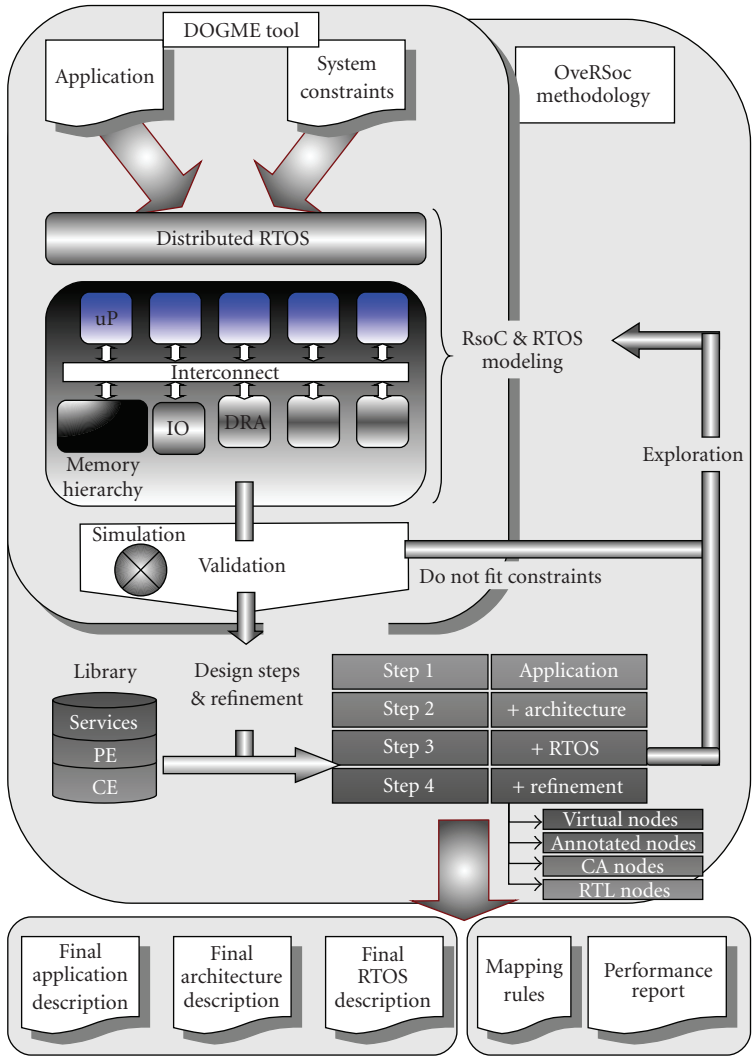


Figure 1: The OveRSoC exploration and refinement flow. Exploration is defined as an iterative process: modeling, simulation/validation and exploration. The inputs of the method are the specification of the application as a pure functional C code, and the system constraints. Once the system validated, the design process starts again at a lower level of abstraction until the final system description. At each level of abstraction, the goal of the exploration depends on the separation of concerns paradigm (Section 3.2). This paradigm is defined as a 4 steps process where the following concerns are successively addressed: application specification, architecture description, RTOS definition and platform refinement.

help the designer to verify the system constraints such as the PE workload, the communication congestion and so forth.

Examples of metrics that are already provided by the library components concern the tasks sequencing, the number of preemptions, the usage of resources and all events that may occur during the execution (semaphore's pend and post, etc.).

In particular, these metrics help to check the respect of the timing constraints. Obviously, the functional behavior of the application can still be validated by the designer. Once the attributes are completely defined, the whole platform is simulated in SystemC and metrics are evaluated. The analysis step is then manually performed by the designer in order to analyse the results of the simulation and to estimate the impact of specific attributes on the overall performances. The designer may then modify the value of some attributes and iterate the global simulation of the platform to explore the design space.

For the validation of the design choices, both the application (functionality) and the underlying RSoC platform (concurrency and timing) are simulated at high level in order to substantially decrease the simulation time of the whole platform. The exploration flow is conceived in a hierarchical way, according to the refinement concepts, and allows the designer to refine progressively his description of the platform to get more and more detailed results. We identified 4 refinement levels described in Section 3.3. At the highest level, we only consider the duration of tasks and RTOS calls, but not the memory nor the communication time. Then new attributes and metrics may appear as the description

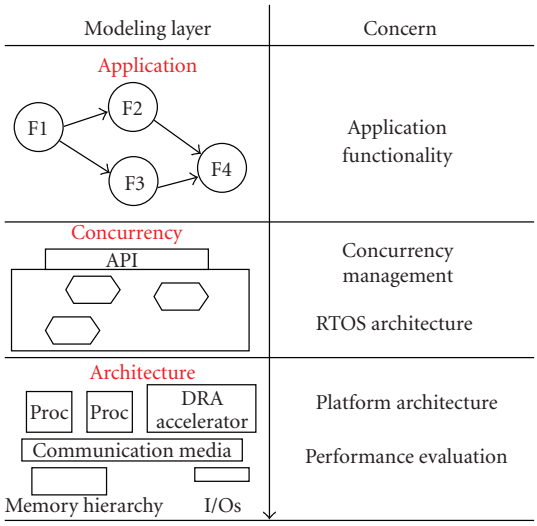


Figure 2: Our modeling approach follows the separation of concerns paradigm. The Application layer is a set of pure C functions and focuses on the functional specification of the algorithms. The Concurrency layer is a set of RTOS services and focuses on the distribution of these services. This layer also brings concurrency between threads according to the type of the associated PE. The Architecture layer is a set of parametrizable PEs, CEs and memories and represents the embedded platform. This layer also brings accurate timing evaluations.

becomes more accurate. For example, communications that are not taken into account in a coarse level of description may be accurately described to get more realistic values of the execution latency. New metrics like deadlocks on an interconnection network may also appear and provide the designer with new information about the global functioning of the platform.

3.2. Separation of Concerns. One of the main challenges of the proposed method is to keep the RTOS model as abstract as possible for exploration reasons while providing accuracy of performance estimation. The RTOS is maintained at a high level of description in order to easily add, remove, and deploy services without impacting the binary code of the cross-compiled application. The application is compiled once and the designer cannot only modify and refine the implementation of the RTOS services, but also scale the number of processors and DRA in the platform. As a result, the modeling space is separated into three independent layers depicted in Figure 2 according to the principle of the separation of concerns [32].

The top layer focuses on the functional specification of the application. This is described as a pure functional C code partitioned in C functions.

Then, some of these functions are associated with the notion of task in the following layer. Functional code calls RTOS services through a standard API (Application Programming Interface) as explained in Section 4. Communications between tasks depend on the synchronization services provided by the RTOS, for example, mutex, semaphores, fifos, mailboxes, and so forth.

In the next step, the OS layer deals with the concurrency between explicitly defined software processes. To reach this goal, we have developed a flexible SystemC model of a RTOS which is described in Section 4. Concurrent tasks are created thanks to specific services within the RTOS API. Multiple scheduling algorithms can be tested at this level according to the application constraints and possible task mapping to the underlying architecture without modification of the functional layer. In this layer, the designer can also explore the architecture of services into the distributed RTOS.

Finally, at the Architecture layer, the architecture of the embedded system is specified as a composition of heterogeneous processing elements (PEs) and communication elements (CEs). Each PE and CE may be modeled at different levels of abstraction and a refinement process can be performed without impacting the other modeling layers. Precisely, an ISS (Instruction Set Simulator) of a general-purpose processor executing a sequence of instructions is a refined model for an abstract function block. The independence of the hardware layer is ensured by a low level API, the Hardware Abstraction Layer (HAL) that always provides the same low-level services but with more or less accuracy as described in Section 3.3. This layer is also responsible for metrics' evaluation: execution time, processor utilization, memory usage, and so forth. Adopting such a modeling approach allows to reach the presented objective, that is, to explore the RTOS implementation at a high level while providing accurate performance evaluation of the entire system. According to the RTOS timer frequency, we observed on our application (see Section 6) that the execution time of the RTOS services represents ≈3 percent of the total application execution time. This observation corresponds to the results presented by Kohout et al. in [33]. Authors characterized the RTOS overhead according to the processing power used by the applications. The measured overhead grows from 2% to 9% for a preemptive RTOS and from 0.6% to 1.25% for a RTOS using a nonpreemptive strategy. But this is only for a monoprocessor system. In our case, when deploying an application on a MP-R-SoC, scheduling strategies and communication will completely change the system behavior and the waiting state durations. To deal with the OS overhead, we propose to keep the OS services at high level to ease exploration of its distribution or implementation. This observation is consistent with our approach that will provide accurate performance estimation on the Application layer which thus represents at least 90% of the total execution time.

3.3. HAL Transactor and System Refinement. Independence between modeling layers is ensured by a set of constant and standard services provided to the upside neighbor layer:

(i) independence between the *Application layer* and the *Concurrency layer* is ensured by the OS API,

(ii) independence between the *Concurrency layer* and the *Architecture layer* is ensured by the HAL API.

Table 1: Example of services provided by the OS and HAL API.

Service component	OS API
Task management	void OScreateTask(code_pointer_t f, intu8 priority);
	void OSdeleteTask(int task_id);
	...
Semaphore management	sem_desc OScreateSem(sem_state init);
	void OSreleaseSem(int sem_id);
	...
Timer management	void OS_time_delayHMSM(int h,int m, int s, int ms)
	...
	...
Architecture component	HAL API
PE	void compute(task_t* t);
	save_context(task_t* t);
	restore_context(task_t* t);
	timer_set(int nbms);
	timer_set_irq_handler(code_irq_handler_t f);
	timer_start();
	timer_stop();
CE	oversoc_t_rsp_t transport(oversoc_t_req_t *REQ);

The set of services provided by the OS depends on the chosen services. An example of service functions provided by the OS and HAL API is presented in Table 1. PEs and CEs provide to the OS components execution and transaction services similar to those presented in [34]. The call to the HAL services remains constant during all the refinement process but their implementation depends on the accuracy of the underlying layer. So both the OS and the HAL API allow to explore and refine lower layers while keeping higher layers unchanged.

Indeed PEs can represent abstract processing components when modeled at high level. They can also represent cycle-accurate processor, FPGA, or dedicated hardware models when described at lower levels. When the embedded application is partitioned and assigned to a PE, the PE mainly provides a *compute()* and *transport()* pseudo service to the RTOS, allowing a timed simulation for the computation and the communication. It also provides a service to trigger interrupts as components of the corresponding RTOS HAL.

The simulation accuracy then depends on the description of the internal architecture of the PE. We identify and advocate 4 refinement levels depicted in Figure 3.

(i) *Virtual nodes*: the PEs are used as empty boxes and the simulation is not timed. It corresponds to the Programmer View of the TLM approach [35], that is, a pure functional verification at high simulation speed.

(ii) *Annotated nodes*: the PEs are described as simple tables containing predicted execution times. The timings correspond to a back annotation of the execution time of each application basic block (Programmer View plus Time [35]) but without any modification of the application source code.

(iii) *Cycle accurate nodes*: at an intermediate level, software PEs are classically modeled as ISS (cycle-accurate) as explained in Section 5.2. In Section 5 we describe an equivalent model for the hardware PE (the DRA). From this refinement level, the HAL is implemented as a transactor, that is, a modeling artifact that translates transactional calls to RTL signals activations.

(iv) *RTL nodes*: at the lower abstraction level, a PE can still be described as an RTL model providing cycle-accurate timing evaluations and bit-accurate informations.

In a more general manner, thanks to the SystemC blocking calls mechanism, the Architecture layer interacts with the simulation core (SystemC) to advance the simulation time of the caller process according to the executed task. As for the synchronization and the preemption of the SystemC processes, it is ensured by the upper level which manages notification and waiting of SystemC events as described in [37]. In the case of MPSoC platforms, synchronization between processors is ensured by interruptions and by a hardware shared semaphore model. But whatever the chosen abstraction level of the Architecture layer, the Concurrency layer (i.e., the OS services) remains at the same abstraction level. This level is called SAT (Service Accurate plus Time) and is described in Section 4.

3.4. The DOGME Tool. Due to the complexity of the exploration process, the HW/SW designer needs tools to apply the OveRSoC methodology. The DOGME (Distributed Operating system Graphical Modeling Environment) software provides an integrated graphical environment to model, simulate, and validate the distribution of OS services on RSoC. The goal of the tool is to ease the use of the exploration methodology and to generate automatically a complete executable model of the RSoC platform (hardware and software). The automation is based on a flexible SystemC model of RTOS described in Section 4. This RTOS model is a package of modular services. To develop each service, an Object Oriented Approach has been adopted and implemented using the SystemC 2.2 library. This tool allows an application specific RTOS to be built by assembling generic and custom OS service basic blocks using a graphical editor [38]. The application is linked to the resulting OS thanks to a standard POSIX API. Finally, the entire platform is simulated using the SystemC kernel.

The developed tool follows five main design steps represented in Figure 4.

(i) *Design of the platform*: the design phase consists in choosing and instantiating toolbox components into the graphical workspace editor in order to assemble the OS services and distribute them onto the RSoC processing elements. Figure 5 shows an example of RTOS composition including services like task management, scheduling, semaphore, IRQ controller... At this step the designer will successively, and according to the separation of concerns paradigm, take decisions about

(i) functions mapping into threads,
(ii) hardware/software partitioning,
(iii) instantiation of the required services,
(iv) distribution of the services onto the PEs.

(ii) *SystemC source code generation*: after interconnecting all components and verifying the bindings between services, the structural source code of all the objects that are instantiated into the platform is automatically generated.

(iii) *Compilation and simulation of the platform*: to complete the design of the platform, the parametrized structural SystemC description is combined with the behavioral source code of the components provided by the user. The global SystemC description is compiled and simulated.

(iv) *Analysis of the simulation results*: graphical diagrams are produced to visualize the evolution of the system metrics during the simulated time. This step helps the designer to evaluate the current design quality. It acts as a decision guide for the exploration of the design solution space.

We are currently implementing the DOGME tool as a stand-alone application based on an Eclipse Rich Client Platform [39]. Typical project management functions like importation of platforms or components into the standard library are supported as well as the creation of new platform models. Reusability is achieved in the tool by the possibility to add the newly created platform to the standard library. All data manipulated by DOGME are loaded and stored using a proprietary XML format dedicated to embedded software modeling as depicted in Figure 4.

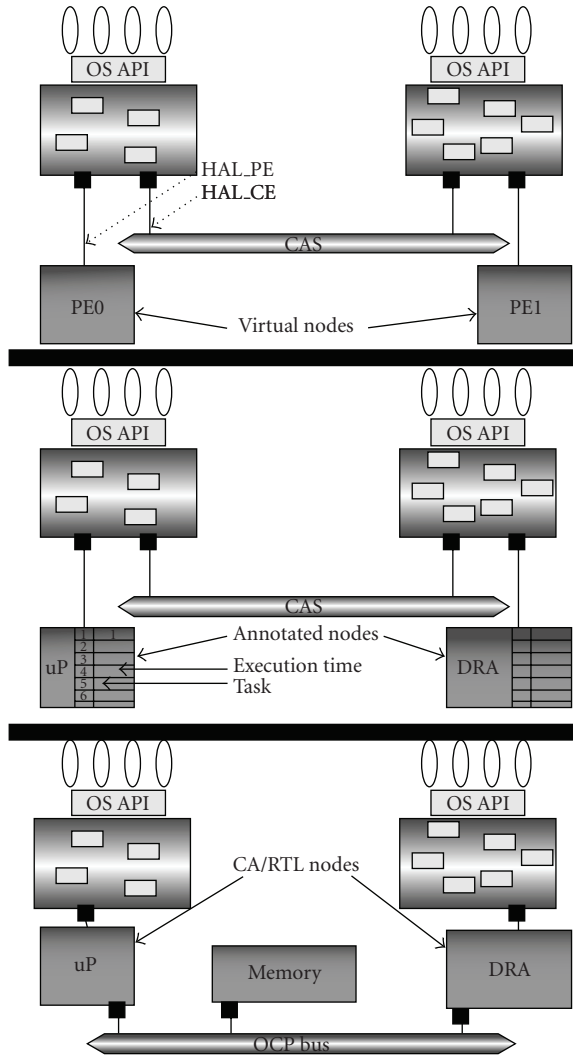


Figure 3: Example of refinement of the minimal RSoC platform. The first level begins after hw/sw partitioning of the application and corresponds to virtual nodes. The second level refines PEs to annotated nodes. At this level, each task has an estimated execution time. The CE is modeled as a transactional bus called CAS (Calling Abstraction Service). The two last levels correspond to Cycle Accurate or RTL nodes. The global CE has been refined to an OCP bus [36]. The memory accesses are now taken into account accurately, that is, all the communications can be evaluated at the lower level.

4. Distributed RTOS Model

This section presents the essential mechanisms needed to jointly model and simulate hardware/software tasks and the RTOS in SystemC.

4.1. A RSoC Model Based on RTOS Services. In order to model complex embedded platforms composed of multiple parallel and heterogeneous (and reconfigurable) resources, it is important to be able to jointly model the functional software, the underlying hardware and the glue between, which is generally composed of RTOS instances.

In the step 3 of the design process (see Figure 1), to explore the design solution space, we choose to model the system at a high level of abstraction, where the hardware is partially hidden. We focus our modeling process on the services provided by the platform.

At the Concurrency layer (see Figure 2), we address the SAT level of abstraction: Service Accurate plus Time. This allows us to very quickly simulate the behavior of the application, compared to lower detailed levels of abstraction. This level of concern is different from the Donlin's CP+T

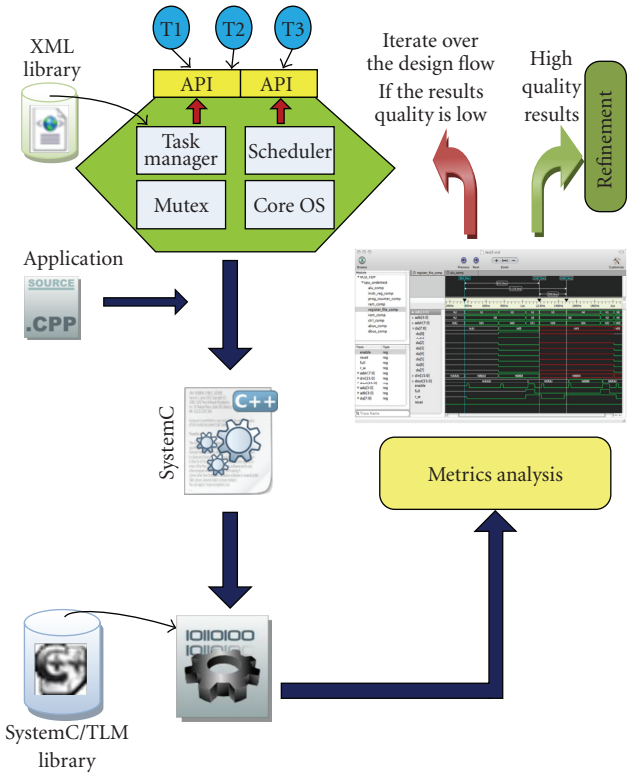


Figure 4: The DOGME tool brings facilities to manipulate the components of the library. These components model RTOS services for the control of an RSoC platform. In the library the services are described both by a SystemC generic source code and an XML exchange file. The designer graphically instantiates the components, then the tool automatically adds debug components for metric evaluation into the specification and generates the code of the corresponding platform. The platform is compiled and linked with the SystemC libraries and simulated thanks to graphical interfaces. The designer can finally evaluate the metrics of his platform and can take decisions about exploration or refinement.

(Communicating Processes with Time) level [35] which mainly focuses on hardware modeling but which does not include the RTOS services. This level of modeling implies that the architecture is not modeled explicitly, all the application tasks are functional, annotated with approximated or measured execution timing, and all the RTOS services are explicit and timed.

The core element of our distributed architecture model is a high-level functional model of a RTOS written in SystemC. Since SystemC does not support OS modeling facilities in its actual version, a first step was then to extend SystemC with embedded software modeling features [37]. The works presented in [40–43] are examples of simulation environments dealing with this challenge.

The proposed RTOS model [37] acts as a *Service Accurate + Time* model of a virtual PE (processor or DRA) in the sense that all the necessary services of an embedded RTOS are modeled as independent modules with their own behavior and timing. The RTOS model is built as a collection of *service modules* implemented in the form of a hierarchical `sc_modules` to foster high level exploration of custom architectures. The main RTOS model instantiates all its modules and uses `sc_export` to provide a global API to the application code as illustrated in Figure 6. Each *service module* has its own interface that furnishes the corresponding services' functions to the embedded application. This model includes mechanisms for modeling dynamic creation of tasks, task preemption and interrupt handling as described in [37]. Figure 6 illustrates the hierarchical structure of the SystemC RTOS model composed of the following service modules:

(i) a task manager that keeps the information and properties of each task according to its implementation (software or hardware): state, context, priority, timings, area, used software or hardware resources ...

(ii) a scheduler that implements a specific algorithm: EDF [7], HPF [7], horizon [44], ...

(iii) a synchronization service using semaphores.

(iv) a time management service that keeps track of time, timeouts, periods, deadlines ...

(v) an interrupt manager that makes the system reactive to external or internal events.

(vi) a specific simulation service (advance time).

Each service module is modeled as a SystemC hierarchical `sc_channel` and is symbolized in the figure using the SystemC representation [30]. A service module thus provides several service functions through its interface.

For example, the task manager provides the following functions: create (dynamically) a task, delete a task, get the state of a task, change the state of a task... The task creation function associates a simulation process (and thus concurrency) to one of the pure C function present at the Application layer.

Some service functions are accessible from the Application layer through the OS API. Those are called external service functions. Others are only accessible from the other service modules through a SystemC port to establish inter-module communications and are called internal service functions.

At this layer, timed simulations of the application use a specific simulation call (called OS_WAIT()), associated to each bloc of task code between two system-calls and redirected through the Concurrency layer toward the Architecture layer. This service, represented in Figure 6, allows each function to progress in time. In addition, each OS service function within the OS itself may also be annotated with timing information (depending on the processor) allowing a timed simulation of a realistic system.

Actually the system library provides a set of basic generic services: interrupt management, timer management, inter-tasks synchronization, and memory management. It also provides hardware and software specific services such as the task management of software or hardware tasks, software scheduling policies and hardware placement algorithms.

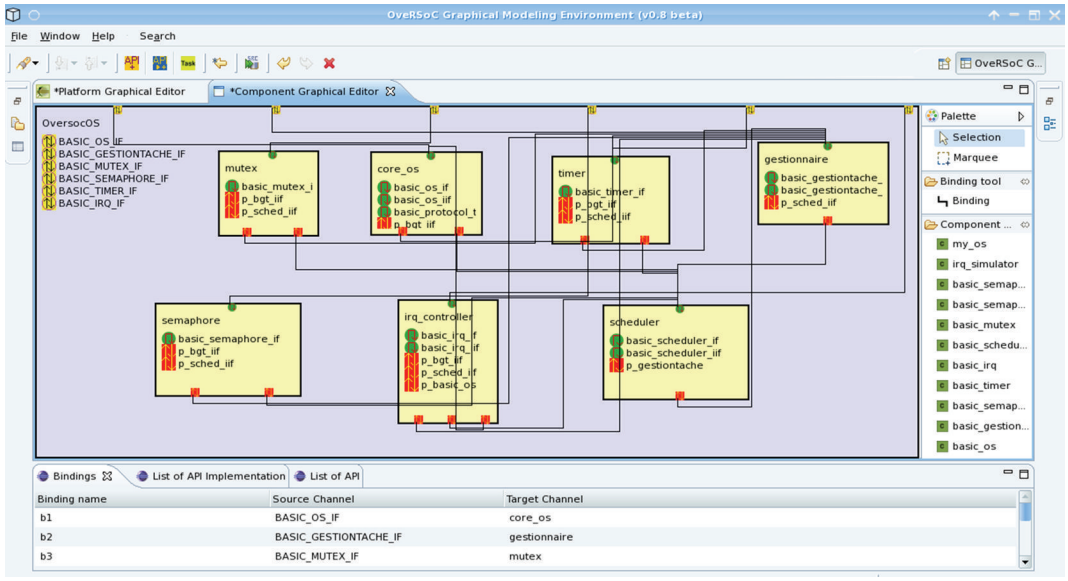


Figure 5: The DOGME tool represents a distributed RTOS through hierarchical views: the Component Graphical Editor, where the services are organized inside each PE, and the Platform Graphical Editor, where the groups of services are composed according to the number and type of PEs into the RSoC platform. Here the Component Graphical Editor is shown. It uses toolbox components to specify and customize the services of a dedicated group. Each service is modeled as a software (C++) component having ports and interfaces. Each service component provides several service functions.

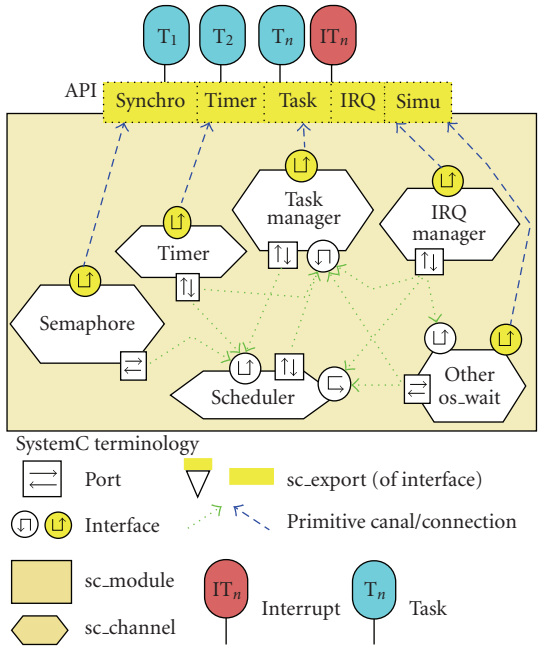


Figure 6: The modular RTOS model and its composed API. Each OS service exports its own interface to the application. Services are connected together to ensure the global OS coherency and behavior.

4.2. Distant Communications and Services Requests. We extend the model for distributed multiprocessor architectures exploration with the following features: the whole application is decomposed into multiple threads sharing the same addressable memory, the application is statically partitioned onto multiple processing nodes, each processor has its own scheduling strategy (policy, priorities, etc). All inter-processor communications are modeled using transactions with respect to TLM 1.0 methodology. A unique `transport` method is used for both requests and replies. All communications are currently performed instantaneously but this allows a communication refinement process and thus a time accurate simulation by introducing bus-related or network-related timings into transactional ports.

Our approach for modeling distributed OS services is inspired from the middleware philosophy which consists in using proxies and skeletons services. A proxy service provides a local entry point to a distant service accessible through an interconnection infrastructure. This adds dedicated ports and interfaces to the RTOS (and also on services modules needing to communicate).

Figure 7 illustrates transactions between two local semaphore services (proxies) and a shared distant semaphore implementation (skeleton). `Get` and `release` semaphore invocations are performed locally to the proxy which forwards transactions to the distant service. By using a simple `transport` method, all distant calls put the caller tasks into an active waiting state. In case of access conflicts, the shared service has its own arbitration policy. Then, replies are sent back to the caller at the end of the service execution.

Communication from a distant service to local proxies are performed by using signals which are similar to interrupt requests that are managed by local proxies. Suspended tasks may then be resumed by their own schedulers depending on local policies.

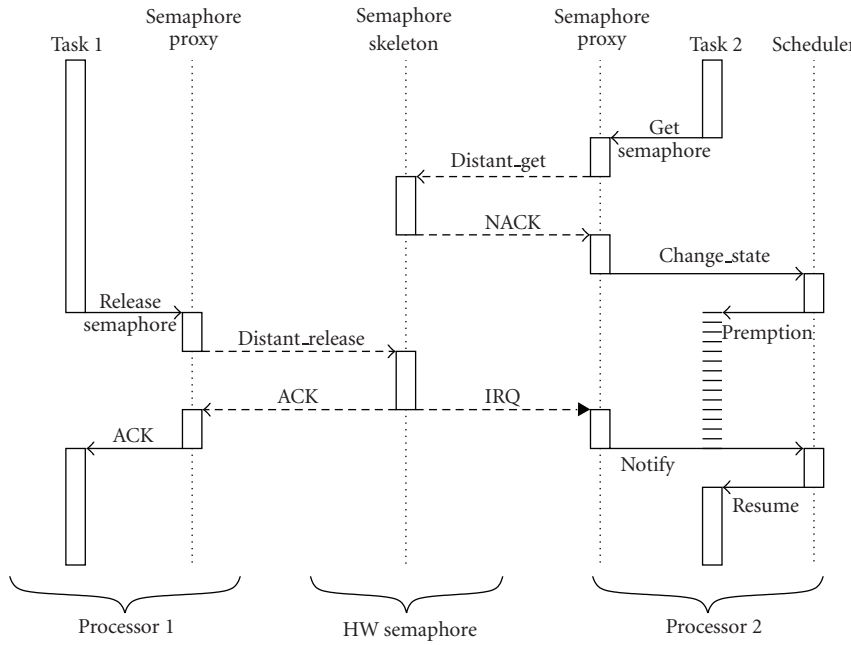


Figure 7: Activity diagram of local/distant calls to a shared semaphore proxy/skeleton between two OS models.

Based on this distant service invocation, we can easily imagine and construct a model of a shared distant synchronization service (potentially implemented in hardware), like a semaphore. Then it allows to quickly map the application onto a multiprocessor platform and evaluate the potential acceleration that distribution of computations could potentially allow, as shown on Figure 8.

Based on this mechanism, we can design a new RTOS with dedicated services for a DRA. We can then explore and evaluate their behavior, as shown in Figure 9, and try different scheduling policies specific to hardware IP placement on the DRA.

As illustrated in Figure 9, we propose a set of high-level models for the preceding specific services. We are able to create, schedule, preempt, and delete hardware tasks onto a distant DRA. All these tasks execute and communicate with the other local or distant tasks indifferently. At this first level, the specific properties of the hardware implementation remain abstract and the scheduler only considers the current free area to take scheduling decisions. At lower levels of abstraction, the services implementation directly depends on the properties provided by the DRA model in the hardware modeling layer as described in the next section.

5. Abstract Models of the Reconfigurable Platform

During the refinement steps of the methodology, we need to refine some elements of the design, as the Dynamically Reconfigurable Area, and the processors for software tasks. This implies to integrate more detailed elements as ISS for processors and also a detailed DRA model referred as a CSS (Configuration Set Simulator). These refined models allow to automatically annotate software and hardware tasks timing and to analyze more accurately their behavior during execution.

5.1. Reconfigurable Architectures Modeling. Reconfigurable modeling is a well known issue and has been addressed for example by Becker in [45] for 1D partial regions.

In the OveRSoC project, the DRA model is composed of both an *active* and a *reactive* component. *Active component* models the hardware physical architecture. It encapsulates the constraints of the physical circuits. It corresponds to the internal organization of the DRA and ensures the execution of hardware tasks. The *Reactive component* models the dynamic behavior of the architecture. It represents the API of the DRA which provides several OS services and attributes through a fixed logical interface. In the OveRSoC project this component constitutes the interface between the external OS model and the DRA model.

These two components represent the reconfigurable hardware unit and must support the exploration strategy and the refinement of all manipulated objects. To ensure the exploration process of OveRSoC and keep complexity under control, the DRA is defined through a multilevel model.

Both active and reactive components are tightly coupled and the refinement of each impacts the other. The exploration process of the *active* and *reactive* parts of the DRA is constrained by the level of description of each component.

Three levels of abstraction for each component are proposed (see Figure 10). The refinement process applied to the DRA consists in successively defining the three proposed levels and their properties.

In the model, the *level 1* corresponds to an annotated node (Section 3.3). The different components are modeled

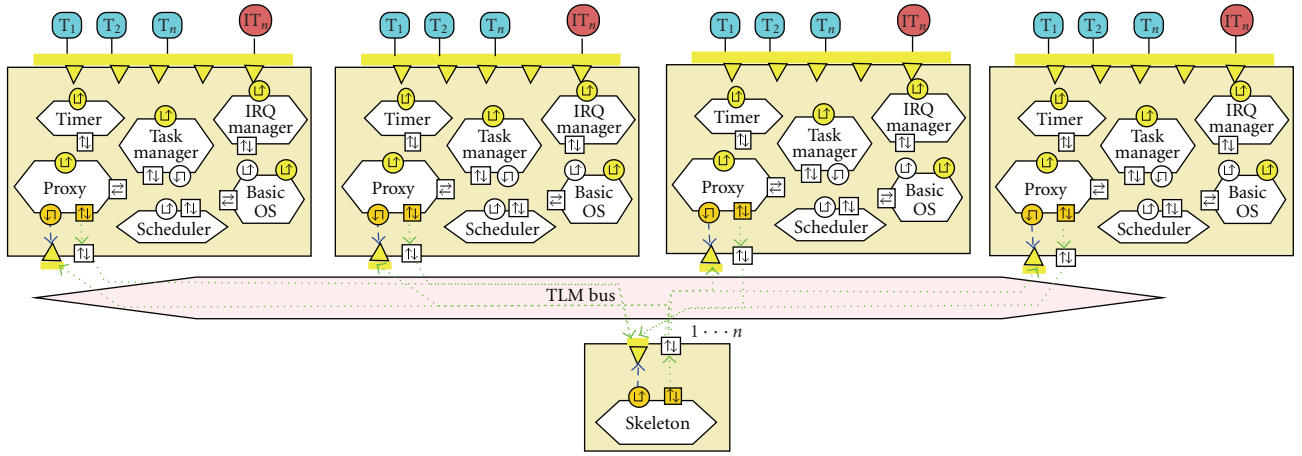


Figure 8: Model of MPSoC RTOSes with a hardware shared semaphore service. Each RTOS has a local *Proxy* service which forwards a (semaphore) request to an external device (the *skeleton*) that processes the real service, as a RPC (Remote Procedure Call), except the skeleton services could be refined in hardware.

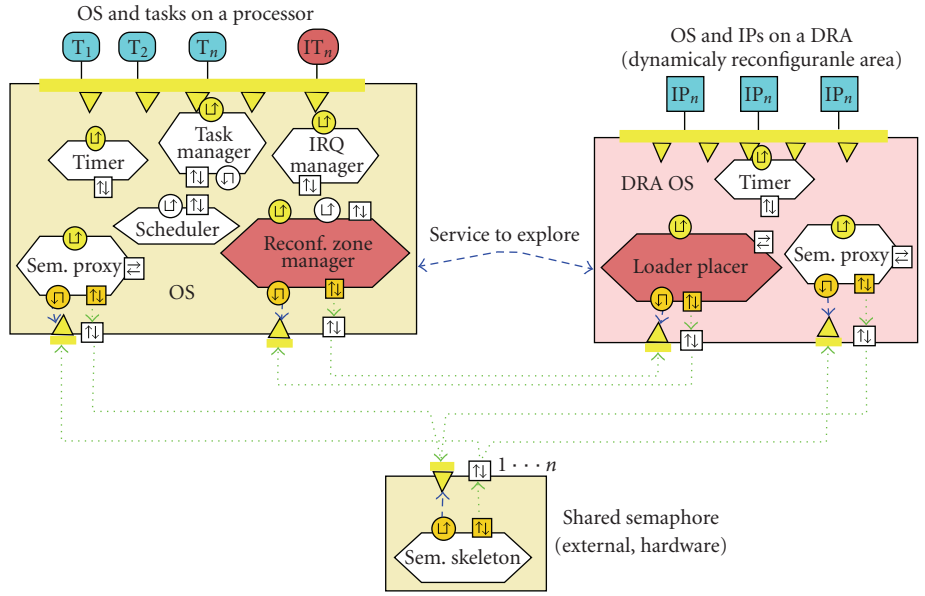


Figure 9: Model of RSoC specific OS: one standard customized with a DRA manager, one specific into the DRA, and another one specific for a refined shared semaphore service alone as an external device.

through a small number of parameters and permit a fast and coarse evaluation of methods and performances. The *active component* is considered as an homogeneous unconstrained rectangular area with a reconfiguration memory. The only parameter which is required to execute the tasks in the DRA is the task's area. At this level, the resources of the DRA are considered as unconstrained, that is, no bandwidth limitation, no latency, no area constraints, and so forth. In terms of performance, the designer evaluates the global area required in the active components, as well as the reconfiguration overhead introduced by its task management services.

The second level refines the *active components* defined as a rectangle which contains a set of heterogeneous resources such as memory, abstract running blocks and interconnect resources with limited bandwidth. The task heterogeneity is present at this level and a minimal placement service is required. At this level, the *reactive component* uses the structural information of the *active component* to verify the constraints of tasks. The corresponding definition of tasks must be completed by parameters, such as the rectangular size, the form factor of the area and so forth.

The *level 3* is the most accurate level of description and all the elementary blocks of the *active components* are described. They are defined as an array of LUT (Look Up Table) with glue logic for arithmetic computation and the corresponding sequential elements, a set of memory allocated throughout the array, columns of hardwired blocks and eventually

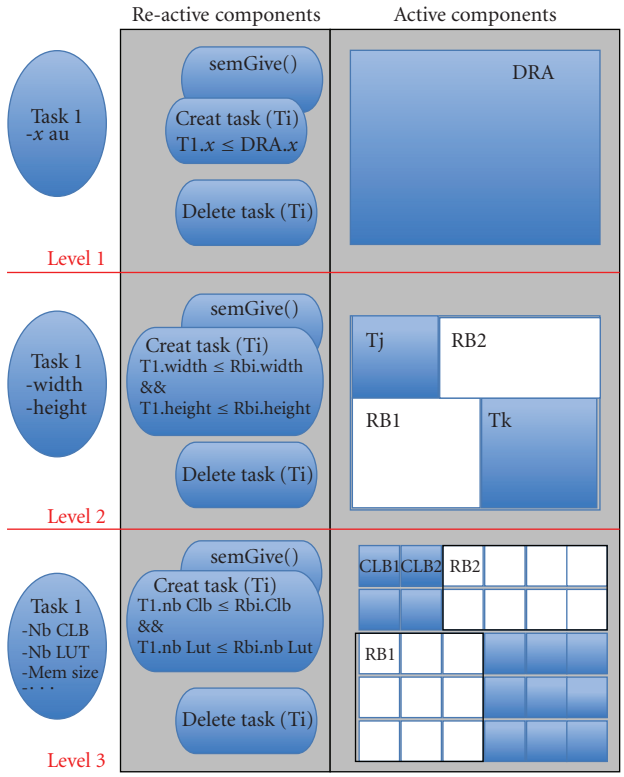


Figure 10: Hierarchical model of the active and reactive components of the DRA. The different levels permit to represents the DRA with more or less details. Refinement process leads to the complete definition of the internal architecture of the DRA. Belonging to the refinement of architectural aspects (*active* component), the supported services can be developed and evaluated (*reactive* component).

hardware core processors like PowerPCs in last Xilinx's technology. The corresponding *reactive component* must implement all the services described in Section 2. These services take both the application constraints and the precise circuit organization into account.

From this model, the DRA management can be explored through the implementation of distributed OS services.

For example, we present a particular implementation of the `createTask` OS service in Figure 11. In this example, a placer and a loader service are also implemented in the DRA. The first sequence of Figure 11 shows the hardware task creation call, `createTask(T3h)`. This OS call is performed by the software task `T1` and is handled by a processor OS service. Since the task to create must be executed onto the DRA, the OS service call is passed to the DRA through the interface, `createHWTask(T3h)`. This interface, implemented by the *reactive component*, calls the DRA OS service of task creation. Before loading the task, the DRA must verify if this new task can be loaded and placed in the reconfigurable area. To do that, the hardware OS calls the placer service, `isLoadable(T3h)`. At this step, the verification depends on the level of DRA description. For example, at level one, the placer checks if the available area is sufficient for this new task. In this case, we can model this verification as

$$\sum_{i=1}^{Nt} A_i + A_{newtask} \leq totalArea, \quad (1)$$

where A_i is the necessary area for the task i, $A_{newtask}$ is the area of the new task to instantiate onto the DRA, Nt represents the number of tasks already instantiated within the DRA, and $totalArea$ the total DRA area.

In the first part of this diagram, we illustrated the case where the placement of a new task is possible. In this case, the placer calls the loader service, `loadTask(T3h)`. The loader ensures the loading of the task bitstream, `loadBistream(T3h)`, and finally starts the task, `start(T3h)`. This sequence can be modified in order to evaluate potential overhead of different implementation solutions.

In the second part of this sequence the *CreateTask* OS call is performed by the software task T2, `createTask(T4h)`. The beginning of the sequence is the same as the first sequence presented above, but in this case, we consider that the placement of the new task into the DRA is not possible, ie. the return value for the `isLoadable(T4h)` function is No OK due to unavailable area. In this case, the task execution is refused by the DRA, and an error signal is returned. To finish this example, we suppose that a software version of task `T4h` exists and the system decides to switch to the software version, `create(T4s)`, and to schedule it immediately.

5.2. Processor Modeling. In this work, we use ISS for software simulation. As a proof of concept of our embedded software modeling approach, we developed a SystemC ISS corresponding to the ATMEL AVR Instruction Set Architecture. Targeting either hardcore processors or ISS follows the same compilation flow. We can thus reuse standard compilation tools. The binary code must then be loaded into SystemC memory models by external modules (bootstrap). The ISS communicates with memory through standard hierarchical channels. At this level of the model framework, communications can be refined towards Register Transfer Level. The ISS fetches instructions and simulates opcode execution. We implemented two modes of operation for the ISS: accurate and fast mode. When functioning in its main (accurate) mode the ISS classically extracts, executes 16-bits opcodes and increments the program counter. Once a basic block, has been executed, the ISS keeps track of the simulated execution time into specific tables to minimize the simulation overhead. Each basic block is thus associated with a block ID which corresponds either to an entire software task code or to instruction blocks within the task code. The ISS can also be interrupted and can thus model task preemption at a very fine level. In fast mode, preemption is also possible but at a coarser level since simulated time advances with a basic block precision. Interrupts can not occur before the end of the single SystemC `wait` time parameter. Once interrupted, the remaining time is saved in tables and reused when the basic block is started again.

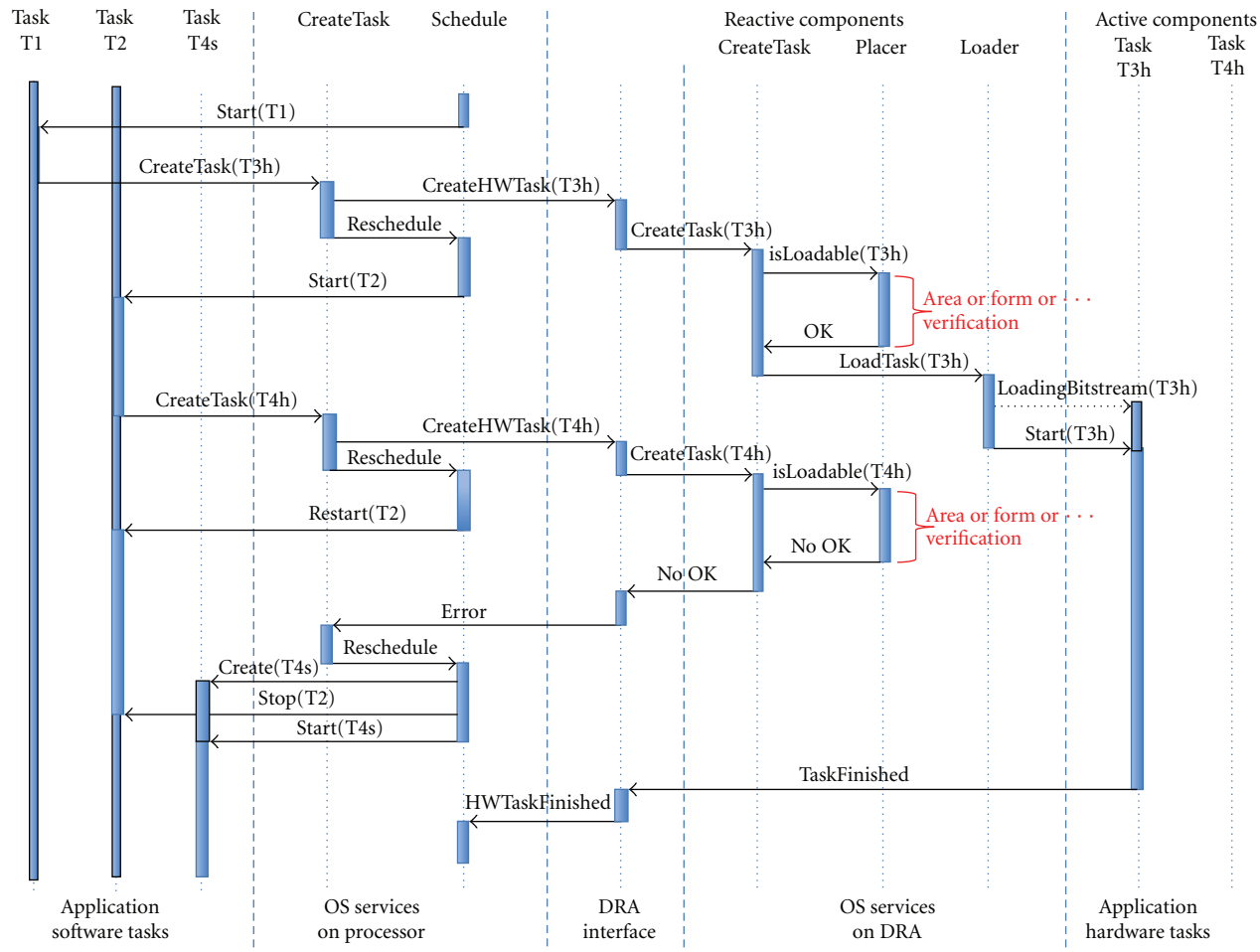


FIGURE 11: Sequence diagram of the *CreateTask* service implementation. After a Create task system call a sequence of system call depends on the services implemented on the DRA. Here we can evaluate and develop the loader and the placer services of the DRA dedicated RTOS.

Since components within each layer can be described at different levels of abstraction, the challenge is therefore to synchronize the functional and timed simulation across the layers. This is particularly difficult for the software models that exist at three different layers simultaneously: the draft application specification is modeled as C functions in the Application layer, RTOS services as SystemC transactions in the Concurrency layer, and advanced version software as instruction-accurate (compiled) descriptions in the Architecture layer. Thanks to the adopted separation of concerns approach, functional (Application layer) and timed (Architecture layer) aspects can be separated. Functional and timed aspects are thus limited to the corresponding layers. Consequently, a cycle-accurate software description has its high-level functional equivalent inside the top layer. Here, the duplication of the application description follows and reinforces the separation of concerns. It eases embedded software design by allowing software IP reuse, simulation of code portions with heterogeneous development levels, and RTOS services exploration. Furthermore, the method can be equally applied to hardware implementation of the application tasks since the Application layer makes no assumption about the hardware/software partitioning. This co-existence of the task description and its implementation version is referred as a Simulation Couple (SC) in our framework. Thus coherent execution of the SC only depends on a common definition of synchronization points. Those correspond to the RTOS system calls present both in the high-level code and in the binary code. So the granularity of the Basic Blocks (BB) for the ISS is defined as the sequences of instructions between two system calls. Each task is associated a SystemC process and a synchronization event managed by the RTOS model and shared by all the BB of the task.

As depicted in Figure 12 the scheduler launches the highest priority ready task by notifying its synchronization event. The corresponding process is activated and its functional code executes in the top layer in zero simulation time till encountering a call to the RTOS API. The RTOS service first uses the HAL API and delegates the execution time evaluation of this BB to the PE. Without interrupt, the PE estimates the duration of the BB and advances the simulation time. If an interrupt occurs in the middle of a BB, the ISS stops at the corresponding date and saves task

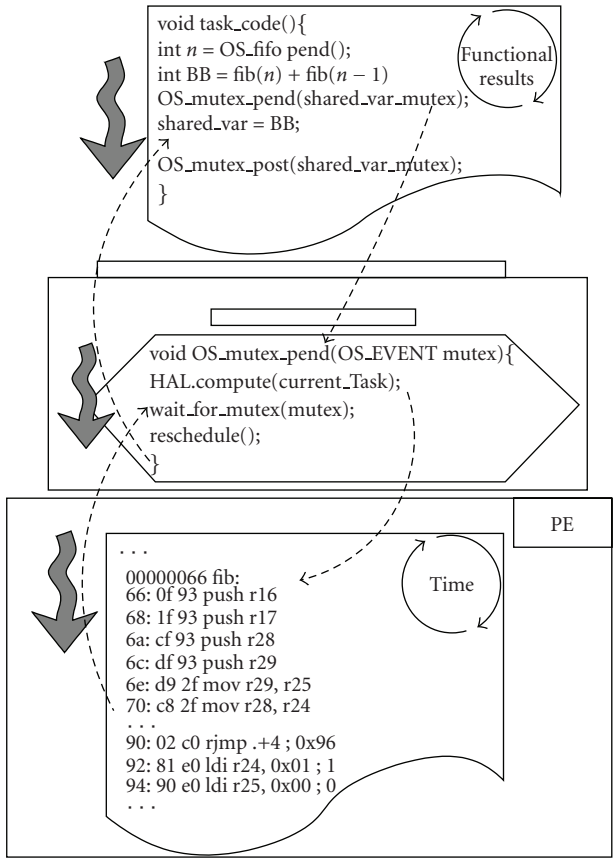


Figure 12: Example of a Simulation Couple. The software part of the application has two representations: a functionnal one used in high level abstraction layer and a timed one based on the use of an ISS.

context. The interrupt is then processed and the related routine is executed in the Concurrency layer. When the scheduler is reactivated, it can decide (according to the chosen scheduling policy) to resume the task or to elect a new one. The same scenario is repeated again until the end of the simulation.

6. Experiments

We applied our framework to a realistic application in the field of image processing for robotic vision. The application (see Figure 13) is used to learn object views or landscapes and extracts local visual features from the neighborhood of image's keypoints.

We specified at the Application layer the application as a set of 30 different communicating tasks and some of them could be run 400 times dynamically in parallel depending on the entry data as depicted in Figure 14. The full description of our application is out of the scope of this paper [46]. However, following a biologically inspired approach, this vision architecture belongs to a larger sensorimotor loop that brings interesting dynamical properties: the degree of parallelism and the execution time varies according to input data, namely the number of interest points, and the robot speed mode (high, intermediate, and low detail mode).

6.1. Software Exploration. In this context, we performed the profiling of the entire application on a hardware SoC platform. We also built the profile of the μC/OS-II [47] services (deterministic). For the purpose of the exploration we have targeted a Nios-II [48] based multiprocessor architecture (MPSoC) prototyped onto an Altera Cyclone-II FPGA circuit. The profiling of embedded software is a long and rigorous work which needs a non-preemptive and non-intrusive measurement technique. For this purpose, we modified the source code of the RTOS in order to provide such a measure technique both for the application basic blocks and for the OS services. After several executions onto a set of representative images, we built a timing data base for this application. For a simulation purpose, assigning a unique and representative execution time to the application tasks is a complex problem when the variance of the measured values is important. According to the refinement layers presented in Section 3.2, we currently recommend the use of an average value as a first approximation of the execution time and a stochastic draw into the timing data-base as a better estimation. Then, these timing data must be back-annotated into the high-level model in order to explore and evaluate the architecture dimensioning and the implementation strategies: tasks distribution, services distribution, scheduling algorithms, and so forth.

At this step, the application and the soft RTOS services were fully annotated into the Architecture layer. Following the design flow presented in Figure 1, we then performed a first set of simulations in order to evaluate the critical parts of the application when partitioned onto several processors. During these simulations the SystemC models related to the Architecture layer estimate the global system execution times. Figure 15(a) summarizes this information. Each plot represents an average value of the system performance for different images (number of keypoints). We can see that a pure software application could not be more accelerated using more than three processors (only a small gain between two and three). This MPSoC implementation reaches a global execution time of $\approx$27000 ms. Moreover we identified that the gaussian pyramid [46] represents the critical part of the application. So, we then explored the implementation of the related tasks into hardware in a reconfigurable device.

6.2. Heterogeneous Exploration Based on System Metrics. We deployed our application using a static partitioning between software and hardware tasks (more details can be found in [46]). The result of the partitioning is a set of 12 software tasks and a set of 18 hardware regular treatments. We realized the design of the hardware blocks in VHDL and back annotated the synthesis results (number of slices, execution times, communication latencies and configuration times) into the functional DRA model. The acceleration of

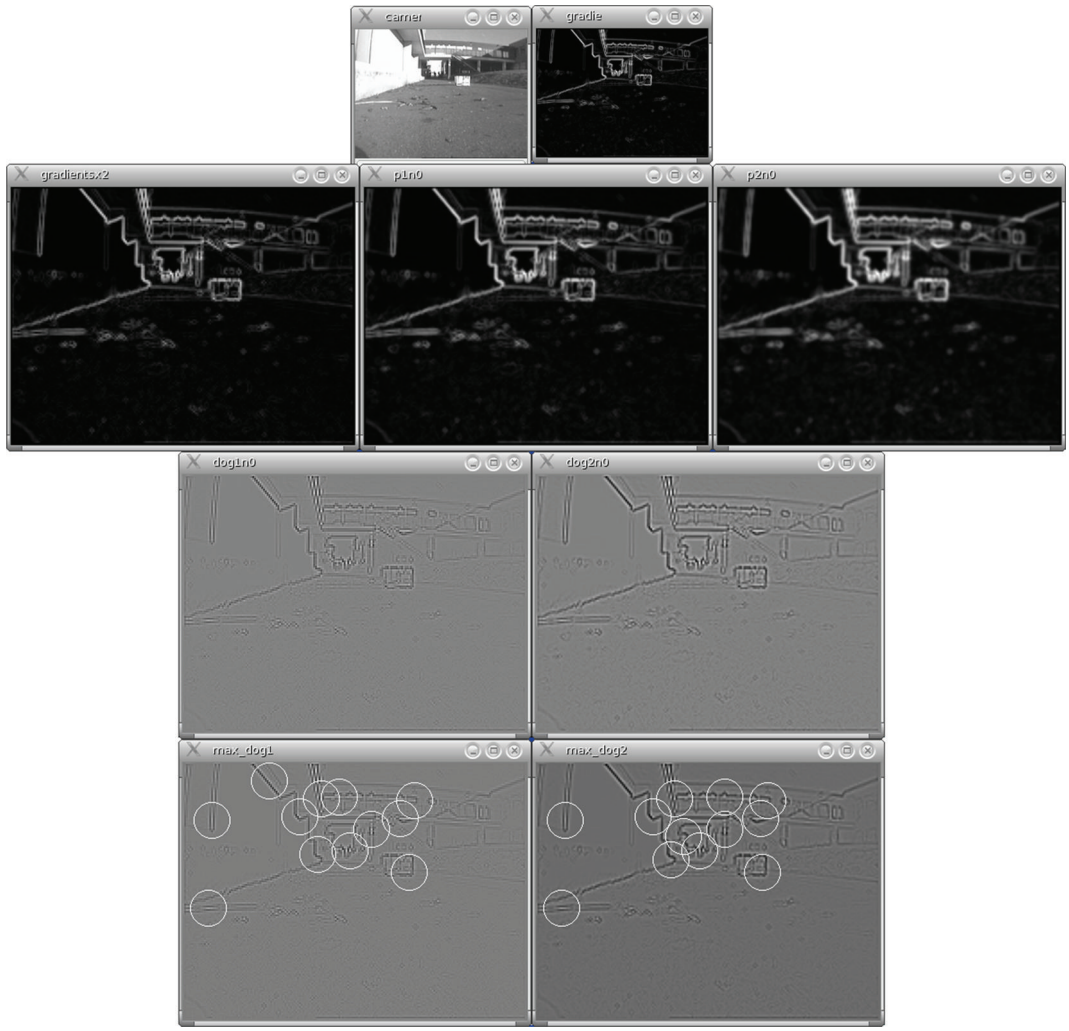


Figure 13: Graphical results of the SystemC functional model simulation of the robotic vision application.

a hardware implementation for the critical software tasks is very important: their total execution time is divided by a factor 4000. A second iteration of simulations (upper loop of Figure 1) was processed in order to define the new adequate architecture.

To figure out the right number of processors, we performed a new set of experimentations, as shown on Figure 15(b). The result of the second exploration is an architecture composed of 3 processors and a DRA with a yet undefined size. Indeed, the gain obtained by the hardware implementation of the gaussian pyramid permits to parallelize the 12 remaining software tasks to have a significant gain.

During this exploration/refinement process, the designer can use the system metrics presented in Section 3.1 and automatically extracted by the tool. Some examples of metrics used for the system dimensioning are the Gantt chart, the DRA chart (Figure 16) and the Communication chart depicted in Figure 17. The Gantt diagram represents the state of each task (software or hardware) along time: ready, running, waiting states and a configuring state for hardware tasks only. The Gantt charts of Figure 16(a) depict the new configuration of the system architecture. The 12 upper lines represent the ordering of the software tasks onto the 3 processors and the remaining lines represent the 18 hardware tasks running in parallel in the DRA. This architecture corresponds to the best achievable performances since the size of the DRA has been computed as the sum of the hardware tasks occupation. More precisely, the hardware partition uses near to 1200 slices (In the Virtex-5 FPGA slices are organized differently from previous generations. Each Virtex-5 FPGA slice contains four LUTs and four flip-flops -previously it was two LUTs and two flip-flops-), 14 BRAM and 12 DSP48 blocks. Hence, the estimated resource utilization for the global architecture (DRA + three processors) is about 4375 slices, 21 BRAM and 16 DSP48 blocks. This estimation would correspond for example to the size of a LX30T Virtex 5 circuit [49]. The global system latency ranging from 950 ms (Gantt of Figure 16) to about 60 ms depending of the application mode. We obtain about x28 acceleration compared to the pure software implementation.

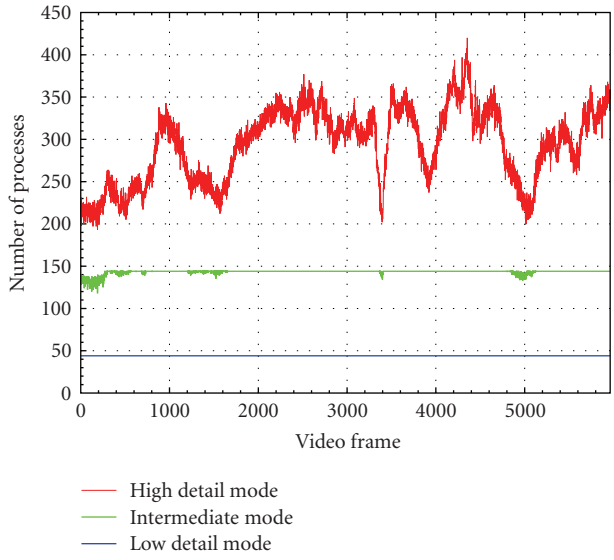


Figure 14: Number of processes created and managed by the OS model during the application simulation on a set of 6000 images for each modes.

6.3. Reconfiguration Management. In order to reduce the size of the hardware partition we vary the number of slices of the DRA and evaluated the capability of the system to adapt the hardware scheduling to a restricted area. In Figure 16, we present the results for one of the explored restricted architecture. We observe on the Gantt chart a different schedule of the hardware IP depending on the occupation rate of the DRA. The comparison between DRA charts of Figures 16(c) and 16(d) shows a clear difference in the utilization of the DRA over the time.

In the first case (Figures 16(a) and 16(c)), the DRA is never full and the tasks are configured as soon as the RTOS puts them in the Ready State. Here, the configuration only depends on the data dependencies in the application graph.

In the second case (Figures 16(b) and 16(d)) we consider a smaller DRA composed of 3000 slices. The DRA can not configure all the tasks at the same time. Here configuration depends both on the data dependencies and on the available resources. At level 1 of the DRA model, the hardware scheduler only manages available resources. It searches for sleeping tasks within the DRA to be replaced by a new task asking for resources. Besides, once a hardware task finishes its execution, it is removed (its resources are freed), enabling another task to be implemented. For the estimation of the configuration time we used a metric which depends on the size of the partial bitstream for the targeted DRA technology (about 50 μs per block of 16 CLBs on a Virtex 5).

As a first conclusion the exploration of the architecture for the robotic vision application leads us to model a complete RSoC platform at a high-level of abstraction. This high-level model focuses on the definition of the RTOS services needed by the identified architectures. For the systems presented in this section, we used as many OS as processors. All these components (Figure 9) are composed of the following services:

(i) a task management service to dynamically create keypoints extraction tasks,

(ii) several shared semaphores and mutex to synchronize the application and to protect image data into the shared memories,

(iii) a priority based scheduler on each processor,

(iv) a time management service for timeouts,

(v) an interrupt manager for the management of the multiprocessor architecture.

Also, another RTOS model is dedicated to the management of hardware tasks. This RTOS model provides several additional services:

(i) at level 1 of the DRA, a specific scheduling service using only the available resources,

(ii) at level 2 of the DRA, a refined scheduling service using also the localization and the shape of the tasks,

(iii) a placement service related to the level of the DRA model,

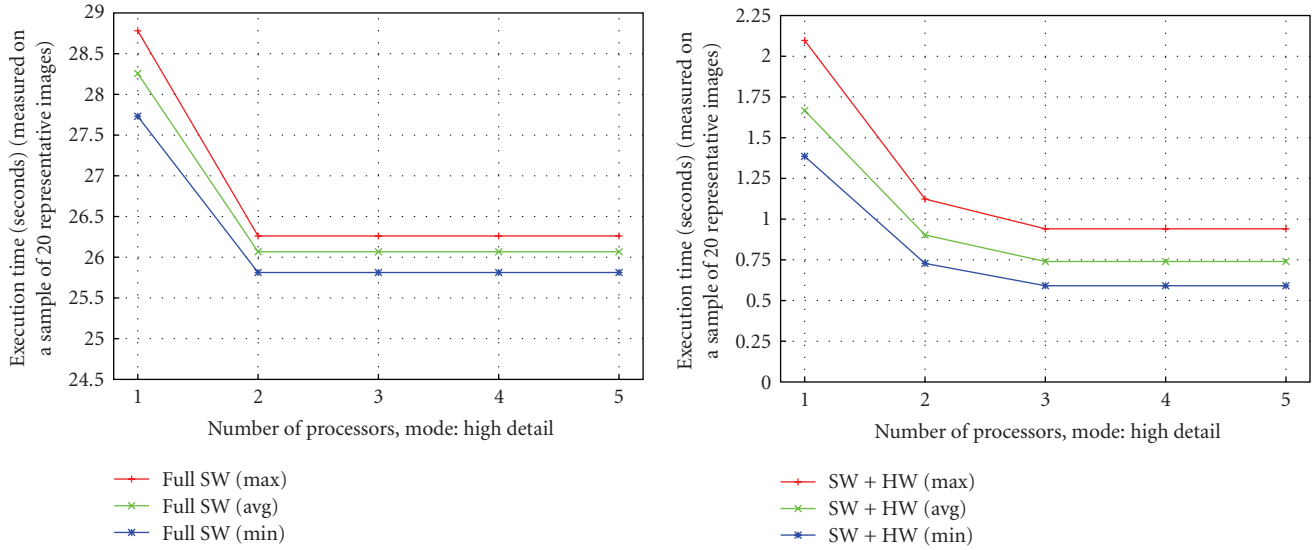
(iv) a communication service using hardware FIFO (results are presented on Figure 17),

(v) several mutex and semaphore proxies for the synchronization with software tasks.

The refinement of the DRA to level 3 allows to test low-level hardware scheduling and placement strategies. We have implemented two simple placement algorithms to manage the DRA resources at a finer grain.

6.4. Accuracy and Simulation Overhead of the Model. To evaluate the efficiency of our modeling approach, we performed two sets of experiments. First, we evaluated the model accuracy and compared the simulated execution time relative to actual board measurements for multiple implementations. The average application times measured on board is 2926 ms and the simulated time gives 2836 ms. Those results validate our high level model considering the simulation's accuracy is within 3-4% of board measurements.

Then we evaluated the simulation time of the application on top of our RTOS model in comparison with a purely functional description. The deployment of the application tasks was explored and simulated using the Application and Concurrency layers of Figure 2. We vary the number of PEs within the architecture from 1 to 6 OS (Processors or DRA). Tasks execute and communicate in the same way on board and in simulation trough a single shared memory space protected with shared semaphores. Table 2 shows the scalability of our model. It indicates the simulation time t_n



(a) Pure software tasks implementation of the application

(b) Mixed hardware and software tasks implementation

Figure 15: Performance gain exploration for several sizes of MPRSOC architectures for case (a) all tasks in software; case (b) partitioned in hardware (on a DRA) and software on multiple processors.

Table 2: Simulation overhead versus number of OS.

n	0	1	2	3	4	5	6
simulation time t_n (second)	5.5	6	7.4	8.6	9.8	11.1	12.8
overhead s_n (%)	−8.9	0	23.3	43.3	63.3	85	113.3

of a platform modeled at the Concurrency layer composed of n RTOS and the average simulation overhead $s_n = (t_n - t_1)/t_1$ for different platform sizes. t_0 represents the execution time of the pure functional application specification (at the Application layer). Simulations were realized on an Intel DualCore workstation running at 1.66 GHz with 2 GB of RAM.

For monoprocessor platforms, the RTOS model does not impact the simulation time since the overhead is only 8.9% more than the purely functional application description. Results indicate that the simulation time overhead is around 23% more per simulated RTOS. This overhead is due to the SystemC simulation kernel that works for the whole list of SystemC `sc_thread` of the system, which increases with the number of RTOS.

Finally, the framework allows to simulate an application in a functional and non-intrusive debug mode as illustrated in Figure 13.

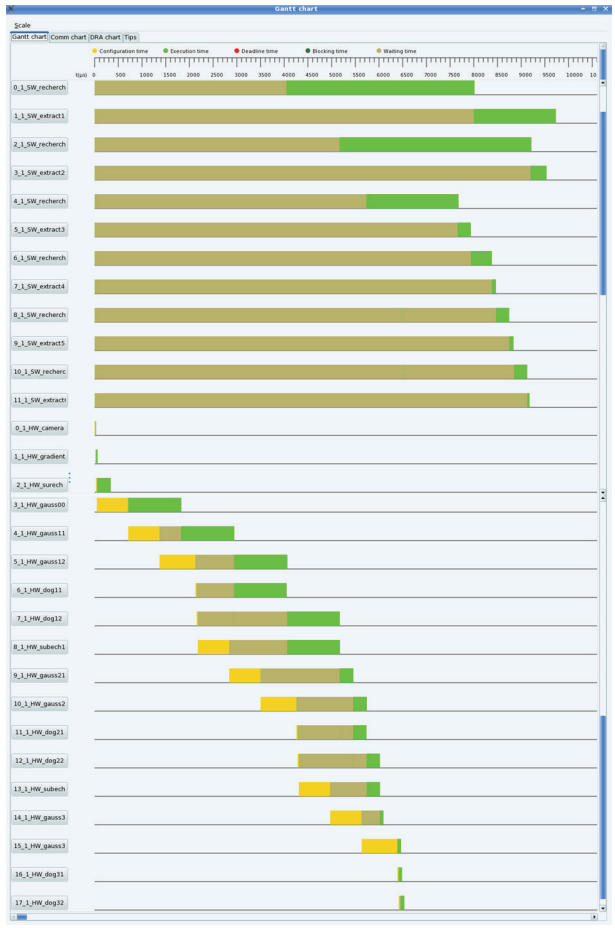
6.5. Perspectives. We are now working on the integration of all the components into a basic and scalable target architecture which is composed of one ISS, a DRA model, a shared bus, a global memory and a distributed OS. The final platform model uses the three layers presented in Figure 2 (Application, Concurrency and Architecture layers) in order to provide a good tradeoff between performance accuracy and simulation overhead. The first experiments show that going down till the cycle-accurate level of the Architecture layer (ISS and CSS models) brings a simulation overhead 500 times longer compared to a timed simulation at the Concurrency layer.

7. Conclusion and Perspectives

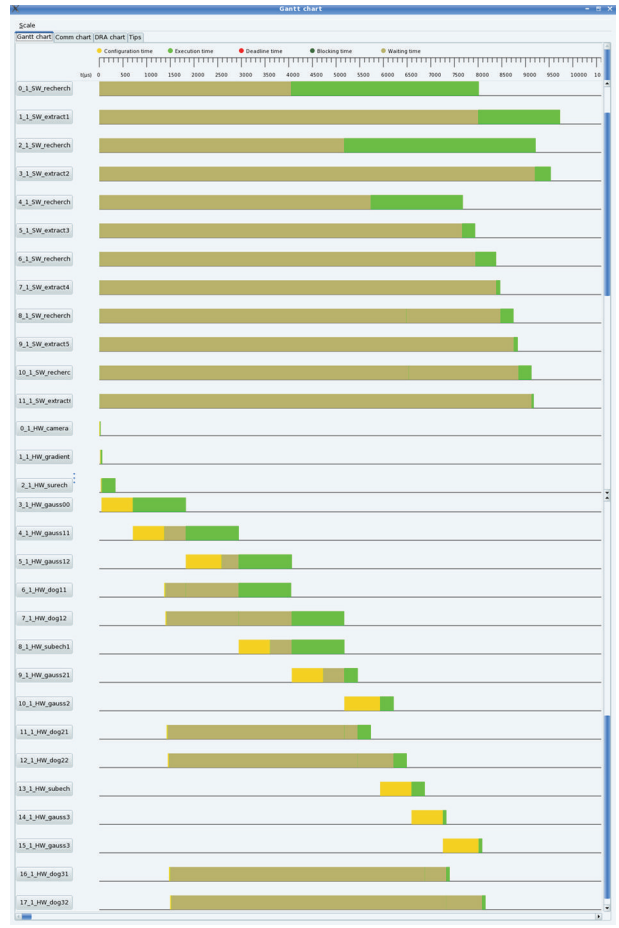
In this paper, we have presented a modeling framework for the design of a complete RSoC platform including processor(s), Dynamically Reconfigurable Architecture and OS services. The proposed design flow is based on a system level modeling approach which eases the exploration of the RTOS services distribution both onto processors and directly inside a reconfigurable region of the considered hardware unit. The main contribution of this work consists in proposing a unified modeling and refinement methodology for the software and the hardware parts of a dynamically reconfigurable system.

We have also listed the specific services that are needed in the literature for the management of the reconfigurable resources of the architecture. Thanks to a modular and flexible modeling approach we developed a library of generic components for the description of RSoC platforms. Among them, we developed basic hardware services such as hardware task management, hardware/software synchronization and bitstream management at high level of abstraction. The global method and the SystemC models were validated on an image processing application.

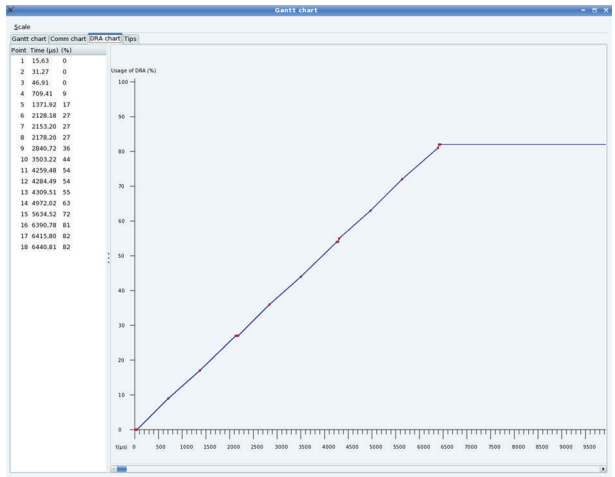
Today, the presented results show that the framework allows to define, simulate, and explore the specific services of RTOS for RSoC platforms very early in the design flow.



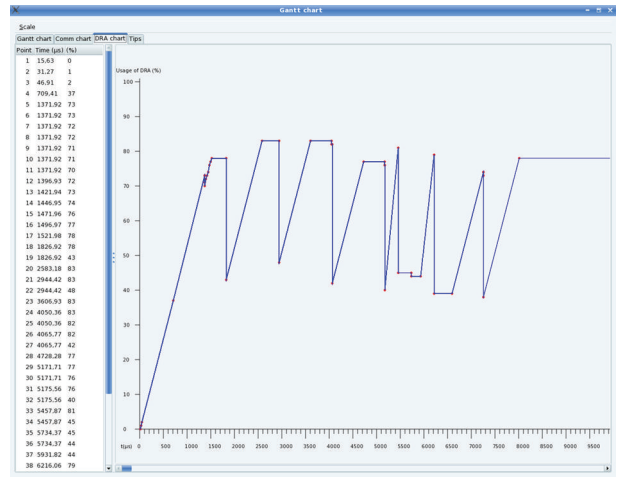
(a) Gantt Chart for large DRA



(b) Gantt Chart for smaller DRA



(c) Occupation rate of large DRA



(d) Occupation rate of smaller DRA

FIGURE 16: (a) and (b) represent the Gantt diagram for all the application tasks in both software on 3 processors and in hardware on a DRA of 4500 slices on the left and 3000 on the right. (c) and (d) represent the evolution of the DRA occupation over the time.

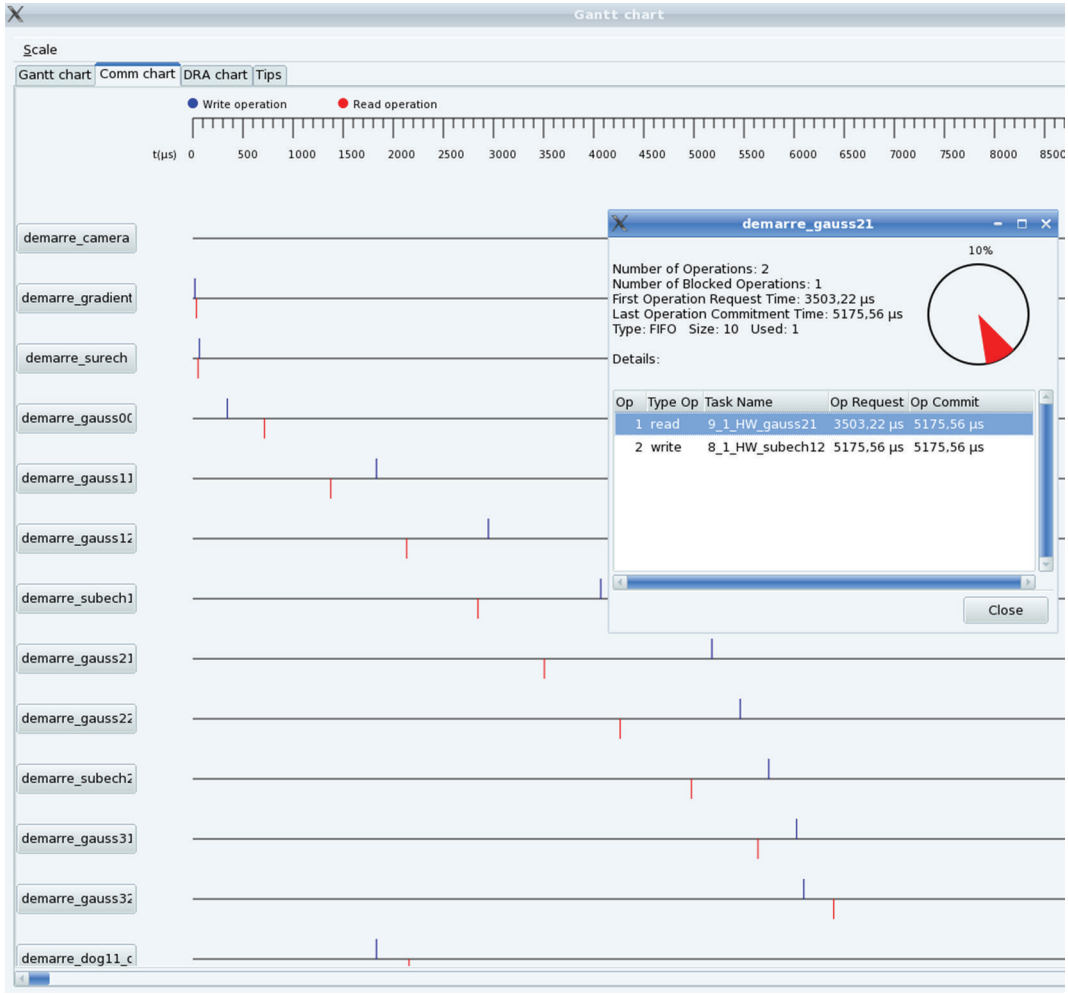


Figure 17: The DOGME tool provides several metrics helping the designer to evaluate the simulated design solutions. The window shows communications between tasks over time. It also computes the filling ratio for FIFO based communications.

Now, we have to refine some existing services such as the hardware scheduler at lower levels of abstraction in order to manage and estimate more accurately the resources used by an application on a real FPGA. We also have to extend the library of models: processing units, refined communication media and services such as placement algorithms from the literature. The OveRSoc framework could then be used as a comparison environment for upcoming methods in the context of DRA management.

Acknowledgments

We would like to thank Sylvain Viateur for his help on the ISS SystemC model. The work presented in this paper was performed in the OveRSoC project which is supported by the french ANR funding.

References

[1] O. Diessel and G. Wigley, "Opportunities for operating systems research in reconfigurable computing," Technical Report ACRC-99-018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Mawson Lakes, South Australia, 1999.

[2] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 7, April 2003.

[3] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[4] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial FPGA exploitation," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, 2007.

[5] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconflgurable architectures," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, pp. 366–371, Munich, Germany, March 2005.

[6] M. Yuan, X. He, and Z. Gu, "Hardware/software partitioning and static task scheduling on runtime reconfigurable FPGAs using a SMT solver," in *Proceedings of the 14th IEEE*

Real-Time and Embedded Technology and Applications Symposium (RTAS '08), pp. 295–304, St. Louis, Mo, USA, April 2008.

[7] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[8] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," in *Proceedings of Australasian Conference on Computer Systems Architecture (ACSAC '01)*, pp. 130–137, IEEE Computer Society, 2001.

[9] F. Engel, I. Kuz, S. Petters, and S. Ruocco, "Operating systems on SoCs: a good idea?" in *Proceedings of IEEE Embedded Real-Time Systems Implementation Workshop (ERTSI '04)*, December 2004.

[10] I. Benkhermi, M. E. A. Benkhelifa, D. Chillet, S. Pillement, J.-C. Prevotet, and F. Verdier, "System-level modelling for reconfigurable SoCs," in *Proceedings of the 20th Conference on Design of Circuits and Integrated Systems (DCIS '05)*, Lisboa, Portugal, November 2005.

[11] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[12] A. Kuhn, F. Madlener, and S. Huss, "Resource management for dynamic reconfigurable hardware structures," in *Proceedings of Reconfigurable Communication Centric System-on-Chips (ReCoSoC '06)*, 2006.

[13] J. C. Van Der Veen, S. P. Fekete, M. Majer, et al., "Defragmenting the module layout of a partially reconfigurable device," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 92–101, Las Vegas, Nev, USA, June 2005.

[14] K. Puma and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for re-configurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, 1999.

[15] J. Resano, D. Mozos, D. Verkest, F. Catthoor, and S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*, pp. 119–124, San Diego, Calif, USA, 2004.

[16] L. Levinson, R. Manner, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," in *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, pp. 301–302, 2000.

[17] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 121–130, Springer, Berlin, Germany, 2000.

[18] D. Koch, A. Ahmadinia, C. Bobda, and H. Kalte, "FPGA architecture extensions for preemptive multitasking and hardware defragmentation," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT '04)*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 433–436, Brisbane, Australia, December 2004.

[19] T. Marescox, A. Bartic, B. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection networks enable fine-grain dynamic multitasking on FPGAs," in *Proceedings of the 12th Field Programmable Logic and Applications Conference*, vol. 2438, pp. 795–804, September 2002.

[20] Symbad, "SYMBAD—Formal Verification in SYsteM Level Based Design," 2002, http://www.setnet.org/Research/SYMBAD.htm.

[21] M. Borgatti, A. Capello, U. Rossi, et al., "An integrated design and verification methodology for reconfigurable multimedia systems," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, pp. 266–271, Munich, Germany, March 2005.

[22] P. Hsiung, C. Liao, C. Tseng, S. Lin, Y. Chen, and K. Chiu, "Hardware-software codesign and coverification methodology for dynamically reconfigurable system-on-chips," in *Proceedings of Workshop on Object-Oriented Technology and Applications (OOTA '04)*, 2004.

[23] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-level modeling of dynamically reconfigurable co-processors," in *Field Programmable Logic and Application*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 881–885, Springer, Berlin, Germany, 2004.

[24] P.-A. Hsiung, C.-H. Huang, and C.-F. Liao, "Perfecto: a systemc-based performance evaluation framework for dynamically partially reconfigurable systems," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL '06)*, Lecture Notes in Computer Science, pp. 190–195, Madrid, Spain, August 2006.

[25] A. Raabe, P. A. Hartmann, and J. K. Anlauf, "ReChannel: describing and simulating reconfigurable hardware in systemC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 112–120, 2008.

[26] K. Benkrid, A. Benkrid, and S. Belkacemi, "Efficient FPGA hardware development: a multi-language approach," *Journal of Systems Architecture*, vol. 53, no. 4, pp. 184–209, 2007.

[27] A. Herrholz, F. Oppenheimer, P. A. Hartmann, et al., "The ANDRES project: analysis and design of run-time reconfigurable, heterogeneous systems," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 396–401, August 2007.

[28] J. J. Lee and V. J. Mooney III, "Hardware/software partitioning of operating systems: focus on deadlock detection and avoidance," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 167–182, 2005.

[29] G. Smit, E. Schüler, J. Becker, J. Quévremont, and W. Brugger, "Overview of the 4S project," in *Proceedings of International Symposium on System-on-Chip*, pp. 70–73, Tampere, Finland, November 2005.

[30] OSCI, "IEEE 1666^{TM} Standard SystemC Language," http://www.systemc.org/.

[31] E. A. Lee and S. Neuendorffer, "Concurrent models of computation for embedded software," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 239–250, 2005.

[32] K. Keutzer, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000.

[33] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 45–51, Newport Beach, Calif, USA, October 2003.

[34] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. A. Jerraya, "Flexible and executable hardware/software interface modeling for multiprocessor SoC design using systemC," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '07)*, pp. 390–395, IEEE Computer Society, Washington, DC, USA, 2007.

[35] A. Donlin, "Transaction level modeling: flows and use models," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (CODES+ISSS '04)*, pp. 75–80, Stockholm, sweden, 2004.

[36] OCP-IP, "Open core protocol international partnership," http://www.ocpip.org/.

[37] E. Huck, B. Miramond, and F. Verdier, "A modular systemC RTOS model for embedded services explorations," in *Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP '07)*, Grenoble, France, 2007.

[38] B. Miramond, F. Verdier, and M. Aichouch, "DOGME distributed operating system graphical modeling environment," http://oversoc.ensea.fr/ oversoc-graphical-modeling-environment-1.

[39] "Eclipse rich client platform," http://eclipsercp.org/.

[40] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," in *Proceedings of Design Automation Conference*, pp. 396–401, 2000.

[41] P. Hastono, S. Klaus, and S. A. Huss, "Real-time operating system services for realistic systemc simulation models of embedded systems," in *Proceedings of Forum on Specification and Design Languages (FDL '04)*, pp. 380–392, Lille, France, September 2004.

[42] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: a POSIX model," *Design Automation for Embedded Systems*, vol. 10, no. 4, pp. 209–227, 2005.

[43] Z. He, A. Mok, and C. Peng, "Timed RTOS modeling for embedded system design," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 448–457, San Francisco, Calif, USA, March 2005.

[44] P.-A. Hsiung, C.-H. Huang, and Y.-H. Chen, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC," *Journal of Embedded Computing*, vol. 3, no. 1, pp. 53–62, 2009.

[45] M. Ullmann, M. Hübner, and J. Becker, "On-demand FPGA run-time system for flexible and dynamical reconfiguration," *International Journal of Engineering Simulation*, vol. 1, no. 3-4, pp. 193–204, 2005.

[46] F. Verdier, B. Miramond, M. Maillard, E. Huck, and T. Lefebvre, "Using high-level RTOS models for HW/SW embedded architecture exploration: case study on mobile robotic vision," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 349465, 17 pages, 2008.

[47] J. Labrosse, "MicroC/OS-II: the real-time kernel," CMP Media, 2002, http://www.micrium.com/page/support/bookstore.

[48] "Altera," http://www.altera.com/.

[49] Xilinx, "Virtex 5 familly overview," http://www.xilinx.com/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 973475, 11 pages
doi:10.1155/2009/973475

Research Article

Parallel Processor for 3D Recovery from Optical Flow

Jose Hugo Barron-Zambrano, Fernando Martin del Campo-Ramirez, and Miguel Arias-Estrada

Computer Science Department, National Institute of Astrophysics, Optics and Electronics, 72840 Puebla, Mexico

Correspondence should be addressed to Jose Hugo Barron-Zambrano, jhbarronz@inaoep.mx

Received 16 March 2009; Revised 10 August 2009; Accepted 8 October 2009

Recommended by Cesar Torres

3D recovery from motion has received a major effort in computer vision systems in the recent years. The main problem lies in the number of operations and memory accesses to be performed by the majority of the existing techniques when translated to hardware or software implementations. This paper proposes a parallel processor for 3D recovery from optical flow. Its main feature is the maximum reuse of data and the low number of clock cycles to calculate the optical flow, along with the precision with which 3D recovery is achieved. The results of the proposed architecture as well as those from processor synthesis are presented.

1. Introduction

3D recovery from a sequence of frames has been one of the main efforts in computational vision. Unlike the case of a single static image, the information contained in a sequence is important because it allows the motion and structure recognition of the objects in a scene. The purpose of this technique is to build a 3D model from an object, using data extracted from the analysis of a 2D representation of such object. The implementation of a depth recovery algorithm depends highly on the application to develop, and the search of higher precision in 3D recovery has led to the implementation of more complex and computational demanding algorithms [1]. The main methods for 3D recovery are stereo vision, shading, and motion estimation. Stereo algorithms perform the calibration of two or more cameras to then triangulate the computing points of the scene [2]. Other methods try to recover the depth information from the motion and intensity of elements in the images. The methods based on correspondence try to perform pairing between images features and then estimate a single motion during the image sequence, to finally apply triangulation, as the stereo algorithms do [2]. These methods present good performance, but only when there is a correspondence between features, which recover only disperse surfaces.

Other techniques propose to perform a depth estimation using operations that manipulate the changes in the intensity of the image sequence and incorporate the information in a Kalman filter [3]. The problem in correspondence and intensity methods is that neither the features nor the intensity of the images is constant or continuous, decreasing the reliance of the results.

There are also other methods that compute the depth from known objects in the scene. Other techniques for depth estimation proposed to calculate the normal component of the flow [4]. For its correct operation it is necessary to know the trajectory followed between the camera and the scenery [3]. Most of the algorithms impose restrictions as the knowledge of the camera motion, its position with respect to the scenery or complicated calibration techniques.

In the case of the optical flow approach, recovery can be obtained by a camera without knowing its parameters. Neither multiple cameras aligning nor previous knowledge of the scene or the motion is necessary. All what is needed is the relative motion between the camera and the scene to be small.

The present paper introduces an FPGA-based processor for the 3D recovery from the optical flow under a static environment, so no object performs a movement in the scene. The motion of the camera along each image of the video sequence must be short, that is, a maximum of two

pixels per frame. The processor meets the constraint that it is capable of operating in near video rate time, that is, 22 frames per second (fps) for images with *Video Graphics Array* (VGA) resolution: 640 × 480 pixels.

The paper is divided as follows. Section 2 includes an analysis of related works in the field. Section 3 describes the theoretical bases for the development of the research. The functional description and interaction of the processor blocks are discussed in Section 4. While in Sections 5 and 6, performance analysis and results of the architecture are presented. Finally, conclusions and future work are presented in Section 7.

2. Background

There have been several efforts to solve the problem of depth recovery according to the characteristics and constraints of the applications to develop. Fife and Archibald [5] report an implementation for the navigation of autonomous vehicles, using feature correspondence. For each frame in the sequence, the system locates all the identified features in the previous frame, and then update the actual position estimation in the 3D space. This implementation was done by reconfigurable computing and the use of embedded processors. The performance obtained by this implementation is 30 fps with a resolution of 320 × 240 pixels. The main disadvantage of this architecture is that it only computes Z (the depth) for specific points in the video sequence.

In [6], Diaz et al. present an FPGA implementation that uses structured light. A code simplification is performed in this work by looking for redundant operations. Moreover, fixed point arithmetic is used under a format of 14 bits for the integer part and 5 or 6 bits for the fractional one. The processing time is around 20 seconds for images with a resolution of 640 × 480 pixels.

Zach et al. [7] present a work for dense 3D recovery in stereo images. Their implementation was built as a hardware-software codesign: the hardware part of the system is based on a pairing procedure to avoid the accuracy loss due to the limited resolution in 3D processors. Once the pairing is performed, depth recovery is obtained applying epipolar geometry. The software section only performs the flow control and the information transfer. A calculation of over 130,000 depth values per second running on a standard computer is reported.

3. Theoretical Framework

3.1. Optical Flow. Optical Flow is the apparent motion of patterns of objects, surfaces, and edges caused by the relative change of position among the observer (a camera) and the scene [8, 9]. There are, in the literature, comparative works of hardware implementations for several optical flow algorithms [10–12]. Basically, the majority of the optical flow implementations are based on two types of algorithms: gradient-based algorithms and correlation-based algorithms.

The gradient-based algorithms calculate the optical flow with space-time derivatives of the intensity of the pixels in an image, or through the filtered versions of the images (using low-pass and high-pass filters) [13].

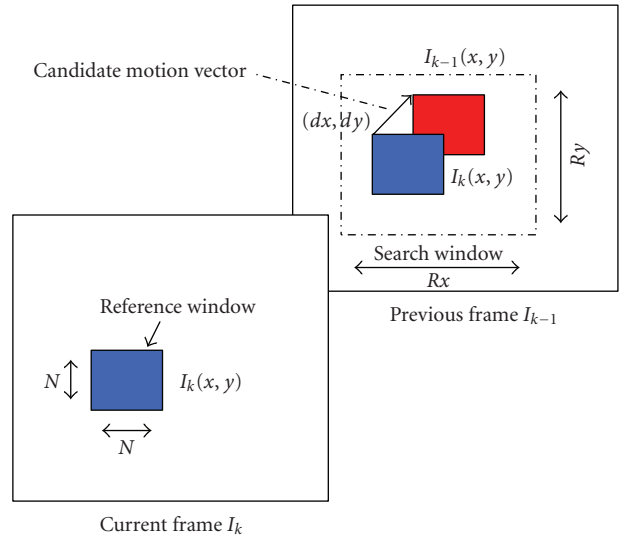


Figure 1: Basic operation of the correlation algorithms.

On the other hand, correlation-based algorithms work by comparing windows or blocks: two consecutive frames are taken from the video sequence, that are divided in periodic and equal size blocks. These blocks can present overlapping but always maintaining the same size. Given the regularity of the operations, correlation-based algorithms are better suited for hardware implementation. Figure 1 shows a graphical representation of the algorithms based on correlation.

One of the simplest correlation metrics found in the literature is the Sum of Absolute Differences (SAD) [14]. The main characteristics are its easy implementation and its reduced use of hardware resources

$$\mathrm{SAD} = \sum_{m=x}^{x+N-1} \sum_{n=y}^{y+N-1} \left| I_g(m,n) - I_{g-1}(m+dx, n+dy) \right|. \quad (1)$$

3.2. 3D Recovery from Optical Flow. This section discusses the equations that describe the relation among the depth estimation and the optical flow generated by the camera motion. The same notation found in [15] is used here. It is also assumed that the camera motion is through a static environment. The reference coordinate system is shown in Figure 2.

The coordinate system X, Y, is fixed with respect to the camera. The Z axis is located across the camera optical axis so any motion can be described by two variables: translation and rotation. $\vec{T}$ denotes the translational component of the camera, while $\vec{\omega}$ the angular velocity. Finally, the instant coordinates of the point P in the environment are (X, Y, Z) [15]. From these variables, (2) can be obtained, and from it the value of Z can be calculated

$$Z = \frac{\alpha^2 + \beta^2}{(u - u_r)\alpha + (v - v_r)\beta}, \quad (2)$$

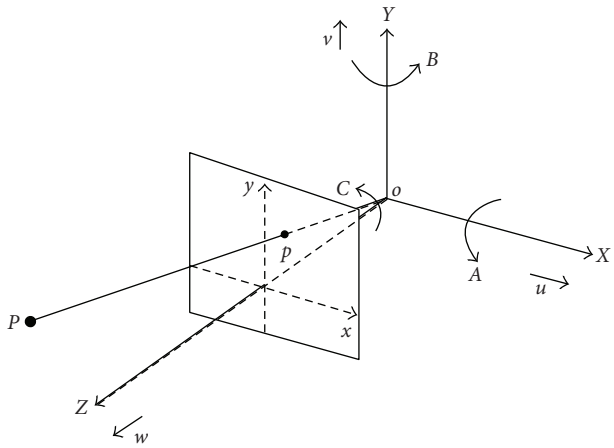


Figure 2: Reference coordinate system.

where α, β, u_r, and v_r are defined as

$$\begin{aligned} \alpha &= -U + xW, \\ \beta &= -V + yW, \\ u_r &= Axy - B(x^2 + 1) + Cy, \\ v_r &= A(y^2 + 1) - Bxy - Cx. \end{aligned} \tag{3}$$

Equation (2) calculates Z in terms of the parameters of the translation and rotation components. As there is no calibration on the camera, these parameters are still unknown. Nevertheless, their values are useful only to scale the value of Z and they do not affect the recovered structure. Therefore, it is possible to assume constant values for each of these parameters. The disadvantage of this consideration is that only a relative value of the depth information can be obtained.

4. Architecture

The proposed architecture has the purpose of recovering the 3D information from the optical flow found in a video sequence. The system presents a maximum reuse of data and is optimized for minimum hardware resources. Processing is achieved in a predictable time, that is, under real-time constrains. The architecture meets the following specifications: it works with images in 256 levels gray scale, and a resolution of 640 × 480 pixels. The image rate processing obtained 22 frames per second limited by the FPGA platform capacity, and maintaining a low *relative error* for the 3D recovery.

4.1. Description. The architecture is divided in three main functional blocks: the first one is for the calculation of the optical flow, the second one for the 3D recovery, and the last one is dedicated to data routing. The general operation of the design is as follows. The data corresponding to the pixels of the reference and search windows from a consecutive image pair in the sequence are sent to the system through a data bus.

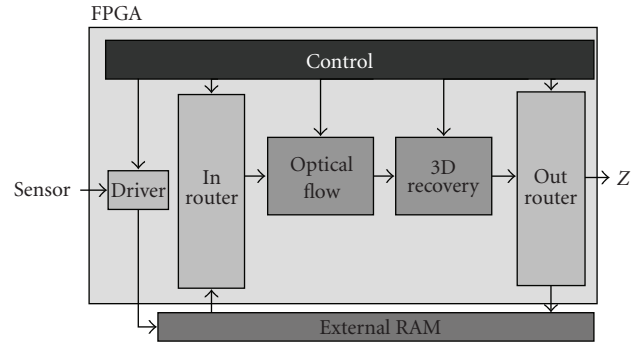


Figure 3: General block diagram of the architecture.

Subsequently, the optical flow processing is carried out. Here the motion vectors are obtained and then sent to the block for 3D recovery. Finally, the obtained values are presented to the real world, stored in external memories or sent to another process through an output data bus. The read and write addresses are generated by the routers that control the data flow. The signals that control the architectures operation are arranged in a control bus. The architecture is shown in Figure 3.

4.2. Optical Flow Module. The Optical Flow Module operates with a 4 × 4 pixels reference window and a 8 × 8 pixels search window. These values are usually used in the literature [14]. Due to the nature of (4), the Optical Flow Module can be formulated as a window-based operator considering that the coefficients of the window mask are variable and new windows are extracted from the first image to constitute the reference window. Once the processing in the search area has been completed, the window reference can be replaced with a new one, and the processing goes on the same way until all data is processed.

The number of operations per second (OPS) for the calculation of the motion estimation is given by

$$\text{OPS} = 3 * 2p * 2p * Nh * Nv * f, \tag{4}$$

where N_h and N_v are, respectively, the vertical and horizontal of the image in pixels, p is the size of the search window, and f represents the frames per second rate. For this particular work, $p = 8$, $f = 5$, $Nh = 640$, and $Nv = 480$, the result of (4) indicates a computational charge of 235,929,600 integer arithmetic operations per second. Thus, a sequential approach or the use of a general purpose processor is inadequate and insufficient for the motion estimation. The Optical Flow Module is composed by a series of basic blocks called Processing Elements (PEs). Each PE is in charge of the subtraction operations among pixels and the absolute value computation for the SAD equation. The block diagram of a PE is depicted in Figure 4.

A Window Processing Module (WPM) is assembled with 20 PEs working in parallel (Figure 4), where a set of operations are performed at the same time in a single clock cycle. The processing elements work in a systolic

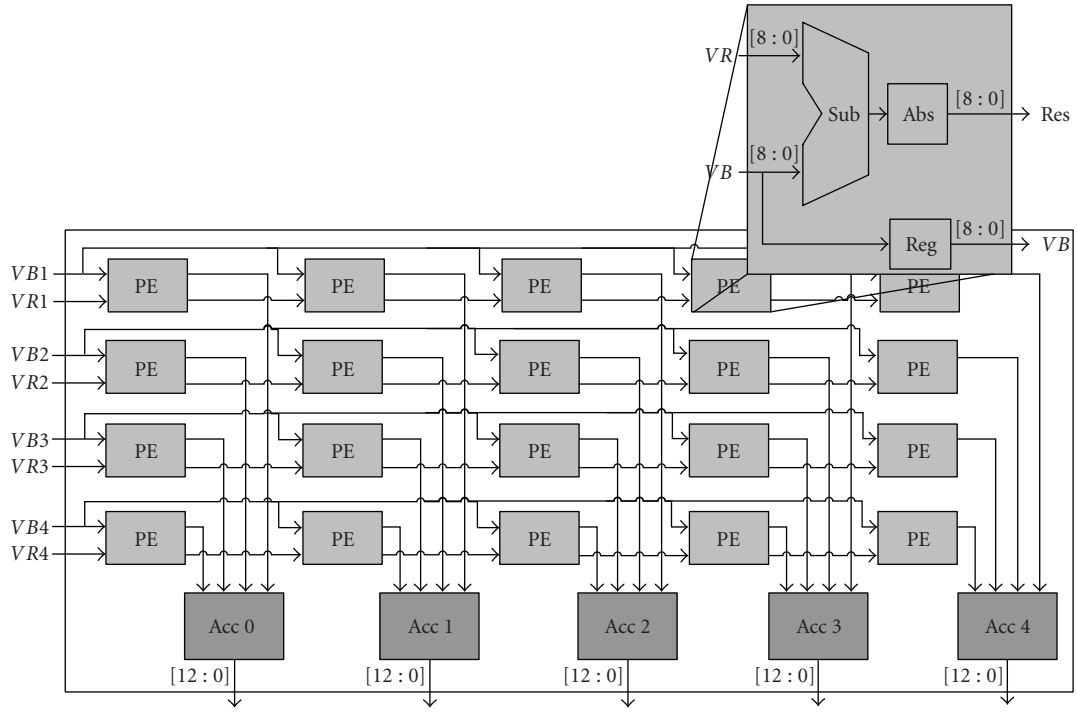


Figure 4: Block diagram of the Processing Element PE (basic building block). The Window Processing Module (WPM) integrates 20 PEs.

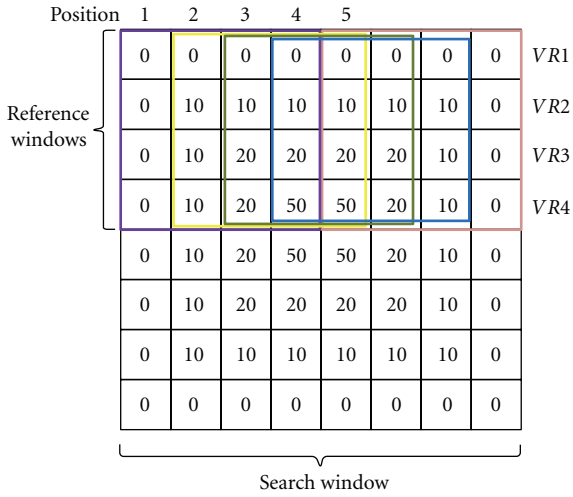


Figure 5: Computation of the correlation between the search and the reference window with a WPM.

pipeline. When the window data moves through the buffers to the next pixel location in the input image, several pixels are overlapped with the previous windows. Therefore, it is possible to use multiple instances of the WPM to compute incrementally at several consecutive pixel locations partial results of the window comparison operation. An accumulator adds together the partial results until all the data has been processed. Then, the accumulator is reset and a new window comparison operation is started. In the current implementation, the reference window is 4×4 pixels and the search window is 8×8 pixels.

The WPM performs in four clock cycles the computation of the correlation between the search and the reference window, but with the advantage that while calculating the correlation in the first position inside the search window, the correlations corresponding to the adjoining three positions of the reference window begin to be calculated (Figure 5). The design uses data recycling, taking advantage of the same information previously read from external memory more than once, to perform several calculations in the same WPM. The WPM is replicated 5 times to cover the whole search window.

The optical flow module presents a maximum data recycling, exploiting the vertical and horizontal overlap of the reference window (Figure 6). In addition, the processing time using this implementation is reduced 50 times with respect to the sequential approach.

A full search in a window is processed in 8 clock cycles and the full image in 2,386,432 clock cycles. The motion vectors are calculated pixel by pixel, contrary to other hardware implementations where the motion vectors are obtained only for each reference window. In Figure 7, an approximation of the performance, obtained experimentally, of the architecture for the calculation of the motion estimation through optical flow is shown. The necessary clock cycles for the processing of the reference and search windows can be seen in (a), while (b) shows the number of clock cycles that are necessary for the processing of a full image. Both quantities depend on the number of PE blocks used.

4.3. 3D Recovery Module. The implementation of equation (2) can be achieved in two ways: the first one is to implement

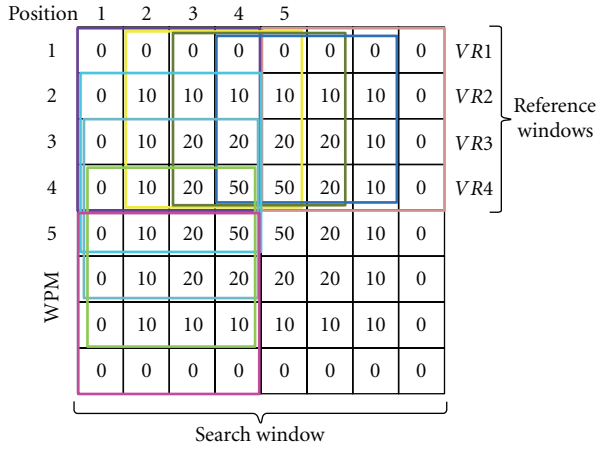


Figure 6: Computation of the correlation between the search and the reference window with 5 WPMs.

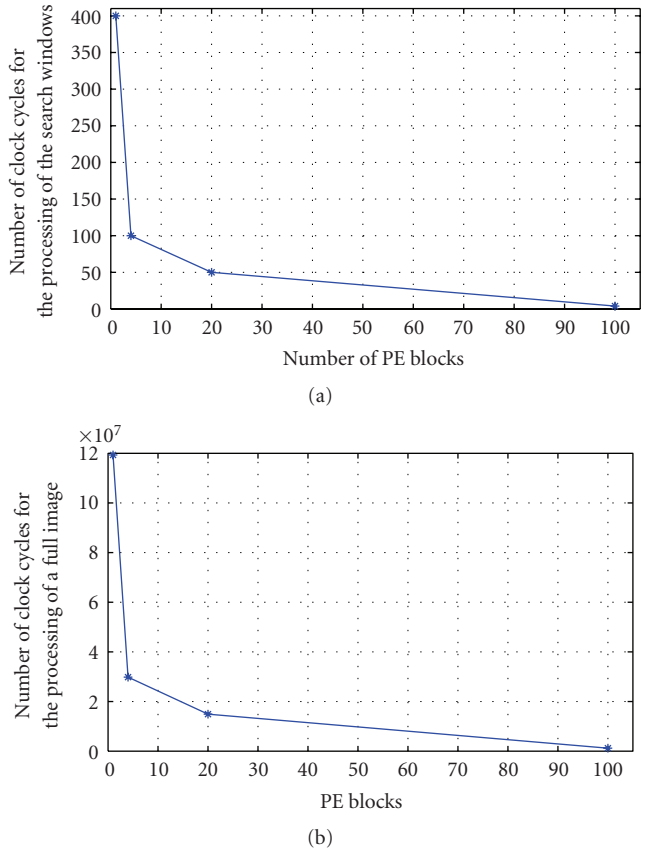


Figure 7: Performance analysis of the Optical Flow estimation for different PEs per WPM. The best area/performance compromise is around 20 PEs per WPM.

the equation with fully combinational elements. This option is inconvenient due to the complexity of the mathematical operations, which can lead to a significant degradation of the architecture performance. A more attractive option is the implementation using a *pipeline architecture* approach.

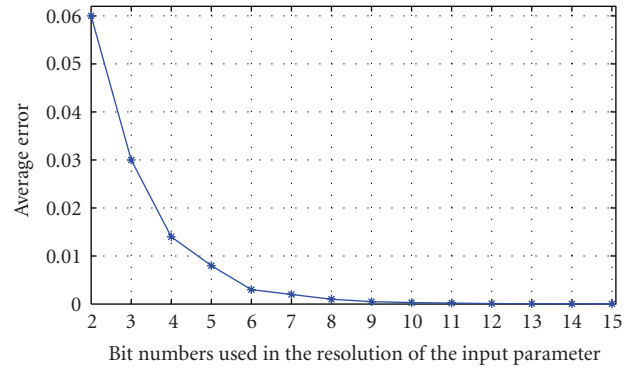


Figure 8: Average error as a function of the resolution used in A, B, C, U, V, y, W.

The 3D Recovery Module has A, B, C, U, V, W, x, y and motion vectors data as input. In the first pipeline stage, the values α and β are calculated: these variables depend only on the inputs. Part of the u_r and v_r values are calculated in this stage too. The square values of α and β and other part of u_r and v_r are calculated in the second stage. In the third stage, the variables u_r, v_r and the equation numerator are calculated. The fourth stage computes the denominator of the equation. Finally, in the last stage, Z is obtained.

Figure 9 shows the pipeline stages of the design with each of the intermediate modules. The operations in each stage are performed in fixed-point arithmetic, with different lengths for the integer and fractional part. Small errors are introduced due to the limited fixed point precision used for data representation in processing elements of the architecture. Currently a quantitative estimation of the error is being performed avoiding the excessive use of hardware resources.

The depth value is obtained with a 9-bit representation. Z uses 0 bits for the integer and 9 bits for the fractional part. The graphic in Figure 8 shows the average error when representing fractional values, using variables with different resolutions. For each case, all the variables have the same resolution, and their representation is always in fixed-point arithmetic. All the bits in this representation are used for the fractional part of the variables. The values shown were obtained through hardware implementation experimentation.

In Figure 8, it can be appreciated that the average error drops as the resolution of the input variables is incremented. This reduction is not linear, and the graphic shows a point where such reduction is not significant, no matter the increment in the number of bits of the input variables. 9 bits were chosen as a good compromise between area and average error.

Table 1 shows the calculation of depth in each of the *pipeline* stages. Once the motion vectors have been computed, the process in which the value of the depth is obtained begins. The input values A, B, C, U, V, and W simulate a translational motion of an object in the direction of the X axis. The motion of the object is of one pixel per image in the simulation.

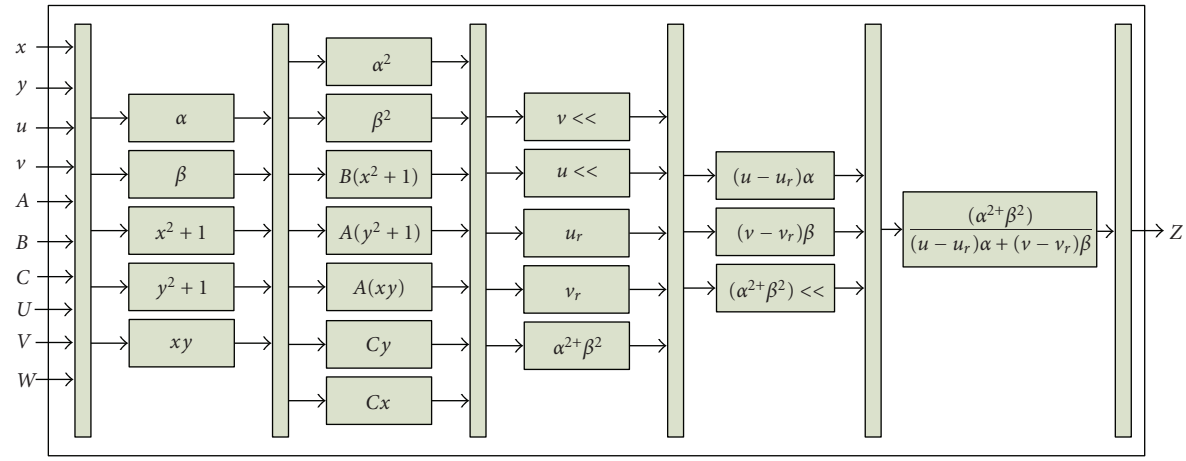


Figure 9: Architecture for the calculation of Z.

Table 1: Calculation of the depth in the different stages of the *pipeline*.

Parameters	Pipeline cycle	Stage 1		Stage 2		Stage 3		Stage 4		Stage 5	
$A = 0.00$		α	0.50								
$B = 0.00$		β	0.00								
$C = 0.00$	1	$y^2 + 1$	2.00								
$U = 0.50$		$x^2 + 1$	2.00								
$V = 0.00$		y^*x	1.00								
$W = 0.00$				α^2	0.25						
$x = 1.00$				β^2	0.00						
$y = 1.00$				$A^*(y^2 + 1)$	0.00						
$u = 1.00$	2			$B^*(x^2 + 1)$	0.00						
$v = 0.00$				$A^*(x^*y)$	0.00						
				$B^*(x^*y)$	0.00						
				C^*x	0.00						
				C^*y	0.00						
	3					u_r	0.00				
						v_r	0.00				
	4							$(u - u_r)\alpha$	0.50		
								$(v - v_r)\beta$	0.00		
										$\alpha^2 + \beta^2$	0.25
	5									$(u - u_r)\alpha+(v - v_r)\beta$	0.50
										z	0.50

4.4. Routers. The function of the Router units is to generate the addresses for the data read and write operations. To avoid a huge number of memory accesses, the routers store several rows from the images before the execution of any other action regarding the external memory. The *In-Router* module is composed by 12 buffers that store the rows of the images.

The block works as follows: eight rows from the current image (frame 1) are read and stored (search window). Then, four rows from the previous image (frame 0) are also read and stored (reference window). These pixel rows are stored in independent internal RAM memories (buffers). The router feeds 12 pixels in parallel to the optical flow module. When a full search window has been processed, a new pixel is read from the external memory and then stored in the last buffer. Each pixel of the actual address is transferred to the past buffer. The generation of the new read addresses is performed at the same time. The functional block diagram with input and output variables can be seen in Figure 10.

The *OUT-Router* (Figure 11) performs the writing of the architecture results to the external RAM memory. This block is simpler than the *In-Router* and is composed by an address generator and a register.

The *Gen_Addr_Esc* module controls the storage address of the datum corresponding to the depth, obtained in the *Depth calculation* module. The Register module put together 4 depth values calculated by the architecture in order to align them for memory write. This concatenation has the

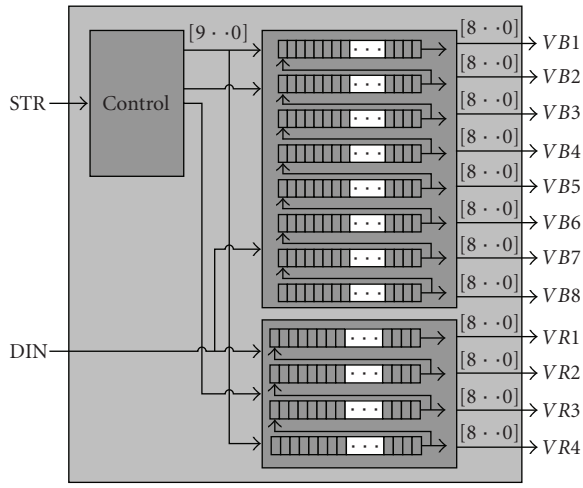


Figure 10: *IN-Router* block diagram.

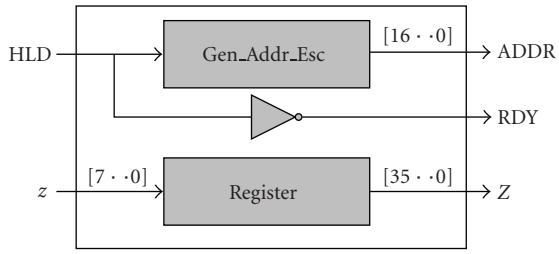


Figure 11: Component of the *OUT-Router* module.

purpose of storing a 36-bit value in each of the RAM locations.

4.5. Final Architecture. The final version of the architecture works as follows: in the first step, a pair of images from a sensor or another process is stored in the external memory. Next, a data array of the two images is read by the *In-Router* module: 8 rows of the previous image and 4 rows of the actual one. Once this is done, the data stored in the module are addressed to the WPM elements blocks, for the motion estimation. After being calculated, the motion vectors are passed to the next module for the depth calculation. In a parallel fashion, a new datum of the actual image and another of the reference image are acquired, and the process of the motion estimation is started.

After the computation of the depth has been performed, the result is stored in the external memory, where the system waits for the motion vectors for performing the process once again. This is repeated until the two images have been completely processed. When this is finished, a new pair of images is stored in the external memory.

5. Performance Analysis

From the general description of the architecture, an estimation of the performance of the architecture can be obtained. The processing speed of the architecture can be estimated as a function of the size of the image, the number of necessary cycles for the processing of a reference window of $n \times n$, the number of PEs operating in parallel, and the number of times the WPM are instantiated in the complete architecture. The analysis is based on the critical path which is the slowest module (the *Optical Flow Calculation*).

The number of clock cycles that are necessary to process a row of the search window of $m \times m$ pixels with the reference window of $n \times n$ pixels is given by the number of cycles required to process a row of the reference window plus the number of cycles that would take to process the positions that the reference window occupies over the search window, in a horizontal way. In order to compute the number of cycles required to process a row of the search window, the following is used:

$$\text{cycles}_{\times\text{row}} = \text{cycles}_{\times Vr} + \frac{\left(\text{cycles}_{\times Vr}\right)\left(V_{r\_\text{pos\_hor}}\right)}{V_{r\_\text{proc\_par}}}, \quad (5)$$

where $\text{cycles}_{\times\text{row}}$ is the number of cycles to process a row of the search window, $\text{cycles}_{\times Vr}$ is the number of cycles to process a row of the reference window, $V_{r\_\text{pos\_hor}}$ is the number of positions that the reference window occupies over the search window, in the horizontal direction, and $V_{r\_\text{proc\_par}}$ is the number of windows processed in parallel.

The number of cycles that are necessary to process a search window is the size of the reference window n, multiplied by the number of cycles required in processing a row of the search window by the number of positions occupied by the reference window above the search window in the vertical direction, divided by the number of processors that work in parallel and the number of times that the PE blocks array is repeated. Once the processing of the search window is done, two data are read from the external memory, so this will add 2 more clock cycles. Equation (6) allows the calculation of the number of cycles necessary to process the search window:

$$\text{cycles}_{\times V_b} = \frac{n\left(\text{cycles}_{\times\text{row}}\right)\left(V_{r\_\text{pos\_ver}}\right)}{\left(\text{PE}_{\text{par}}\right)\left(\text{WPM}_{\text{blocks}}\right)} + \text{cycles}_{\text{read}}, \quad (6)$$

where $\text{cycles}_{\times V_b}$ is the number of cycles required to process a search window, n is the size of the search window, $\text{cycles}_{\times\text{row}}$ is the number of cycles required to process a row of the reference window, $V_{r\_\text{pos\_ver}}$ is the number of positions of the reference window over the search window, PE_{par} is the number of processing elements working in parallel, and $\text{WPM}_{\text{blocks}}$ is the number of times that the PE array is repeated.

Finally, (7) represents the total number of cycles necessary to process a full image. This total is calculated multiplying the number of cycles required to process the search window by the number of search windows present in the image, both in a horizontal and in a vertical way,

$$\begin{aligned}\text{cycles}_{\times\text{Imag}} = {} & \left(\text{cycles}_{\times V_b}\right)\left(\text{Img}_{\text{hor}} - m + 1\right) \\ & \times \left(\text{Img}_{\text{ver}} - m + 1\right).\end{aligned} \quad (7)$$

Table 2: Synthesis results for the *Optical Flow Calculation* module.

Resources	Usage
FFs	1,986 of 10,944
LUTs	5,192 of 10,944
Slices	3,519 of 5,472
Max. Operating Freq.	70 MHz

Table 3: Synthesis results for the *Depth Calculation* module.

Resources	Usage
FFs	68 of 10,944
LUTs	1,882 of 10,944
Slices	983 of 5,472
Max. Operating Freq.	100 MHz

Table 4: Synthesis results for the full architecture.

Resources	Usage
FFs	2,177 of 10,944
LUTs	8,024 of 10,944
Slices	4,739 of 5,472
Block Rams	12 of 36
Max. Operating Freq.	66 MHz

For validation in this work, the following values were used:

(i) $n = 4$,
(ii) $m = 8$,
(iii) $\text{cycles}_{\times Vr} = 4$ cycles,
(iv) $V_{r\_\text{pos\_hor}} = m - n = 8 - 4 = 4$ positions,
(v) $V_{r\_\text{proc\_par}} = 4$ windows,
(vi) $V_{r\_\text{pos\_ver}} = m - n + 1 = 8 - 4 + 1 = 5$ positions,
(vii) $\text{PE}_{\text{par}} = 4$ PE blocks working in parallel,
(viii) $\text{WPM}_{\text{blocks}} = 5$ blocks of 20 PEs,
(ix) $\text{Img}_{\text{hor}} = 640$,
(x) $\text{Img}_{\text{ver}} = 480$.

Replacing the values of $\text{cycles}_{\times Vr}$, $V_{r\_\text{pos\_hor}}$, and $V_{r\_\text{proc\_par}}$ in (5), the following equation is obtained:

$$\text{cycles}_{\times V_{\text{row}}} = 4 + \frac{(4)(4)}{4} = 8\,\text{cycles}_{\times \text{row}}. \tag{8}$$

Now the values of n, $\text{cycles}_{\times V_b}$, $V_{r\_\text{pos\_ver}}$, $\text{WPM}_{\text{blocks}}$, and PE_{par} are replaced in (6) to obtain the number of cycles necessary to process the search window

$$\text{cycles}_{\times V_b} = \frac{4(8)(5)}{(4)(5)} + 2 = 10\,\text{cycles}\ p/Vb. \tag{9}$$

To obtain the total number of cycles necessary to process an image, the values of m, $\text{cycles}_{\times V_b}$, Img_{hor}, and Img_{ver} are replaced in (7)

$$\begin{aligned}\text{cycles}_{\times \text{Imag}} &= (10)(640 - 8)(480 - 8) \\ &= 2,994,090\,\text{cycles}\ p/\text{frame}.\end{aligned} \tag{10}$$

Table 5: Percentage of consumed resources of the FPGA device, by the modules of the proposed architecture.

Module	% resources
Optical Flow Calculation module	47
Depth Calculation Module	17
Routers and logic	9
Complete Architecture	73

Figure 12: Image from the used sequence.

For example, with a clock frequency of 50 MHz, the architecture could process 16 fps for a 640 × 480 pixels resolution image stream.

6. Implementation and Results

6.1. Architecture Implementation and Synthesis. The hardware design was described in Handel-C 4, the design was simulated in MatLab 7, and the synthesis design was carried on with Xilinx ISE Project Navigator 9.1. The board used for testing was an ML403 from Xilinx. The ML403 integrates a XC4VFX12 FPGA of the Virtex 4 family. The memory ZBT RAM was used to store the image.

Table 2 shows the consumption of hardware resources used by the *Optical Flow Calculation* module. Table 3 shows the use of resources of the *Depth Calculation* module. Table 4 refers to the resources usage of the full architecture, and finally Table 5 shows the percentage of resources used the modules and the complete architecture.

6.2. Results. An image sequence of a soda can was used to test the architecture (Figure 12). The sequence simulates the translational movement of the object on X axis by 1 pixel per frame.

Figure 13 shows the results of the optical flow obtained by the hardware module using the images sequence.

In the computation of optical flow, to try to summarize the resulting quality of millions of motion vectors as only a number is a complicated task, so several metrics were

Table 6: Comparison performance of the proposed architecture with other works.

Work	FPS	Resolution	Implement platform	Processed pixels/sec
Fife and Archibald [5]	30	320 × 240	FPGA	2304000
Diaz et al. [6]	0.05	640 × 480	FPGA	15630
Zach et al. [7]	2.46	640 × 480	HW/SW	755712
Proposed architecture	22	640 × 480	FPGA	6758400

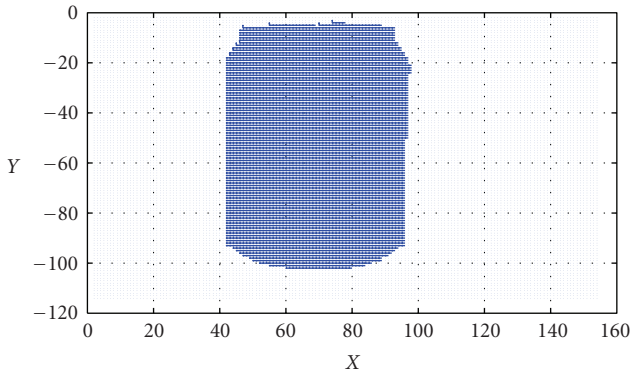


Figure 13: Optical flow calculated for the sequence.

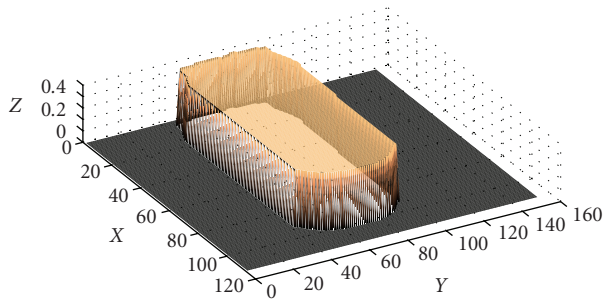


Figure 14: 3D recovery from optical flow for the can images.

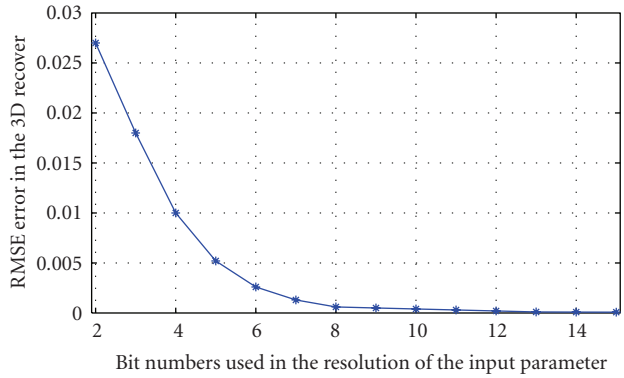


Figure 15: RMSE error in the 3D recovery, as a function of the resolution of the input variables A, B, C, U, V, y, W.

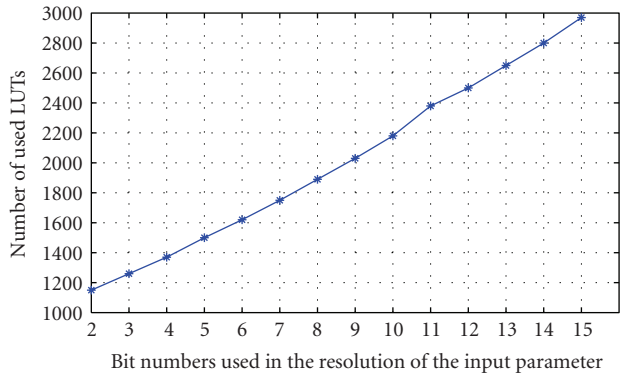


Figure 16: Number of used LUTs against the resolution of the input variables A, B, C, U, V, y, W.

evaluated. The first one is the error between the angle of the vectors obtained by software simulation and by the architecture [12]. The 100% of the vectors obtained by the architecture are correct with respect to those of the software implementation.

Figure 14 shows the results obtained by the processor for the 3D recovery.

Figure 15 shows the error obtained in the calculation of the depth against the resolution of the input variables. As in the graphic of Figure 8, the curve decreases quickly for the first values and then it stabilizes. At this point, the data representation precision increment has a little effect in the reduction of the error when calculating the 3D recovery.

Finally, the graphic in Figure 16 depicts the variation in the number of used LUTs of the device against the resolution of the input variables. Contrary to the last two graphics, this one presents an almost linear behavior. The consumption of resources grows as the number of bits used in the variables is incremented.

From the graphics, the number of bits for the input variables can be selected. In Figures 8 and 15, it can be seen that a resolution of 8 bits gives good results. Moreover, the amount of resources used with this resolution is moderated. It can also be seen that, when incrementing the number of bits in more than 8, the reduction in the error of the calculation of the 3D recovery is minimum. As a result of these points, the selected resolution for the 3D recovery based on the optical flow was 8 bits.

To measure the quality of the depth recovery, the RMSE metric was used. The average error given was of 0.00105 for several performed recoveries.

6.3. Discussion. The performance of the architecture is given as a function of the number of processed images, the number of operations performed in one second, and the number of

computed depth values. The following show a quantitative analysis of the architecture.

The processing time of the images in the architecture is conditioned by the maximum operating frequency, which is established as a function of the delays of the combinational logic, the way in which the clock signal is distributed, and the internal routing of the device. In the specific case of the implemented architecture, the maximum operating frequency is 66 MHz, which allows the processing of 22 frames per second, operating with 640×480 images. The architecture has the capacity of processing 4, 915, 200 depth values per second, with an average error of 0.00105.

Once the number of images per second that the design can process is known, the number of operations per second (OPS) performed by the architecture can be calculated. The OPS is obtained by multiplying the number of fps, the number of operations of the search window, and the amount of search windows in the image

$$\begin{aligned} \mathrm{OPS}_{\mathrm{SAD}} &= \mathrm{fps} * \left(\#\mathrm{operations}_{V_b}\right) \\ &\quad * \left(\mathrm{Img}_{\mathrm{hor}} - m + 1\right) * \left(\mathrm{Img}_v er - m + 1\right), \\ \mathrm{OPS}_{\mathrm{SAD}} &= 16 * (3 * 16 * 25) * (640 - 8 + 1) \\ &\quad * (480 - 8 + 1) = 5.748 \times 10^9. \end{aligned} \tag{11}$$

For the 3D recovery from the optical flow, the number of operations is obtained by multiplying the number of frames per second, the number of operations necessary to calculate a single depth value, and the number of motion vectors calculated for the image. Equation (12) allows the calculation of the number of performed operations that have to be completed for the 3D recovery through optical flow

$$\begin{aligned} \mathrm{OPS}_Z &= \mathrm{fps} * (\#\mathrm{operations}_Z) \\ &\quad * \left(\mathrm{Img}_{\mathrm{hor}} - m + 1\right)\left(\mathrm{Img}_v er - m + 1\right), \\ \mathrm{OPS}_Z &= 16(32)(640 - 8 + 1) \\ &\quad * (480 - 8 + 1) = 153.297 \times 10^6. \end{aligned} \tag{12}$$

The architecture performs 7.904 GOPS (Giga Operations per Second) in an integer representation for the optical flow, and 210 millions OPS in fixed-point representation for the 3D recovery. Thus, the architecture performs a total of 8.115 GOPS during the full 3D recovery process.

Our results compares favorably (see Table 6) with other implementations.

7. Conclusions and Future Work

The present work has discussed a parallel processor for the 3D recovery through Optical Flow inside a video sequence with real-time restrictions. The designs exhibit a balance between area utilization and processing speed. The implementation is capable of obtaining the optical flow from image sequences with VGA resolution in a predictable time, as it can process 22 fps.

It is possible to scale the proposed design so it can operate over the 30 fps or work with higher resolutions. This is performed by adding the necessary Optical Flow modules to process more search windows in a parallel fashion. In this way it could be possible to exploit the overlap of the search windows.

The computational load to perform the 3D recovery is of about 8 GOPS, which is difficult to perform in a short period (in the order of the milliseconds) with current sequential processors.

The architecture presents a small size, it is possible to implement it in systems where the space restrictions and the power consumption are the main concern, as in the case of mobile vision systems and in robotic vision.

Some points regarding future work are the following.

(i) Test different reference and search window sizes and analyze the results.

(ii) Analyze other algorithms for optical flow and their adaptation to the proposed architecture. The reuse of the architecture modules would imply minimum changes and a small or even null increment in the complexity of the proposed architecture.

(iii) Incorporate predictive algorithms and their hardware implementation to achieve a better 3D recovery.

Acknowledgment

The first author thanks the National Council for Science and Technology from Mexico (CONACyT) for financial support through the M.Sc. scholarship no. 206814.

References

[1] R. Koch and L. J. Van Gool, *3D Structure from Multiple Images of Large-Scale Environments*, Springer, London, UK, 1998.

[2] G. P. Martisanz and J. M. de la Cruz Garcia, *Vision por Computador: Imagenes Digitales y Aplicaciones*, Ra-Ma, Madrid, Spain, 2001.

[3] V. H. Rosales, *Recuperacion 3D de la estructura de una escena a partir del Flujo Optico*, M.S. thesis, INAOE, Puebla, Mexico, 2003.

[4] H. G. Nguyen, "Obtaining range from visual motion using local image derivatives," Technical Document 2918, RDT&E Division, Naval Command, Control and Ocean Surveillance Center, San Diego, Calif, USA, 1996.

[5] W. S. Fife and J. K. Archibald, "Reconfigurable on-board vision processing for small autonomous vehicles," *Eurasip Journal of Embedded Systems*, vol. 2007, Article ID 80141, 14 pages, 2007.

[6] C. Diaz, L. Lopez, M. Arias, C. Feregrino, and R. Cumplido, *Taller de Computo Reconfigurable y FPGAs, Implementacion FPGA del Calculo de Profundidades en la Recuperacion 3D Usando luz Estructurada*, Encuentro Nacional de Computacion, Apizaco, Mexico, 2003.

[7] C. Zach, A. Klaus, B. Reitinger, and K. Karner, "Optimized stereo reconstruction using 3D graphics hardware," in *Proceedings of the Vision, Modeling, and Visualization Conference (VMV '03)*, pp. 119–126, München, Germany, November 2003.

[8] A. Burton and J. Radford, *Thinking in Perspective: Critical Essays in the Study of Thought Processes*, Routledge, London,UK, 1978.

[9] D. H. Warren and E. Strelow, *Electronic Spatial Sensing for the Blind: Contributions from Perception*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1985.

[10] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, no. 1, pp. 43–77, 1994.

[11] A. Hernandez, *Recuperacion en Tiempo Real del Flujo Optico de una Secuencia de Video usando una arquitectura FPGA*, M.S. thesis, INAOE, Puebla, Mexico, 2007.

[12] B. McCane, K. Novins, D. Crannitch, and B. Galvin, "On benchmarking optical flow," *Computer Vision and Image Understanding*, vol. 84, no. 1, pp. 126–143, 2001.

[13] K. P. Horn and B. G. Rhunck, "Determining optical flow," Tech. Rep. A.I. Nemo 572, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass, USA, 1980.

[14] P. Kunh, *Algorithms, Complexity Analysis and VLSI Architectura for MPEG-4 Mostion Estimation*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[15] A. R. Bruss and B. K. P. Horn, "Passive navigation," Tech. Rep. A.I. Nemo 662, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass, USA, 1981.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 762362, 10 pages
doi:10.1155/2009/762362

Research Article

Hardware Accelerated Sequence Alignment with Traceback

Scott Lloyd and Quinn O. Snell

Department of Computer Science, Brigham Young University, Provo, UT 84602, USA

Correspondence should be addressed to Scott Lloyd, gscott@ieee.org

Received 15 March 2009; Revised 4 August 2009; Accepted 13 October 2009

Recommended by Cesar Torres

Biological sequence alignment is an essential tool used in molecular biology and biomedical applications. The growing volume of genetic data and the complexity of sequence alignment present a challenge in obtaining alignment results in a timely manner. Known methods to accelerate alignment on reconfigurable hardware only address sequence comparison, limit the sequence length, or exhibit memory and I/O bottlenecks. A space-efficient, global sequence alignment algorithm and architecture is presented that accelerates the forward scan and traceback in hardware without memory and I/O limitations. With 256 processing elements in FPGA technology, a performance gain over 300 times that of a desktop computer is demonstrated on sequence lengths of 16000. For greater performance, the architecture is scalable to more processing elements.

1. Introduction

Searching and comparing biological sequences in genomic databases are essential processes in molecular biology. The collection of genetic sequence data is increasing exponentially each year and consists mostly of nucleotide (DNA/RNA) and amino acid (protein) symbols. Approximately 3 billion nucleotide pairs comprise the human genome alone. Given the large volume of data, sequence comparison applications require efficient computing methods to produce timely results.

Biologists and other researchers use sequence alignment as a fundamental comparison method to find common patterns between sequences, predict protein structure, identify important genetic regions, and facilitate drug design. For example, sequence alignment is used to derive flu vaccines [1] and by the nation's BioWatch [2] program in identifying DNA signatures of pathogens. Sequence alignment consists of matching characters between two or more sequences and positioning them together in a column. Gaps may be inserted in regions where matches do not occur to reflect an insertion or deletion evolutionary event. A count of the matching characters results in a measure of similarity between the sequences. Pairwise alignment involves two sequences (see Figure 1) and multiple alignment considers three or more sequences. Finding the optimal multiple sequence alignment is NP-hard in complexity. As a first step, multiple alignment algorithms [3, 4] often compute a pairwise alignment between all the sequences.

Global and local pairwise alignments are the two most common alignment problems. Global alignment [5] considers both sequences from end to end and finds the best overall alignment. Local alignment [6] identifies the sections with greatest similarity and only aligns the subsequences. Both alignment problems are typically solved with dynamic programming (DP), which fills a two-dimensional matrix with score or distance values in a forward scan from upper left to lower right, followed by a traceback procedure. Traceback occurs from a designated lower right position following a path to upper left, thereby determining the best alignment.

The computational cost for an optimal sequence alignment increases exponentially with the length of each sequence and with the number of sequences. This complexity poses a challenge for sequence alignment programs to return results within a reasonable time period as biologists compare greater numbers of sequences. Using current methods, an alignment program may run for days or even weeks depending on the number of sequences and their length.

Unlike most acceleration methods that focus on sequence comparison, this research describes and evaluates a space-efficient, global sequence alignment algorithm and

```
- - TTCT - - T - - TAGATTC
CCTTCTACTGCTA - CTTC
```

Figure 1: Example pairwise alignment.

architecture that includes traceback for implementation on reconfigurable hardware. Given a pair of sequences, the accelerator returns a list of edit operations constituting the optimal alignment. A library of accelerator functions is easily incorporated into multiple sequence alignment programs that run on platforms equipped with reconfigurable hardware.

2. Related Work

Most efforts to accelerate biosequence applications with hardware have focused on database searches. Ramdas and Egan [7] compare several of these architectures in their survey. Given a query sequence, an entire genetic database is scanned looking for other sequences that are similar. Searching a genetic database for matches with a biosequence is similar in nature to a search of the web that returns "hits" sorted by relevance. Accelerating a database search is a simpler problem than alignment. Only the score for the comparison is computed by hardware in the forward scan; whereas alignment requires traceback in addition to the forward scan. The sequence comparison problem can be mapped to a linear systolic array of processing elements (PEs) requiring $O(\min(m, n))$ space, where m and n are the lengths of the sequences. However, global alignment necessitates extra storage for traceback pointers and a traceback procedure, which are not addressed by sequence comparison solutions.

Traceback support in hardware has the most benefit when the traceback path spans a significant portion of the DP matrix. Global alignment applications realize the greatest performance gain because the traceback path extends across the entire DP matrix; whereas local alignment applications with a shorter path show less benefit. After a forward scan in hardware, any alignment in software must recompute the DP matrix and traceback pointers for the section of interest before determining an optimal traceback path. For instance, accelerated database search applications may compute an alignment in software only between high-scoring matches and the query sequence after the comparison phase. These search applications usually run in acceptable time with relatively short query sequences; however, comparative genomic applications commonly align long sequences at greater computational cost and stand to benefit from accelerated alignment. Examples include whole genome alignment [8], whole genome phylogeny [9], and computation of pathogen detection signatures [10].

The predominant, nonparallel algorithms for global sequence alignment are described by Gotoh [11] and Myers and Miller [12]. Both algorithms execute in $O(mn)$ time. The algorithm presented by Gotoh requires $O(mn)$ space, while the algorithm of Myers-Miller needs only $O(\log m + n)$ space, but it incurs a factor of 2 time penalty. Most of the space is used to hold values of the DP matrix and the traceback pointers. Saving all traceback pointers in an array requires only one forward scan through the DP matrix followed by one traceback pass. Otherwise, multiple passes through the DP matrix are required if not saving all the traceback pointers. The downside of saving all the traceback pointers is the $O(mn)$ space requirement, which can be significant for longer sequence lengths or prohibitive when limited by FPGA memory.

A few efforts propose hardware methods for accelerating pairwise alignment and traceback. The work presented by Hoang and Lopresti [13] describes an FPGA architecture which consists of a linear systolic array of PEs that output traceback data. However, the type of sequences is limited to only DNA and the sequence length is limited by the number of PEs on the accelerator (a couple of hundred nucleotides). The works by Jacobi et al. [14] and VanCourt and Herbordt [15] suggest accelerated traceback methods, but with few details. The sequence length accommodated by their accelerators is also limited by the number of PEs on the accelerator like the one described by Hoang. Another limitation of the Hoang and VanCourt methods is that traceback cannot be overlapped with another forward scan since the systolic array is used for both scan and traceback.

The methods presented by Yamaguchi et al. [16] and Moritz et al. [17] allow longer sequences by partitioning the sequences through the pipeline of PEs. Nevertheless, the traceback data must be saved to external memory, since the size of the data exceeds the amount of available internal FPGA memory. Hence, the traceback performance of both methods is limited by the FPGA bandwidth to external memory. The design described by Benkrid et al. [18] also partitions sequences, but the size of FPGA memory ultimately limits the length of sequences that are aligned with hardware acceleration. Operating at 100 MHz, a systolic array with 256 PEs requires at least 6.4 GB/s of memory bandwidth to store 2-bit traceback data from each PE. As PE densities and clock frequencies increase, the external memory bandwidth is easily exceeded. Internal FPGA memory has sufficient bandwidth, but even modest sequence lengths of 16 K require 64 MB of traceback store, which far exceeds current FPGA internal memory capacities.

The global alignment algorithm presented in this paper overcomes the memory size and bandwidth limitations of FPGA accelerators and does not limit the sequence length by the number of PEs. Long sequences of DNA and protein are accommodated by the algorithm through a space-efficient traceback procedure that is accelerated in hardware. Traceback may occur in parallel with the next forward scan since it is implemented in a separate process from the systolic array.

3. Algorithm

The general algorithm is described first followed by the FPGA architecture in the next section. The algorithm is based on dynamic programming (DP), but partitions the problem into slices for the FPGA hardware. A description of the general sequence alignment problem is also found in [5, 11].

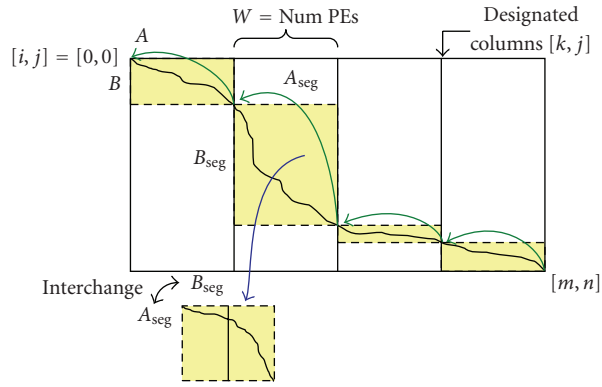


Figure 2: Forward scan and traceback.

Given a pair a *sequences* $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ of *length* $|A| = m$ and $|B| = n$ from the finite alphabet Σ, a *sequence alignment* is obtained by inserting *gap characters* "-" into A and B. The aligned sequences A' and B' from the extended alphabet $\Sigma' = \Sigma \cup \{\text{"-"}\}$ are of equal length such that $|A'| = |B'|$. Let the function $s : \Sigma \times \Sigma \to \mathbb{Z}$ determine the similarity of symbol a_i with b_j, and let the constant α represent the cost of inserting/deleting a gap. Let H denote the DP matrix and the element $H[i, j]$ the similarity score of sequences $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$. An optimal alignment is obtained by maximizing the score in each element of H. The values of H are determined by the following recurrence relations for $1 \le i \le m$ and $1 \le j \le n$:

$$\begin{aligned} H[0,0] &= 0, \\ H[i,0] &= H[i-1,0] + \alpha, \\ H[0,j] &= H[0,j-1] + \alpha, \\ H[i,j] &= \max \begin{cases} H[i-1,j-1] + s\left(a_i, b_j\right), \\ H[i-1,j] + \alpha, \\ H[i,j-1] + \alpha. \end{cases} \end{aligned} \tag{1}$$

The matrix fill occurs in a scan from upper left to lower right because of dependencies from neighboring elements. During the forward scan, a pointer $p \in \{\text{DIAG, ABOVE, LEFT}\}$ indicates the current selection of the max function in (1). Given a tie, fixed priority resolves the selection. The value of p is saved to the traceback matrix T, thus $T[i, j] = p$. Following the forward scan, traceback proceeds from $T[m, n]$ to $T[0, 0]$, thereby determining the best alignment. The result is a list of edit operations $e \in \{\text{SUBSTITUTE, INSERT, DELETE}\}$.

The scan algorithm presented here builds upon the space-saving concepts described by Edmiston et al. [19], and the divide-and-conquer scheme of Guan and Uberbacher [20]. Since sequence lengths are often longer than the number of PEs available in a systolic array, the problem is often partitioned [21]. The forward scan consists of two fundamental scan procedures ScanPartial and ScanFull. The Partial and Full descriptors refer to the amount of traceback data saved by the procedures. ScanPartial partitions the DP matrix H into slices of width W. The slices are processed iteratively. The result of processing each slice is a column of traceback pointers $R[k, j]$ that refer to a row in a prior slice (see Figure 2). The designated columns k are given by $k \in \{c \mid c \bmod W = 0 \vee c = m\}$. The row pointers form a partial traceback path through H that link only the right-most columns of each slice. Given that p indicates the heritage of element $H[i, j]$, the following recurrences for $1 \le i \le m$ and $1 \le j \le n$ determine R

If $i \bmod W = 1$, then

$$R[i,j] = \begin{cases} j-1 & \text{if } p = \text{DIAG}, \\ j & \text{if } p = \text{LEFT}, \\ R[i,j-1] & \text{if } p = \text{ABOVE}, \end{cases}$$

else

$$R[i,j] = \begin{cases} R[i-1,j-1] & \text{if } p = \text{DIAG}, \\ R[i-1,j] & \text{if } p = \text{LEFT}, \\ R[i,j-1] & \text{if } p = \text{ABOVE}. \end{cases} \tag{2}$$

Only the designated columns of R are actually stored, which correspond to the right-most columns of a slice. The values for the other columns are retained temporarily with a vector variable that follows the wavefront of the scan. In contrast, the ScanFull procedure does not partition the DP matrix and produces a full matrix T of traceback pointers that refer to adjacent elements of H.

The TracePartial procedure differs from TraceFull in that the partial set of traceback pointers from R are followed instead of the full set from T. The row pointers, from $R[m, n]$ to $R[0, 0]$ in designated columns, identify waypoints on the optimal path through the DP matrix. Since the row pointer in $R[k, j]$ refers to a row in a prior slice, a block between the columns is identified, along with corresponding segments of A and B. The segments of A and B are passed to ScanFull and TraceFull to determine the full path from $[k, j]$ back to $[k_{\text{prev}}, R[k, j]]$. The alignment results from each block are concatenated and thereby form a complete path from $[m, n]$ to $[0, 0]$.

Since the vertical height of a block (the length of a B segment) is unbounded, the traceback space available to the Full procedures may be exceeded. To avoid this case, a vertical threshold Y is defined such that if exceeded, the Partial procedures are called instead, with the segments of A and B interchanged in the calls. Algorithm 1 shows the procedure that is central to bounding the memory required for traceback. TracePartial is called recursively a maximum of once. Any segments passed to the Full procedures will not exceed W and Y in length because of the partitioning done by ScanPartial. In the worst case, the length of sequence A is bounded by the first call to ScanPartial and the length of B is bounded by the second call.

```
procedure TracePartial(A, B, m, n, R, E)
{
  x2 ← m, y2 ← n
  while (x2 > 1) do
    x1 ← ⌊(x2 − 1)/W⌋ · W + 1
    y1 ← (x1 > 1 ∧ y2 ≥ 1) ? R[x2, y2] + 1 : 1
    xlen ← x2 − x1 + 1, ylen ← y2 − y1 + 1
    if (ylen = 0) then
      Add xlen DELETE operations to E′
    else if (ylen ≤ Y) then
      ScanFull(Ax1, By1, xlen, ylen, T)
      TraceFull(Ax1, By1, xlen, ylen, T, E′)
    else // interchange A and B
      ScanPartial(By1, Ax1, ylen, xlen, R′)
      TracePartial(By1, Ax1, ylen, xlen, R′, E′)
      ∀e ∈ E′: replace DELETE ⇔ INSERT
    end if
    E ← E ∪ E′
    x2 ← x1 − 1, y2 ← y1 − 1
  end while
}
```

Algorithm 1: Procedure for TracePartial.

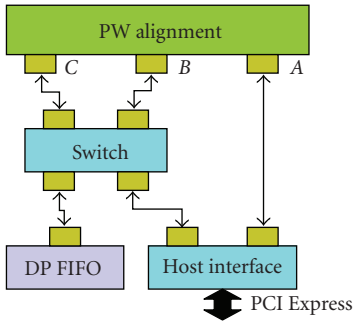


Figure 3: System architecture.

4. Architecture

The global alignment accelerator is implemented using Qnet [22], an open-source packet-switched network architecture similar to DIMEtalk [23]. Qnet components interconnect the host and other FPGA accelerator modules in the system. The architecture facilitates system design with reusable modules that encapsulate sharable devices or resources. Qnet encourages parallelism by offering concurrent, high-performance data paths between modules. Figure 3 shows the alignment system constructed with Qnet modules and components. A few specifics of Qnet are given before describing the alignment accelerator module and system operation.

4.1. Qnet Components. The basic network components consist of a switch, Qports, and Qlinks. As the central figure in the network, the switch provides a path for communicating packets to other modules. Qports are the interface between modules and the network, and are the addressable endpoints of communication. Qports are connected by Qlinks, which consist of paired, unidirectional, point-to-point signaling channels that are each 32-bits wide in this system, but may be implemented with other bit widths. Each Qport has word-based flow control that will apply back-pressure on a link, delaying communication until the port is ready to receive. Hence, packets are not arbitrarily discarded, and the requirement to buffer an entire packet at the input of a module is removed while still maintaining performance. Qnet communication performance has been shown to be very near the theoretical max bandwidth between modules on the FPGA while also maintaining latencies very near theoretical minimums.

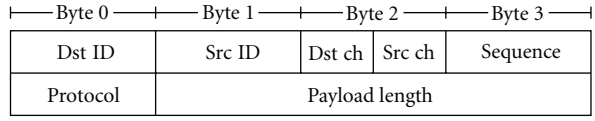


Figure 4: Qpacket header.

Qnet reliably transfers data packets between endpoints through a simple protocol that requires minimal FPGA resources. Packets consist of a small header (see Figure 4) and a payload of variable size. The header specifies the source and destination endpoints with unique port identifiers and also indicates the payload length. When a packet header enters the switch, the output port is determined from the destination endpoint and remains the same for all following words of the packet. With a cut-through packet forwarding method, the full packet is not buffered in the switch. Packets that enter the switch simultaneously with different destinations pass through concurrently. This architecture allows parallel data transfer on all ports of an accelerator module.

4.2. System Modules. Host Interface. The host computer communicates with the FPGA accelerator through the PCI Express [24] module, which contains DMA engines and translates PCI packets into Qnet packets. Two ports on this module allow both sequences to be sent in parallel to the accelerator.

DP Matrix FIFO. If the length of sequence A is longer than the number of PEs in the accelerator, the DP matrix H must be processed in slices of width W = (num. PEs) as described in Section 3. After processing a slice, the right column of DP matrix values will exit the pipeline of PEs. These H values are sent in a packet to the DP matrix FIFO and retained for processing the next slice through the pipeline. Any packet sent to the DP matrix FIFO will be returned to the originating Qport, as indicated by the packet header, thus cycling the pipeline output to the input. The FIFO may be implemented with any memory technology of sufficient bandwidth and size to handle the stream of data from the PE pipeline. Since only one H value exits the pipeline each clock cycle, the bandwidth requirement is not excessive.

Pairwise Alignment Module. The compute intensive portions of the alignment algorithm are performed by the pairwise

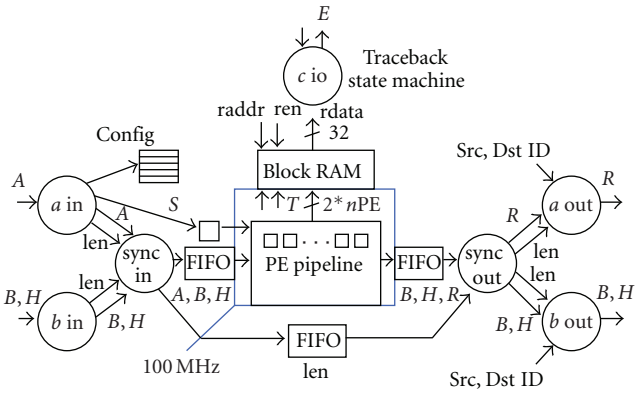


Figure 5: Pairwise alignment module.

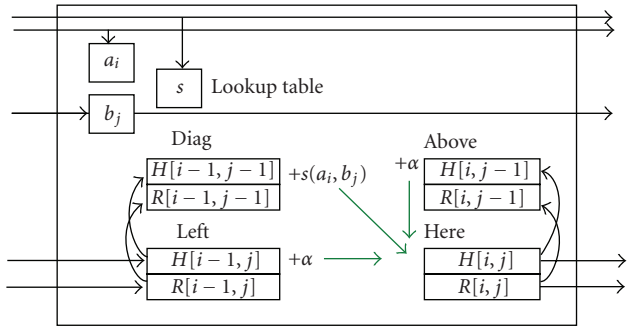


Figure 6: Processing element architecture.

alignment module, which contains the pipeline of PEs. This module has 3 Qports through which the sequences are provided and results are returned (see Figure 3). In parallel, Sequence A is input on port A and sequence B is input on port B, while the traceback results are returned on port C.

Figure 5 shows the internal architecture of the alignment module. The front-end of the pipeline synchronizes the A and B streams of symbols, and the back-end sends the partial traceback results R out on port A and the H values on port B. The symbols of sequence B that flow through the pipeline are merged with the H values on output, since they will also be needed in processing additional slices. Merged B and H values that exit the pipeline are sent in a packet to the DP matrix FIFO. As sequence A is fed into the pipeline, merged B and H values from the end of the pipeline flow from the alignment module through the DP matrix FIFO and back into the front-end of the pipeline at port B. This cycle occurs for each slice of the scan, except for the last.

Most systems commonly load a segment of A into the pipeline and then shift in B; whereas this system enters A and B in parallel [25]. Sequence B is shifted in as usual, but A is bussed to each PE and latched when the first symbol of B reaches a PE in the pipeline (see Figure 6). The recurrence equations described in Section 3 are calculated by the PEs each time a pair of symbols enter the pipeline. As a forward scan proceeds from upper left to lower right, the pipeline of PEs operates in parallel along an antidiagonal wavefront through the DP matrix. Figure 7 shows the progression of symbols in the pipeline and shows the mapping of PEs to DP matrix cells over several cycles.

Both of the forward Scan procedures are implemented by the pipeline of PEs. ScanPartial enables the R (partial row pointer) output, while ScanFull enables the T (full traceback pointer) output. Configuration bits in the packet header of sequence A determine which pointer type is enabled. For each slice processed by ScanPartial, a column of R is returned to the host in a packet. ScanFull will only process one slice, while saving the full traceback data in FPGA block RAM, which has the bandwidth to store pointers from every PE in parallel. The vertical threshold Y, as described in Section 3, is determined by the depth of FPGA block RAM allocated to full traceback.

A state machine implements the TraceFull procedure that follows the pointers saved in block RAM by ScanFull. To initiate a full traceback, a request packet is sent to Port C of the pairwise alignment module from the host. The results, a list of edit operations $e \in E$, are returned to the host from Port C. TracePartial is implemented in software on the host, but calls the Full procedures for most of the work (see Algorithm 1).

Access to traceback pointers $T[i,j]$ in block RAM requires a skewed addressing scheme because of the storage method used in the forward scan. Storing a diagonal wavefront of pointers as a row in block RAM skews the traceback matrix T in memory (see Figure 8). A full traceback begins with a request packet that contains the cell address of $T[1,1]$ and the lengths of sequences A and B. The address of $T[1,1]$ is saved at the start of a full forward scan and will always be the lowest address in a row (leftmost). From the address of $T[1,1]$ and the width W of block RAM in cells, the address of $T[m,n]$ is calculated

$$\begin{aligned} m' &= m - 1, \\ n' &= n - 1, \\ \text{addr}_{T[m,n]} &= \text{addr}_{T[1,1]} + W(m' + n') + m'. \end{aligned} \tag{3}$$

Traceback proceeds from $T[m,n]$ to $T[0,0]$ following the pointer in each accessed cell. Given a traceback pointer p from the current cell, the following equation determines the address of the next cell in block RAM

$$\text{addr} = \begin{cases} \text{addr} - (2W+1) & \text{if } p = \text{DIAG}, \\ \text{addr} - (W+1) & \text{if } p = \text{LEFT}, \\ \text{addr} - W & \text{if } p = \text{ABOVE}. \end{cases} \tag{4}$$

Since block RAM is dual-ported, traceback reads can occur while the next forward scan concurrently saves pointers in another portion of the traceback memory. Address calculations into block RAM wrap around when the range is exceeded.

4.3. System Parameters. Most system parameters are implemented with VHDL generics. For example, symbol width, number of PEs, traceback memory depth, and various register sizes are all specified at a high level in the module

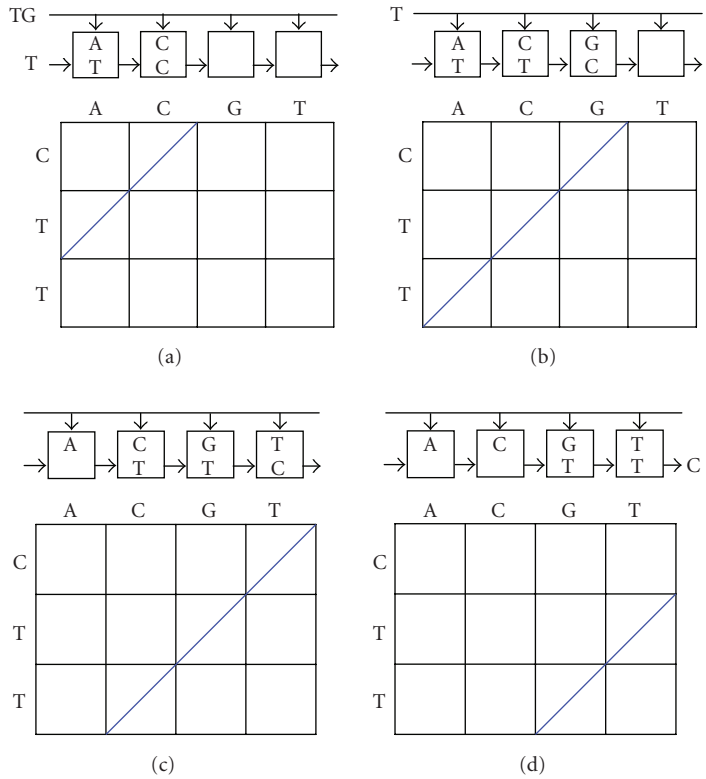


FIGURE 7: Symbol flow and the corresponding DP matrix wavefront for sequential cycles of the PE pipeline.

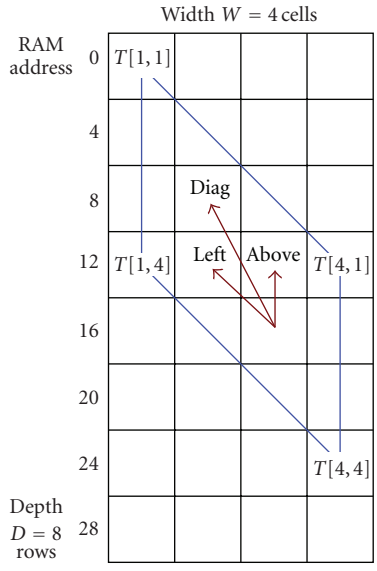


FIGURE 8: The traceback matrix T is skewed in memory. The pointers show how to address neighboring cells during traceback in the skewed matrix.

hierarchy and passed as generics to lower modules. This allows different configurations of the accelerator with minimal changes to the source. Protein sequences require 5-bit symbols and DNA sequences require at least 2-bit symbols. Mega-length sequences may be handled by the architecture and algorithm by setting system constants and rebuilding a system. The number of PEs is scalable to match the target hardware resources.

Several system parameters affect the maximum sequence length $L_{\max}$ that can be processed by the accelerator. As mentioned previously, the DP matrix FIFO must be deep enough to hold the merged B symbols and H values that come from the end of the pipeline. The FIFO length limit is determined by $L_F = N_{\text{FIFO}}/N_{BH}$, where N_{BH} denotes the number of bytes for a single B-H pair and N_{FIFO} denotes the DP matrix FIFO size in bytes. Also, the substitution and gap costs combined with the H register size affect the maximum sequence length. Each stage of the pipeline increments an H value by the gap cost α or the result of the similarity function $s(a_i, b_j)$. To avoid H register overflow, the H length limit is $L_H = (2^{N_H - 1} - 1)/I_{\max}$, where N_H denotes the number of bits in H registers, and $I_{\max}$ denotes the maximum absolute value of the gap cost α or the similarity function s. In conjunction with the other parameters, the R register size affects the maximum sequence length. A register for R must hold an index into sequence B without overflow. Given N_R, the number of bits in R registers, the R length limit is $L_R = 2^{N_R} - 1$. From the contributing length limits, the maximum sequence length is determined by $L_{\max} = \min(L_F, L_H, L_R)$.

5. Timing Model

A timing model is presented for the sequence alignment algorithm and architecture described in Sections 3 and 4.

First, constants for the system are defined with the values in parenthesis being specific to the evaluation system:

$$\begin{aligned} W &: \text{number of PEs } (256),\\ Y &: \text{threshold for length of sequence } B \ (768),\\ C_{\text{pad}} &: \text{cycles to pad pipeline } (8),\\ t_s &: \text{communication startup } (1.5\,\mu\text{s}),\\ t_h &: \text{host overhead } (3\,\mu\text{s}),\\ t_{\text{clk1}} &: \text{period of clock 1 } \left(\frac{1}{100\,\text{MHz}}\right),\\ t_{\text{clk2}} &: \text{period of clock 2 } \left(\frac{1}{150\,\text{MHz}}\right). \end{aligned} \tag{5}$$

Timing varies as a function of the following variables:

$$\begin{aligned} l &= |A'| = |B'|, \text{ aligned length},\\ m &= |A|, \text{ length of sequence } A,\\ n &= |B|, \text{ length of sequence } B,\\ N_{\text{slice}} &= \lceil m/W \rceil, \text{ number of slices.} \end{aligned} \tag{6}$$

The time for processing a slice is determined by the length of B or the length of the pipeline plus padding, whichever is greater. Flush time depends on how much of sequence B is left in the pipeline after processing a slice and is calculated from the length of B minus padding (zero limited) or the length of the pipeline, whichever is less

$$\begin{aligned} t_{\text{slice}} &= t_{\text{clk1}}\left[\max\left(n, W + C_{\text{pad}}\right) + 1\right],\\ t_{\text{flush}} &= t_{\text{clk1}} \min\left(W, \max\left(0, n - C_{\text{pad}}\right)\right). \end{aligned} \tag{7}$$

Based on the previous definitions, execution times for the Scan and Trace procedures are

$$\begin{aligned} t_{\text{scan}F} &= t_{\text{slice}} + t_{\text{flush}} + 4t_s,\\ t_{\text{trace}F} &= t_{\text{clk2}}(2l + 4) + 2t_s,\\ t_{\text{scan}P} &= N_{\text{slice}}(t_{\text{slice}} + t_s) + t_{\text{flush}} + 4t_s,\\ t_{\text{trace}P} &= N_{\text{slice}}(t_{\text{scan}F} + t_{\text{trace}F}). \end{aligned} \tag{8}$$

Finally, the time to perform a global sequence alignment is given by:

$$t_{\text{align}} = \begin{cases} t_{\text{scan}F} + t_{\text{trace}F} + t_h & \text{if } m \le W \wedge n \le Y,\\ t_{\text{scan}P} + t_{\text{trace}P} + t_h, & \text{else.} \end{cases} \tag{9}$$

This analytical model matches experimental results and predicts the scalability and performance of the architecture under various system configurations.

6. Experimental Setup

Application. Three global alignment implementations are tested in the evaluation: (1) as a baseline, a software-only version of the algorithm presented in this paper; (2) a version accelerated by the FPGA; and (3) an implementation of the Myers-Miller global alignment algorithm for an additional point of reference. The host computer is used to evaluate the software only versions of the algorithms. Seq-Gen [26] produced varying lengths of test sequences ranging from 128 to 16383 symbols for the evaluation. The applications use a gap cost of −2, a substitution score of 1, and a match score of 2.

Host. The host platform consists of a desktop computer with a 2.4 GHz Intel Core2 Duo processor running Fedora 6 Linux as the operating system. All benchmark applications execute in a single thread and are compiled with gcc using −O3 optimization. For accurate timing, the processor's performance counters are used.

Accelerator. An 8-lane PCI Express add-in card with a Xilinx Virtex-4 FX100 FPGA provides the hardware acceleration. To conserve FPGA resources, only 4 of the 8 PCI Express lanes are used in the experimental system. All of the components are implemented in VHDL. As shown in Figure 3, a 4-port switch connects the three FPGA modules using 32-bit Qlinks that run at 150 MHz. For simplicity and minimal latency, the switch is implemented with a fixed address table and a fixed port priority resolution scheme. The DP matrix FIFO uses 64 KB of FPGA block RAM, which is enough to hold 16 K entries of B symbols and H values. Driven by a 100 MHz clock, the pipeline consists of 256 PEs placed in a tiled pattern. DNA and protein sequences are accommodated with 5-bit symbol values. An 8-bit look-up table that requires one block RAM per PE implements the similarity function $s(a_i, b_j)$. Each PE outputs a 2-bit traceback pointer p that is stored in traceback memory, which is instantiated in 64 KB of block RAM with a width of 512 bits and a depth of 1024. The traceback memory depth determines the Y threshold. Within the system, DP matrix values H and row pointer values R both require 16-bits.

Through the use of constraints and floor planning, 90% slice utilization is achieved. First, an area shape and size constraint for one PE is determined, in this case, by repeated place and route trials. Then, given this shape and size, a simple (75 line) Perl script tiles the PEs in a programed pattern by generating area constraints for each PE. Keep-out areas are also given to the Perl script. The text output from the Perl script is pasted into the user constraints file for use by the place and route tools along with the other constraints. Only slice resources are constrained for the PEs, since the block RAM needed for each PE may not reside within the area constraint. To meet timing, the first and last PEs of the pipeline are kept closer to the Qport interfaces of the switch and alignment module, which is shown in Figure 9 along with the tiling pattern. The traceback block RAMs are constrained to a centrally located area of the FPGA to minimize path lengths from distant PEs. For proximity to the traceback memory, the traceback state machine is also centrally located. Table 1 shows the relative resource usage of the various components.

Table 1: Resource usage.

Component	Slices	FPGA percentage
PCI Express	6175	14.6%
Host interface	1221	2.9%
4-port switch	448	1.1%
Traceback	283	0.7%
DP FIFO	192	0.5%
PE (one)	111	0.3%

Table 2: Speedup between implementations.

Sequence length	t_{FPGA} μs	$\frac{t_{\text{Myers}}}{t_{\text{FPGA}}}$	$\frac{t_{\text{Host}}}{t_{\text{FPGA}}}$
511	64	131	107
1023	128	171	124
2047	327	264	181
4095	969	357	236
16383	11696	471	304

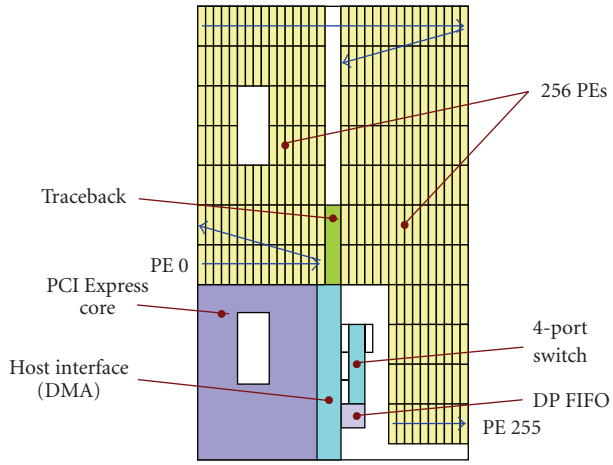


Figure 9: FPGA floorplan.

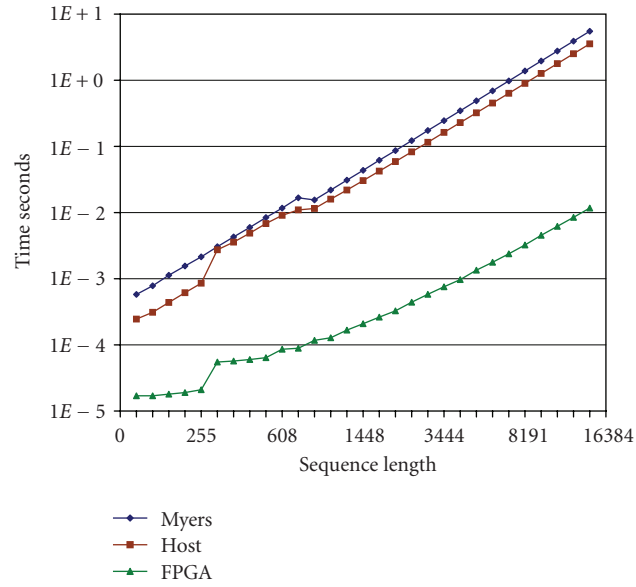


Figure 10: Global alignment execution time.

7. Results

Figure 10 shows the performance of the three global sequence alignment implementations with varying lengths of sequences and Table 2 compares the speedup between the implementations. The host-only version averages a speedup of 1.6 over the Myers-Miller implementation and the accelerated version achieves a max speedup of 304 over the host version. During the forward scan, the accelerator reaches a peak dynamic programming rate of 25.6×10^9 cell updates/s (CUPS). Traceback occurs at a peak rate of 75×10^6 pointers/s. Figure 11 shows the actual performance compared with the timing model from Section 5. For longer sequences, the actual performance is near the theoretical peak. The timing model suggests a high degree of scalability for the presented algorithm and architecture. For example, performance predicted by the model gives a speedup of 580 with 512 PEs operating at 100 MHz on a larger FPGA.

Supported by the low communication overhead of Qnet, sequences of length 10 or greater are aligned faster on the accelerator. Sending a single packet from the host to the accelerator takes minimally 1.5 μs. The demonstration system takes a minimum of 14 μs for an alignment with most of the time being attributed to the overhead of several packets, since only 2.65 μs is required for a single pipeline fill and flush once sequences are ready at the front-end of the pipeline.

Sequences shorter than W have a lower bound on alignment time, because unused PEs must be filled with null symbols. Longer sequences realize greater performance on the accelerator because the pipeline does not require a flush between adjacent slices. Adjacent slices need only 1 cycle of spacing in the pipeline. Longer sequences are also more efficient because of proportionately less time spent in the traceback. The average traceback time relative to the forward scan can be visualized in Figure 2 as the area of the sub-blocks relative to the area of the whole matrix.

Even though the algorithm presented here requires $O(mn)$ space, the traceback memory is reduced by a significant constant. For example, given sequences with 100 K symbols, saving all the traceback data requires 2.5 GB. By saving the partial traceback pointers in a system with 256 PEs, the traceback data is reduced to 78 MB. Perhaps more importantly, the necessary memory bandwidth to store the partial traceback pointers is reduced to a practical level that is achievable between the host computer and the FPGA accelerator. With the pipeline running at 100 MHz and 16-bit R values, the partial traceback data rate is only 200 MB/s.

Qnet provides communication bandwidth up to 600 MB/s per link in each direction between modules, which exceeds the rate needed by the alignment module to maintain maximum throughput in the pipeline. With excess

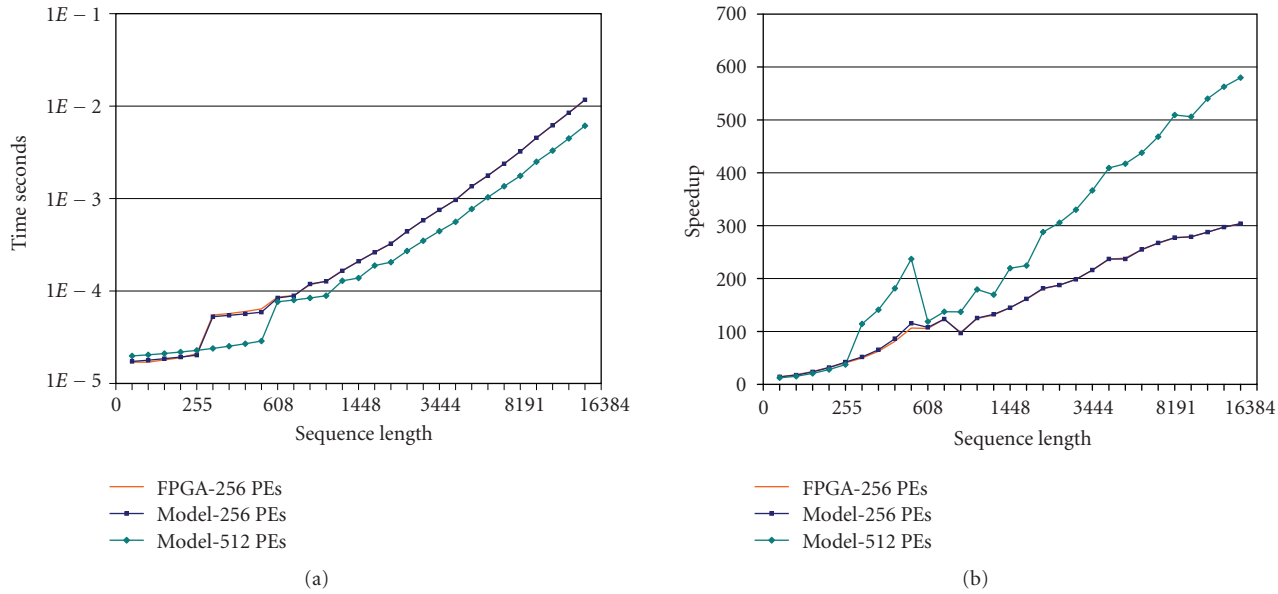


Figure 11: Timing model compared with actual FPGA performance. The model is nearly indistinguishable from the FPGA time. (a) Sequence alignment execution time, (b) speedup relative to the host-only version.

bandwidth at each end of the pipeline, stalls occur infrequently. Sequences enter the alignment module on ports A and B at a rate of 100 MB/s. Concurrently, partial traceback pointers exit port A at 200 MB/s destined for the host, and merged *B*-*H* values exit port B at 400 MB/s destined for the DP FIFO.

Notice that the presented algorithm does not limit the sequence length by the number of PEs or by the amount of full traceback memory. Matching system parameters, such as the number of PEs and the size of traceback memory, to the available FPGA resources maximizes performance. The experimental results and timing model together demonstrate the scalability of the algorithm without memory bandwidth limitations.

8. Conclusion

With the presented algorithm and architecture, long sequences are globally aligned with supercomputing performance on reconfigurable hardware. A speedup over 300 is achieved with the example implementation on FPGA technology when compared to a desktop computer. The architecture is scalable to larger capacity FPGAs for a further increase in performance. Beyond sequence comparison, the full alignment of long sequences is accelerated without memory and I/O bottlenecks through a space-efficient algorithm. After executing traceback in hardware, the accelerator returns a list of edit operations to the host, which constitutes an optimal alignment. Other global alignment acceleration methods only address sequence comparison, limit the sequence length, or exhibit memory and I/O bottlenecks.

The key features of the algorithm are the bounded space requirement for full traceback memory and the reduced space for partial traceback memory. These space reductions enable high-performance alignment of long sequences on a reconfigurable accelerator and are a match for FPGA memory capacities and bandwidth. Only 64 KB of FPGA block RAM is used for full traceback in the demonstrated implementation. Partial traceback data sent to the host at a rate of 200 MB/s is supported by commodity FPGA boards.

Future work includes combining coarse-grain parallel methods [27] with the fine-grain parallelism of this method for multiplied performance gain on reconfigurable computing clusters. Also, the advantages of the presented method are applicable to accelerating local alignment. A general-purpose accelerated alignment library that consists of both local and global methods may be applied to multiple sequence alignment codes with minimal effort.

Acknowledgment

An earlier version of this paper appeared as "Sequence Alignment with Traceback on Reconfigurable Hardware." In *Proceedings of the 2008 International Conference on ReConFigurable Computing and FPGAs (ReConFig'08)*, Pages 259–264, December 2008.

References

[1] C. Macken, H. Lu, J. Goodman, and L. Boykin, "The value of a database in surveillance and vaccine selection," in *Options for the Control of Influenza IV*, vol. 1219 of *International Congress Series*, pp. 103–106, October 2001.

[2] S. N. Gardner, M. W. Lam, N. J. Mulakken, C. L. Torres, J. R. Smith, and T. R. Slezak, "Sequencing needs for viral diagnostics," *Journal of Clinical Microbiology*, vol. 42, no. 12, pp. 5472–5476, 2004.

[3] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, no. 22, pp. 4673–4680, 1994.

[4] C. Notredame, D. G. Higgins, and J. Heringa, "T-coffee: a novel method for fast and accurate multiple sequence alignment," *Journal of Molecular Biology*, vol. 302, no. 1, pp. 205–217, 2000.

[5] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] T. Ramdas and G. Egan, "A survey of FPGAs for acceleration of high performance computing and their application to computational molecular biology," in *Proceedings of the IEEE Region 10 Annual International Conference (TENCON '05)*, pp. 1–6, Melbourne, Australia, November 2005.

[8] P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak, "An applications-focused review of comparative genomics tools: capabilities, limitations and future challenges," *Briefings in Bioinformatics*, vol. 4, no. 2, pp. 105–123, 2003.

[9] F. Delsuc, H. Brinkmann, and H. Philippe, "Phylogenomics and the reconstruction of the tree of life," *Nature Reviews Genetics*, vol. 6, no. 5, pp. 361–375, 2005.

[10] T. Slezak, T. Kuczmarski, L. Ott, et al., "Comparative genomics tools applied to bioterrorism defence," *Briefings in Bioinformatics*, vol. 4, no. 2, pp. 133–149, 2003.

[11] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[12] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Computer Applications in the Biosciences*, vol. 4, no. 1, pp. 11–17, 1988.

[13] D. T. Hoang and D. P. Lopresti, "FPGA implementation of systolic sequence alignment," in *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, H. Grünbacher and R. W. Hartenstein, Eds., pp. 183–191, Springer, Berlin, Germany, 1992.

[14] R. P. Jacobi, M. Ayala-Rincón, L. G. Carvalho, C. H. Llanos, and R. W. Hartenstein, "Reconfigurable systems for sequence alignment and for general dynamic programming," *Genetics and Molecular Research*, vol. 4, no. 3, pp. 543–552, 2005.

[15] T. VanCourt and M. C. Herbordt, "Families of FPGA-based accelerators for approximate string matching," *Microprocessors and Microsystems*, vol. 31, no. 2, pp. 135–145, 2007.

[16] Y. Yamaguchi, T. Maruyama, and A. Konagaya, "High speed homology search with FPGAs," in *Proceedings of the 7th Pacific Symposium on Biocomputing (PSB '02)*, pp. 271–282, Lihue, Hawaii, USA, January 2002.

[17] G. L. Moritz, C. Jory, H. S. Lopes, and C. R. E. Lima, "Implementation of a parallel algorithm for protein pairwise alignment using reconfigurable computing," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig '06)*, pp. 99–105, San Luis Potosi, Mexico, September 2006.

[18] 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.

[19] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith, "Parallel processing of biological sequence comparison algorithms," *International Journal of Parallel Programming*, vol. 17, no. 3, pp. 259–275, 1988.

[20] X. Guan and E. C. Uberbacher, "A multiple divide-and-conquer (MDC) algorithm for optimal alignments in linear space," Tech. Rep. ORNL/TM-12764, Oak Ridge National Lab., June 1994.

[21] R. Lipton and D. Lopresti, "Comparing long strings on a short systolic array," in *Systolic Arrays*, W. Moore, A. McCabe, and R. Urquhart, Eds., pp. 363–376, Hilger, Bristol, UK, 1987.

[22] S. Lloyd and Q. Snell, "Qnet: a modular architecture for reconfigurable computing," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '08)*, pp. 259–265, Las Vegas, Nev, USA, July 2008.

[23] C. Sanderson, "Simplify FPGA application design with DIMEtalk," *Xcell Journal*, vol. 51, pp. 104–107, 2004.

[24] PCI-SIG, "PCI Express," http://www.pcisig.com/.

[25] P. Faes, B. Minnaert, M. Christiaens, et al., "Scalable hardware accelerator for comparing DNA and protein sequences," in *Proceedings of the 1st International Conference on Scalable Information Systems (INFOSCALE '06)*, ACM, Hong Kong, May-June 2006.

[26] Seq-Gen, http://tree.bio.ed.ac.uk/software/seqgen/.

[27] S. Rajko and S. Aluru, "Space and time optimal parallel sequence alignments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 12, pp. 1070–1081, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 639249, 15 pages
doi:10.1155/2009/639249

Research Article

Reaction Diffusion and Chemotaxis for Decentralized Gathering on FPGAs

Bernard Girau,[1] César Torres-Huitzil,[2] Nikolaos Vlassopoulos,[3] and José Hugo Barrón-Zambrano[2]

[1] *LORIA/University Nancy 1, Cortex Group, Campus Scientifique, Vandoeuvre-les-Nancy, France*
[2] *Cinvestav Tamaulipas, Information Technology Laboratory, Victoria, Mexico*
[3] *INRIA Nancy Grand Est, Maia group, Villers-les-Nancy, France*

Correspondence should be addressed to Bernard Girau, bernard.girau@loria.fr

Received 15 March 2009; Accepted 16 November 2009

Recommended by Lionel Torres

We consider here the feasibility of gathering multiple computational resources by means of decentralized and simple local rules. We study such decentralized gathering by means of a stochastic model inspired from biology: the aggregation of the *Dictyostelium discoideum* cellular slime mold. The environment transmits information according to a reaction-diffusion mechanism and the agents move by following excitation fronts. Despite its simplicity this model exhibits interesting properties of self-organization and robustness to obstacles. We first describe the FPGA implementation of the environment alone, to perform large scale and rapid simulations of the complex dynamics of this reaction-diffusion model. Then we describe the FPGA implementation of the environment together with the agents, to study the major challenges that must be solved when designing a fast embedded implementation of the decentralized gathering model. We analyze the results according to the different goals of these hardware implementations.

1. Introduction

Spatial computing is a large research field where researchers try to propose alternative computing devices that consist of a (huge) amount of computational resources that are spread across some physical structure. This research field includes many different domains, such as biological computation, robot swarms [1] and swarm intelligence [2], amorphous computing [3], and reconfigurable computing. The common constraints are that the communication cost between different computational resources strongly increases with their distance, and that the global functionality emerges from the collective behavior of the resources. In this research field, the main algorithmic question may be summarized as "how to make computing units cooperate to solve a given task?" [4]. Our project (*Amybia* INRIA collaborative research project, led by Nazim Fatès) takes place in this approach, by considering an upstream question, "how to gather or assign enough computing resources to solve a given task?", in a context where faulty units may appear.

More precisely, we consider here the problem of gathering computing units in a strongly constrained context: (1) units use only local rules, (2) units move on a lattice and need to gather to form a compact cluster, (3) units have no idea of their own position and of the position of the other units, (4) units may only send messages that can be relayed (possibly with errors) by the cells of the lattice, (5) units only perceive the state of the neighboring cells, and (6) the only action units may undertake is to move to these cells or change the state of these cells. The possible applications of this problem to several problems of spatial computing still need to be deepened: we discuss them in Section 7. In this paper, our ambition is only to show that a simple model is able to achieve decentralized gathering, while being suitable for efficient distributed implementations. Our approach is inspired by biology, where such decentralized gathering is observed, so as to derive a model and its implementation.

The cellular slime mold Dictyostelium discoideum is a fascinating living organism that has the ability to live as a monocellular organism (amoeba) and to transform

into a multicellular organism when needed. In normal conditions, the amoebae live as single individuals. However, when the environment becomes depleted of food, a gathering phenomenon is triggered and single amoebae aggregate to form a complex organism that moves and reacts with coordinated changes. In the Amybia project, we take inspiration from the first stage of the multicellular organization process, the aggregation stage, which consists in gathering all the amoebae in a compact mass called a mound [5].

In [6, 7], Fatès proposes a simplified model of Dictyostelium discoideum that exhibits the main behavioral properties of the aggregation mechanism: reaction-diffusion and chemotaxis. Our Amybia project is built around this model. It uses a cellular automaton to describe the environment, and a multiagent approach to model the amoebae. This paper focuses on the hardware aspects of our project. It is roughly divided into two parts. In the first part, the FPGA is used as an accelerator for simulations of the dynamics of the environment, especially close to phase transitions. In the second part, beyond being an accelerator, the FPGA is also considered as a representative device for massively distributed computation so as to study the main issues that may appear while using reaction-diffusion chemotaxis for decentralized gathering within heterogeneous spatial computing devices.

Section 2 focuses on the biological inspiration of our work, describing the aggregation process that is observed with Dictyostelium discoideum. The model of [6, 7] consists of an environmental layer and a particle layer. The environmental layer and its properties are described in Section 3, while its hardware implementation for fast simulations is depicted in Section 4. Then Section 5 summarizes the particle layer model and its properties before Section 6 describes its FPGA implementation that performs decentralized gathering on-chip. Section 7 shortly discusses possible contexts of use for such decentralized gathering. Finally, we derive conclusions about the main obstacles and possible modifications of our approach.

2. Decentralized Gathering and Bioinspiration

Decentralized algorithms to gather robots to form circular or polygonal shapes have been proposed in [8], where all robots "see" the positions of each other. Similar problems with a limited visibility range have been studied in [1, 9]. We refer to the work of [10] for recent developments on the decentralized gathering problem. In this paper, we get rid of any assumption on the visibility range using an environment that transmits messages on arbitrarily long distances. The decentralized gathering problem is also related to the Leader Election problem, but our goal is not to select one unit among many, but to gather randomly located units in a compact location that emerges by consensus. This behavior is part of the complex and unusual life cycle of the cellular slime mold Dictyostelium discoideum; it corresponds to the aggregation step of the multiple monocellular organisms that gives birth to a single multicellular organism.

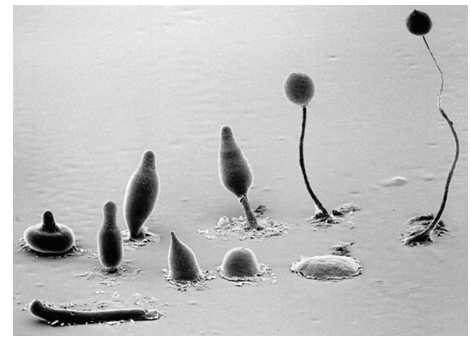


Figure 1: Some steps of the life cycle of *Dictyostelium discoideum* (Copyright, M.J. Grimson & R.L. Blanton, Biological Sciences Electron Microscopy Laboratory, Texas Tech University).

2.1. Dictyostelium discoideum. Despite the biological inspiration of our work, we do not pretend to provide the reader with accurate and complex biological notions. Some biological terms may even be used with some approximation, which does not penalize our work, since we model a behavior and we do not model biological species. Therefore this section only gives an overview of the specificities of *Dictyostelium discoideum*, focusing on its aggregation properties that inspire the model in [6, 7].

2.1.1. Life Cycle. Dictyostelium discoideum amoebae grow as independent cells in natural environments such as moist, decaying wood. In normal conditions, they behave as monocellular organisms, but they are able to interact when a coordinated reaction to extreme conditions is required. Extreme conditions may correspond to a food-depleted environment that might result in starvation for the population of amoebae. By means of their interactions, single cells do not only join to perform a collective reaction, they join to generate a multicellular organism (containing thousands of cells) that is able to better react to extreme conditions than the population of individual cells.

As illustrated by Figure 1, after having grouped together, the population becomes able of cell differentiation, which results in several steps of a life cycle that adapts to the environmental conditions. The mound of cells that results from aggregation is then transformed into some elongated migrating slug, and then into a fruiting body. We are only interested in the process by which amoebae group together, since it fulfills the different constraints for the decentralized gathering of computing units we study.

2.1.2. Aggregation. In vitro experiments show that the aggregation phenomenon of Dictyostelium is triggered by one or several amoebae that attract other amoebae that are located in their vicinity to form groups. The first groups merge until only a few clusters remain; these will attract other amoebae to them to form a cluster where cellular differentiations occur to lead to the multicellular organism.

Attraction is led by the transmission of waves of chemical messengers, which follow typical evolving reaction-diffusion

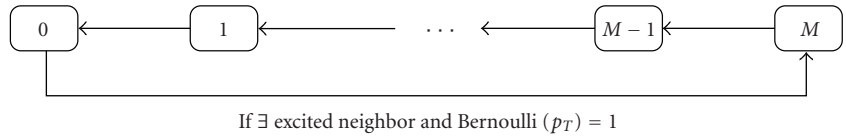


Figure 2: Cycle of states for a cell of the environmental layer.

patterns. The chemical messengers are internally produced by the amoebae. When an amoeba detects a high increase in the external concentration of the messengers, it follows the concentration gradient (this phenomenon is called chemotaxis) and it releases its own internal messengers. Then it becomes insensitive to the messengers during a given refractory period, and in the meanwhile, the released messengers diffuse and attract other sensitive amoebae.

2.2. Previous Models. Several models have been proposed to study the dynamics of Dictyostelium (see a review in [11]). Many of them are based on partial differential equations [12, 13]. Some studies aim at being very close to the biological inspiration, comparing simulation outputs with observations of the aggregation of Dictyostelium [14], or modeling the receptors of the chemical messengers [15]. Most studies use continuous or hybrid models; to our knowledge, the model in [6, 7] that founds our Amybia project is the first fully discrete model that captures Dictyostelium's behavior. By fully discrete, we mean that time and space are discrete and the state of the amoebae is described in qualitative terms rather than quantitative (integers or decimal values). This discretization is useful when digital hardware implementations are expected. The reaction-diffusion mechanism alone is well understood, with explicit links between the discrete and continuous models (e.g., [16]). This mechanism shows problem-solving abilities [17]. In our project, we use the model of Fatès [6, 7] that adds virtual chemotaxis as a new feature to study and use. Two layers compose it: the environmental layer is a cellular automaton that models a reaction-diffusion process while the particle layer describes the moves of virtual amoebae.

3. The Environmental Layer

As explained before, attraction of amoebae is led by the transmission of waves of chemical messengers in the environment. In this section, we only consider this reaction-diffusion process. The study of the qualitative behavior of the environmental layer is an important part of the Amybia project. We aim at characterizing this behavior in terms of complex system dynamics, and we study its robustness to noise and obstacles. Therefore we assume here that waves of excitation are initiated at randomly chosen positions, and then we observe how these waves behave in the long term.

Next subsection defines the discrete model of this environmental layer [6, 7], that mostly depends on one parameter called the transmission rate. Then Section 3.2 summarizes the main results about the dynamics of this reaction-diffusion process, where phase transitions depend on this transmission rate. The goal of the implementation described in the next section is to perform large scale simulations of those phase transitions, and to extend these results to various topologies and perturbations.

3.1. Discrete Model. Space is modelled by a regular lattice $\mathcal{L} = \{1, \ldots, X\} \times \{1, \ldots, Y\}$ in which each cell $c = (c_x, c_y) \in \mathcal{L}$ is associated to a state. The set of possible states for each cell is $\{0, \ldots, M\}$, the state of cell c at time t is denoted by σ_c^t. State 0 is the *neutral* state, state M is the *excited* state.

A "source" cell is an initially excited cell. Any cell may evolve from the neutral state to the excited state if at least one of its neighbors is excited (rule (R1). To model the uncertainty on this transition, we consider that it happens with a given probability p_T, called the *transmission rate*. States 1 to $M-1$ are the *refractory* states. A cell in a refractory state evolves in an autonomous way by decrementing its state by 1 (rule (R2) until it reaches the neutral state. A neutral cell surrounded by neutral cells stays neutral (rule (R3). Figure 2 illustrates the different possible states of the cell.

To express these rules without ambiguity, for a cell $c \in \mathcal{L}$, let us denote by N_c the neighborhood of this cell. Let E_c^t be the set of excited cells in the neighborhood of c at time t: $E_c^t = \{c' \in N_c | \sigma_{c'}^t = M\}$. We also denote by $|S|$ the cardinal of a set S.

With these notations, for a time $t \in IN$ and a cell $c \in \mathcal{L}$, let $\mathcal{B}(p)$ be a Bernoulli random variable that equals 1 with probability p and equals 0 with probability $1 - p$. The local rule governing the evolution of the environment is

$$\sigma_c^{t+1} = M \quad \text{if } \sigma_c^t = 0, \left|E_c^t\right| > 0, \mathcal{B}(p_T) = 1, \tag{R1}$$

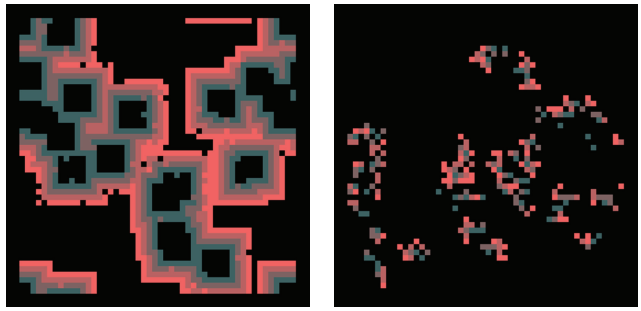
$$\sigma_c^{t+1} = \sigma_c^t - 1 \quad \text{if } \sigma_c^t \in \{1, \ldots, M\}, \tag{R2}$$

$$\sigma_c^{t+1} = 0 \quad \text{otherwise.} \tag{R3}$$

A set of adjacent cells that are all in the excited state M is called an excitation front. In Section 5, we explain how the excitation fronts guide the amoebae that move on the lattice (chemotaxis).

3.2. Properties. The main properties of this model are presented in [6, 7]. Since this paper focuses on the hardware implementation issues, we only summarize the main results below.

The dynamics of the environment depends on two parameters: the excitation level M and the transmission rate p_T. The study of [6, 7] shows that different qualitative behaviors may be observed: the static regime, the non-coherent regime, and the extinction regime.



(a) (b)

Figure 3: Two aspects of the noncoherent regime (a): $p_T = 0.95$, (b): $p_T = 0.25$).

3.2.1. Static Regime. This regime is obtained in the case of systematic transmission of waves ($p_T = 1$): the excitation fronts collide systematically and they annihilate themselves. This phenomenon is well known for reaction-diffusion processes.

3.2.2. Noncoherent Regime. This regime may be observed in the case of nonperfect transmission conditions ($p_T < 1$); the reaction-diffusion waves are independent from the position of the source cells and no organization can occur. Figure 3 illustrates the influence of p_T on the transmission of waves, in a 50×50 environment with $M = 4$ and a Moore neighborhood. Black pixels denote neutral cells, and red pixels stand for excited cells, while shaded colors are used for refractory states. Reaction-diffusion waves remain visible with $p_T = 0.95$; whereas they appear unorganized with $p_T = 0.25$.

3.2.3. Extinction Regime. This regime is attained when the transmission rate is less than a critical value ($p_T < 0.23$ for $M = 3$). In that case, waves spontaneously disappear.

Following well-known studies in statistical physics, the first experiments depicted in [7, 18] indicate that the universality class of the phase transition from the non-coherent regime to the extinction regime might be *directed percolation* [19, 20]. The robustness properties of the model strongly depend on the universality class of its phase transitions. These experiments need to be extended to larger environments, but software simulations are very time-consuming. Therefore we have developed a block-synchronous hardware implementation to handle large-scale simulations.

4. Fast FPGA Simulation of Phase Transitions

The hardware part of the Amybia project is motivated by two main goals. The first one is to develop fast implementations to explore complex dynamics in large-scale environments. The corresponding implementation work is described in this section. The second goal is to perform a preliminary study of the ability of our model to provide an efficient decentralized gathering process for a large amount of distributed computing units. The corresponding implementation is the subject of Section 6.

4.1. Block-Synchronous Implementation. The behavioral description of each environment cell may reduce to a very simple state machine that could be implemented with very few hardware resources. Nevertheless, the most area-greedy computation in the environment layer is not the state transition, but the generation of the Bernoulli law with probability p_T for each cell. As a consequence, a fully parallel implementation of the environmental layer would not be able to implement environment sizes that are out of reach for software simulations (a few thousands of cells at most, see below for the implementation area of the random generators). Therefore, we have chosen to use a block-synchronous (or block-parallel) implementation based on the embedded B-RAM memories (Block RAM) of the FPGA, as in [21]. The environment is partitioned into several blocks, with each block of cells being handled in a fully parallel way by the FPGA while the different blocks are sequentially handled. Let $X \times Y$ be the total number of cells in the environment layer. Let $n \times m$ be the number of cells that may be simultaneously handled on the FPGA. The environment is partitioned into $\mathcal{B} = (X/n) \times (Y/m)$ blocks of $n \times m$ cells.

Let us consider a cell in the environment. It is located at relative position (x, y) in block (α, β), so that its coordinates in the whole environment are $(\alpha n + x, \beta m + y)$. We store its state in a local B-RAM memory, with an address that corresponds to the block number. The local position of the B-RAM memory is sufficient to stand for the (x, y) coordinates, so that they do not appear in the address. The computation of the block is performed by using the same block-dependent address for all local B-RAM memories, thus handling all cells in this block. Then the computations are performed for the next blocks by increasing the common address used for all B-RAM memories. It should be pointed out that the choice of B-RAMs to store cell states in this implementation is only related to the need to store many states at the same location in the FPGA; whereas the cell states will be stored in simple elementary flip-flops when considering the fully parallel implementation of the model with amoebae in Section 6.1.

Figure 4 illustrates the decomposition of the environment into blocks and the block-synchronous scheduling of the computation. The different blocks are shown, each one containing an outlined cell at relative position (x, y): the

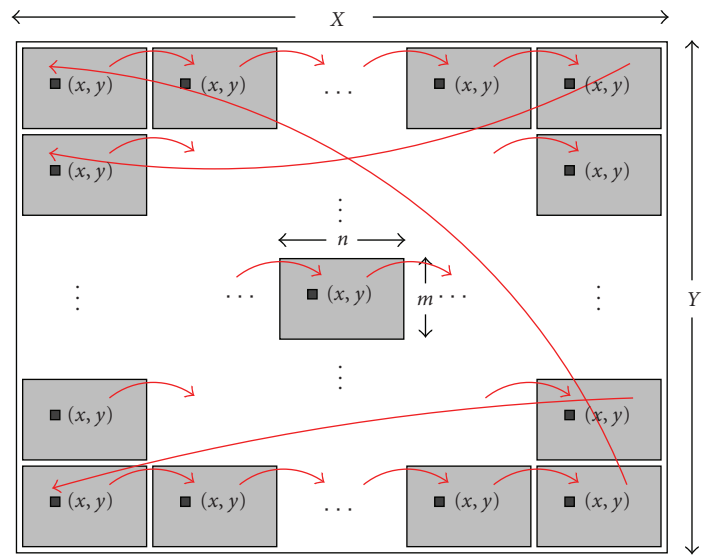


Figure 4: Block partitioning of the environmental layer.

states of all these outlined cells are stored in the same B-RAM memory of the FPGA. All cells are handled simultaneously in a given block, and the red arrows denote the cyclic block-scheduling of the computation.

4.2. General Architecture. Figure 5 schematizes the general architecture of our implementation of the environmental layer. Since the environment is split into several blocks, this architecture mostly consists of a grid of $n \times m$ identical cell modules (gathered as groups of 4 or 6 cells using the same B-RAM memory to handle on-chip data storage and access) surrounded by border modules. An additional control module computes the memory addresses that are used by all modules (block-scheduling) and computes the number of excited cells in the environment. Figure 5 only illustrates a simple 4-neighborhood, but our implementation handles the 8-neighborhood. The role of each component is as follows.

4.2.1. Cell Module. Each cell module updates the state of its corresponding cell within the currently handled block. More precisely, we use a bufferized storage of the states of all cells so as to synchronize the computations of all blocks: a most significant bit δ or $1 - \delta$ is added to the addresses that are sent to the dual-port B-RAM memories; the current states are read with δ, whereas updated states are stored with $1 - \delta$. When the current iteration of the equations of rules (R1), (R2), and (R3) has been performed for all blocks, buffers are exchanged by means of $\delta \leftarrow 1 - \delta$.

Depending on the value of M, the cell modules are split in groups of 2×2 cells ($M < 16$) or 2×3 cells ($M < 8$) that share common storage resources: a single 18 Kbit dual-port B-RAM memory stores the states of 4 or 6 cells for all blocks of cells, using 18-bit words.

4.2.2. Border Module. The border modules are simpler than the cell modules. They only store one bit for each one of the immediate neighbors of the most outer cells within each block; this bit stands for the cell being excited or not. The only difficulty is to handle the addressing scheme so that the information stored within each of the 4 possible borders is updated when the block that contains the corresponding cells is being handled. This update requires long-range connections from the cell modules on each side of the block to the opposite border modules. Moreover, when the borders lie outside the whole environmental layer, the border modules simply generate the constant value 0 (not excited).

4.2.3. Control Module. The control module uses a 10-bit counter to perform block-scheduling, and a 16-bit counter to handle iterations on the environment. Moreover, our goal is to study the phase transition between the non-coherent regime and the extinction regime. Therefore, the control module computes the number of excited cells within each row of cells, it adds these numbers for all rows, and then it accumulates the results for all blocks. Nevertheless, it is sufficient to detect if the number of excited cells tends to zero, so that all numbers are computed up to 64, which reduces the cost of the adders.

4.3. Implementation of a Cell. A cell module mostly consists of two parts: a random number generator (RNG) to compute the Bernoulli random variable $\mathcal{B}(p_T)$, and the cell state update.

4.3.1. Generating the Bernoulli Law. In the software implementation developed by Fatès, the same RNG is used for all cells, thanks to the assumed independence of the successively generated numbers. It should be pointed out that generating high-quality random variables to ensure a real

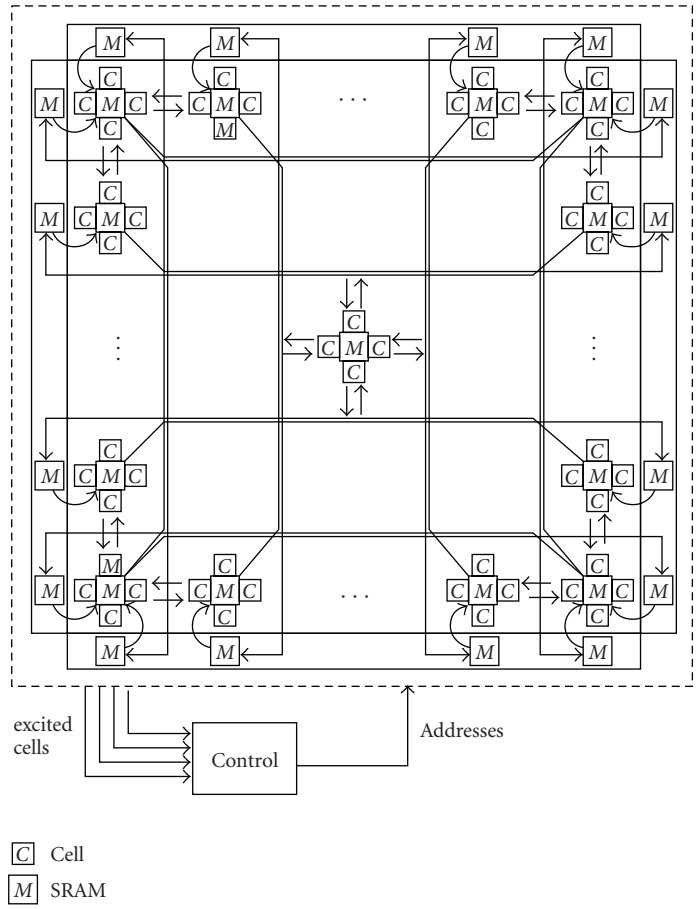


Figure 5: General architecture (environment).

independence of successive random numbers still remains a research subject. Nevertheless, this issue of software RNGs does not appear as relevant for our model, where the quality of usual RNGs is sufficient to break the symmetry of wave transmissions (see [6, 7]). But in a parallel hardware implementation, all cell modules must generate their own random variable in parallel. Therefore we have to implement $n \times m$ RNGs. The precision of the random processes is particularly important when studying phase transitions. Moreover, the spatial independence that is required for symmetry breaking implies that the hardware RNGs we use must be sufficiently good (long period, and independent seeds). This induces an important cost in space for the random aspects of the environmental layer. This hardware resource cost is the counterpart of the computation time that is mostly spent in generating random numbers in the software implementation.

Our choices for the implementation of the random processes have been carefully studied. Most digital hardware solutions are based on LFSR or cellular automata (CA) [22]. Another approach takes advantage of large numbers stored in parallel [23]. Since the LUTs of the FPGA logic cells may be efficiently configured as synchronous RAMs standing for shift registers [24], we use LFSR-based RNGs. See Section 8 for an extension of our work to spatially distributed and mutualized CA-based RNGs.

Experiments in [6, 7] show that the transmission rate needs to be taken into account with a rather high precision (more than 16 bits). Taking into account this precision and the need for random bitstreams as aperiodical as possible, we use a 168-bit RNG adapted from [24] (a similar 63-bit RNG will be depicted later in Figure 14), comparing at least 16 of its generated bits to p_T so as to output 0 or 1 as $\mathcal{B}(p_T)$. To ensure spatial independence, all RNGs use different seeds (set during initialization through on-chip registers).

4.3.2. Updating the Cell States. Figure 6 shows the simplified architecture of a cell module. The current state of the cell is read in the local B-RAM memory. It is compared to M so as to send to the neighboring cells a signal that is equal to 1 if the local cell is in the excited state. The current state is decreased (if excited or refractory) by the "state decrease" module, while a large AND gate outputs 1 if the state is neutral and if there is at least one excited neighbor and if the Bernoulli law generator currently outputs 1. A final multiplexer chooses between M and the computed decreased state according to the output of the AND gate. The resulting value is written in

Table 1: Synthesis results for a single cell.

Single cell hardware resource utilization Xilinx FPGA XC4VLX160ff1513-12	
Number of Slice Flip Flops	24/135,168
Number of 4 input LUTs	53/135,168
Number of occupied Slices	44/67,584
Max Frequency	143 MHz

Table 2: Synthesis results for a block of cells.

32×32 block hardware resource utilization (16×16 groups of 2×2 cells) Xilinx FPGA XC4VLX160ff1513-12	
Number of Slice Flip Flops	25,397/135,168 (18.79%)
Number of 4 input LUTs	54,315/135,168 (40.18%)
Number of occupied Slices	39,728/67,584 (58.78%)
Frequency	100 MHz

the local B-RAM memory. It must be noticed that the storage of the states in the B-RAM memories makes it impossible to implement the cell as a simple finite state machine (unlike the implementation in Section 6).

4.4. Performance

4.4.1. Implementation Results. The prototyping platform is a PCI-based board (DN8000K10PCI) with three virtex-4 family FPGAs. For experimental results, the FPGA implementation of the model is only targeted towards the XC4VLX160fff1513-12 device of this board. This FPGA has a capacity of 135, 160 logic cells, and it contains 288 embedded 18 Kbit B-RAM memories. The design was synthesized, placed, and routed with the Xilinx Foundation ISE 9.2i tool suite. According to the reported synthesis results in Table 1, a compact implementation was obtained since a single cell requires 44 slices. It is important to point out that these resources take into account the implementation of the 168-bit RNG adapted from [24] which was efficiently implemented as a LFSR using FPGA shift register LUT primitives.

As summarized in Table 2, a block of 16×16 groups of 4 cells only requires around 59% of the total logic resources available in the FPGA device, taking advantage of the optimization of the slices that are partially used by a single cell. The size of the grid, 1024 cells, is limited by the amount of embedded distributed B-RAM memories in the FPGA. For this grid size, 256 B-RAM memories are used since the 4-bit states (we consider here the case $M < 16$) for the 4 adjacent cells of a group in a block are stored in the same memory.

In order to achieve large-scale efficient simulations, larger grid sizes are desirable, corresponding to interesting experimental environments. Therefore, this block implementation is used as the basic computational unit for each part of the partitioned environment. Only 8 additional B-RAM memories are required to store the excitation states of the $2 \times (32 + 32)$ border cells in the border modules. Therefore, 264 out of the 288 B-RAM memories of the XC4VLX160 are used. Finally, the module that controls the computation scheduling of all blocks and that accumulates the number of excited cells found in each block uses less than 2% of the logic resources, so that the whole architecture uses 60,5% of the FPGA resources.

4.4.2. Fast Large-Scale Simulations. The embedded B-RAM memories are able to store the states of 512 groups of 4 cells (with state buffering). Therefore we implement a total size of 512×1024 cells for the environmental layer. Despite this large size, we still have to face border effects when studying phase transitions in these environments. Therefore, we take advantage of the methodology inspired by [25] so as to study phase transitions only at the limit of the stable state (where all cells are neutral): all cell states are initially set to neutral, except the central cell of the central block, that is initially excited.

We estimate here the simulation speedup of our FPGA implementation with respect to the software simulation tool developed by Fatès. These estimations should be considered with great caution, since the software tool and the hardware implementation are difficult to compare: this software is a not-optimized version written in Java with jdk 1.6; moreover, the hardware and software computations are not fully equivalent (considering the way random numbers are generated). Therefore, we consider that the computed speedup should only be interpreted in terms of order of magnitude. It should be noted that unlike the widely spread idea that Java is slow, recent benchmarks show that Java 1.6 easily compete with C, C#, or C++. Yet, it is not possible to extrapolate this comparison to a software with cache optimization or similar improvements, for which the performance improvements might be great, but highly dependent on the application. For a 512×1024 environmental layer, Java-based software simulations on a microprocessor-based computer, Pentium 4.2 GHz, require 0.5 s per evolution step, resulting in very long experiments (thousands of iterations are required for each run, and thousands of runs are required to reach significant statistical results for each value of p_T). The computation time mostly lies in both the generation of the random values and the cache management, because of the huge number of cells.

With the above FPGA implementation, each iteration lasts 512 clock cycles (number of blocks), so that the observed speedup is $\Omega(10^5)$. Beyond this order of magnitude that might be reduced if an optimized software was designed, the important result is that experiments that are obviously not within our grasp with a software approach may be easily performed on the FPGA (some tens of seconds being sufficient to have a valuable statistical estimate of the behavior of the system for a given set of parameters).

Finally, we mention the fact that many experiments handle values of M lower than 7, and the most up-to-date FPGAs (XC6VLX160) contain up to 720 embedded 36 Kbit B-RAM memories (each B-RAM being able to store the states

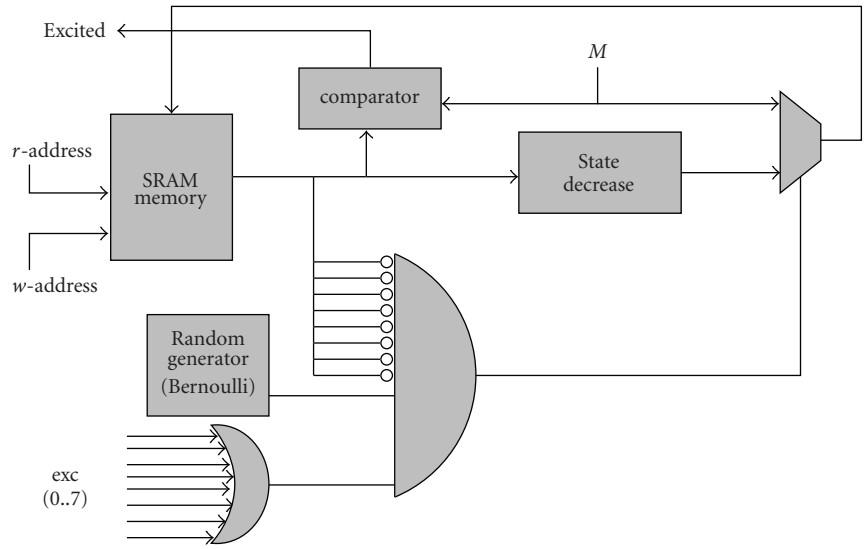


Figure 6: Implementation of a cell of the environmental layer.

of a group of 3×4 cells). Therefore our implementation might scale up to more than 4 300 000 cells (the logic resources utilization rate remaining markedly below 100%).

5. The Particle Layer

When restricted to the environmental layer, the model of [6, 7] only takes advantage of a reaction-diffusion mechanism. We are mostly interested in the decentralized gathering that occurs when amoebae are subject to a chemotaxis process. We now focus on these amoebae that are modeled by agents.

5.1. Discrete Model. The amoebae are supposed to be all identical, and in constant number as no birth or death process is considered. Several amoebae may be located at the same cell. We arbitrarily allow only one amoeba to move from a nonempty cell at each time step. We do not limit the number of amoebae that can simultaneously move to a given cell, but we arbitrarily choose to allow an amoeba to go on a neighboring cell only if this cell contains less than two amoebae [5–7]. Let us define a cell that contains no amoeba as an *empty* cell, and a cell that contains strictly less than two amoebae as a *free* cell. The movement rules state that, at each time step, for each non-empty cell, one single amoeba may

(i) move to an adjacent free cell (rule (R4),

(ii) move to an adjacent excited free cell (rule (R5),

(iii) stay on the same cell (rule (R6).

To apply rule (R4) (noise rule), we consider that each non-empty cell may send an amoeba to one of its neighbors with probability p_A, called the *agitation rate*. This neighbor is randomly selected among all neighbors that are free. Similarly, to apply rule (R5) (chemotaxis rule), amoebae move to a cell that is randomly selected among the excited free cells of the neighborhood. Rules (R4) and (R5) are made mutually exclusive. Formally, for $t \in IN$ and $c \in \mathcal{L}$, let $\widetilde{N}_c^t$, respectively, $\widetilde{E}_c^t$, be the set of *free* cells, respectively *excited free* cells, in the neighborhood of c. For a finite set S, we denote by $\mathcal{R}(S)$ the operation of selecting one element in S with uniform probability, with the convention $\mathcal{R}(\emptyset) = \emptyset$. $\mathcal{R}$ randomly selects a neighbor for moving. We use a Bernoulli function to impose noise on the moves of an amoeba with probability p_A. To represent the move of one amoeba from a *non-empty* cell c to another cell Δ_c^t, with the convention $\Delta_c^t = \emptyset$ if no move occurs, we have

$$\text{if } \mathcal{B}(p_A) = 1, \quad \text{then } \Delta_c^t = \mathcal{R}\left(\widetilde{N}_c^t\right), \tag{R4}$$

$$\text{else if } \sigma_c^t = 0, \quad \text{then } \Delta_c^t = \mathcal{R}\left(\widetilde{E}_c^t\right), \tag{R5}$$

$$\text{else } \Delta_c^t = \emptyset. \tag{R6}$$

5.2. Coupling of Environment and Particles. Amoebae act on the environment by emitting excitations that propagate to neighboring cells. We do not take into account the number of amoebae contained in each cell; a non-empty neutral cell may become excited with probability p_E called the *emission rate*. Since this rule may interfere with rule (R1), we combine both rules into rule (R1'):

$$\sigma_c^{t+1} = M \quad \text{if } \mathcal{B}(p_T) = 1,\ \sigma_c^t = 0,\ \left(\left|E_c^t\right| > 0 \text{ or } (c \text{ non-empty and } \mathcal{B}(p_E) = 1)\right). \tag{R1'}$$

5.3. Properties. Similar regimes may be observed as in Section 3.2: the non-coherent regime ($p_T < 1$), the extinction regime (p_T less than a critical value that depends on M), and the static regime, that is obtained in the case of systematic transmission of waves ($p_T = 1$) and if amoebae constantly initiate wave fronts ($p_E = 1$). In the static regime, excitation fronts collide systematically, so that amoebae are

not attracted by each other (no move can occur, since no information may be exchanged between different amoebae).

The most promising behavior, the *self-organizing regime*, is observed when the transmission is perfect and when the emission rate is less than 1 (typically $p_E = 0.1$) and for various values of agitation rate. In this regime, a gathering phenomenon shows a progressive merging of the amoebae from small clusters to large clusters, after a few tens to a few thousands of iterations (depending on the environment). The complexity of this hierarchical dynamics results from successive emerging behaviors: formation of waves, formation of first groups, extension and shrinking of the regions according to their respective size, and captures of small clusters by a few clusters. Among interesting properties observed in the system, Fatès has shown that gathering could also occur in the presence of obstacles as the virtual amoebae could take advantage of narrow corridors to find their way to an attracting cluster [6, 7].

Figure 7 illustrates the resulting aggregation and the propagation of waves (simulated by the software implementation developed by Fatès) in a "perfect" environment. Figure 8 illustrates the same phenomenon in an environment with both obstacles and noise. Purple pixels are the amoebae, green pixels are obstacles, excited and refractory cells are drawn with shaded orange colors, and neutral cells are white. The behavior of the model satisfactorily reproduces the aggregation properties of Dictyostelium discoideum, while fulfilling all required constraints. Moreover, the decentralized gathering appears as robust to noise and irregular topologies. For further details and illustrations about the model dynamics and its self-organizing regime, see [6, 7].

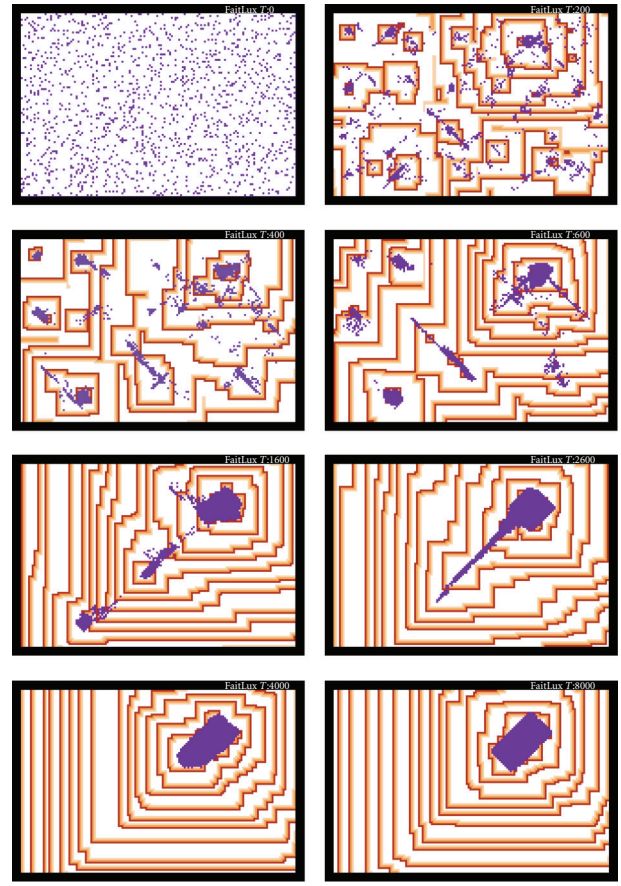


Figure 7: Aggregation of amoebae (purple) and propagation of waves (shaded orange) in a "perfect" environment (steps 0, 1, 2, 4, 10, 15, 20, and 40).

6. Hardware Implementation of the Model

Following the study of the dynamical behavior of the model in [6, 7], we also set $p_T = 1$ since aggregation only occurs with perfect transmission. From now on, we arbitrarily use the 8-neighborhood, and we set the excitation level to $M = 3$.

6.1. Cell and Amoebae Implementation. For implementation purposes, we define a *node* as a cell together with the amoebae it contains. The moves of amoebae may be simply described as the evolution of the "population" of each node as a part of its internal state. Figure 9 shows the I/O of the node module.

6.1.1. State of a Cell. Considering the environmental layer only, the state of each cell belongs to $\{0, 1, 2, 3\}$, so that we code it with two bits (`s1, s0`). This state evolves according to rules (R1'), (R2), and (R3). These rules may be expressed as the state machine depicted in Figure 10. Signal `pE` stands for $\mathcal{B}(p_E)$. Input `not_empty` codes for the presence of at least one amoeba in the node (it is an internal signal generated by the module that codes the population of the node).

6.1.2. Population of a Node. Considering the amoebae, they are coded as the number of amoebae that are located in the cell that corresponds to the local node. Amoebae may move towards free cells only. Free cells contain at most one amoeba. Since up to 8 amoebae may simultaneously move towards a free cell, each node contains at most 9 amoebae. Instead of coding the population size (using 4 bits and counting at each time the number of arriving amoebae), we use 9 flip-flops: though less compact in terms of number representation, this solution does not require coding and counting resources, so that it uses significantly less logic cells. The first flip-flop stores "1" if there is at least one amoeba. Then the 8 other flip-flops directly receive arriving amoebae. Each time an amoeba leaves the node, one of the flip-flops storing "1" is reset to "0" (the reset command is transmitted among flip-flops until finding a "1"). Similarly, if amoeba arrivals occur when the cell is empty, then the first flip-flop is set to "1" and one of the other flip-flops is reset to "0". Figure 11 depicts the resulting architecture to store the node population. The node indicates whether it is free or not with signal `free_out`.

6.1.3. Amoeba Moves. Figure 12 shows how the moves of the amoebae are implemented (rules (R4), (R5), (R6)). Signal `pA` stands for $\mathcal{B}(p_A)$. It controls 8 multiplexers (one for each neighbor) that indicate whether the corresponding neighbor is free or excited and free. Moreover signal `neutral` is used in the second case (R5). It is internal and it codes for $\sigma_c^t = 0$,

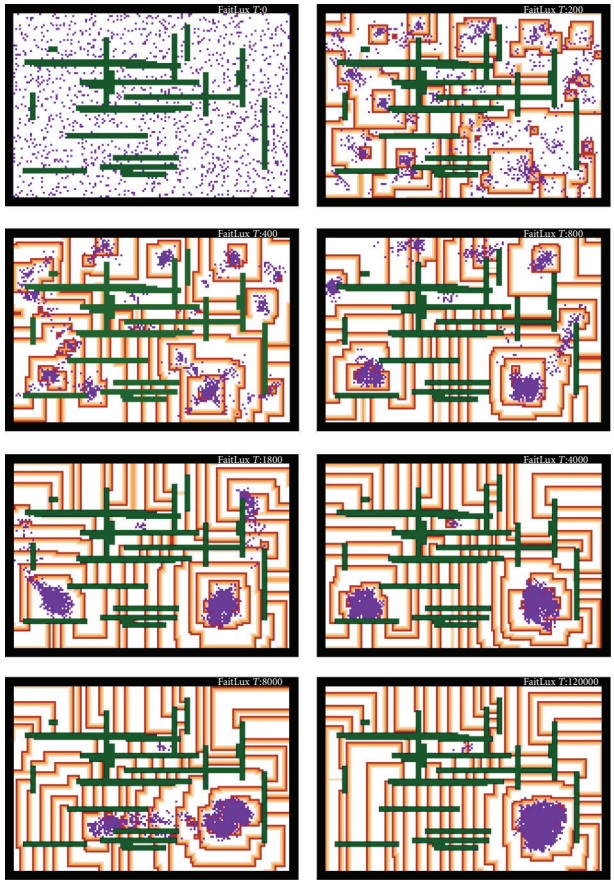


FIGURE 8: Aggregation of amoebae (purple) with obstacles (green) and noise (steps 0, 1, 2, 4, 10, 20, 40, and 60).

that is, $(s1, s0) = (0, 0)$. In the same way, if signal `not_empty` is "0" then all choices are set to "0" because no amoeba may move if the cell is empty. Then the `Select` module randomly selects only one choice among possibly several. Finally an `OR` gate determines if an amoeba will move while the bus `am_towards` indicates where it will move.

The random selection of a signal set to "1" among possibly several is complex. In our implementation, we use a cyclic priority module, where the main priority is given to a signal that is randomly specified by three bits provided by a linear feedback shift register (LFSR), as shown in Figure 13. This implementation suffers the following drawbacks: (1) it is not uniformly random, and (2) though it is fair, it introduces some systematic bias in the selection of close signals because of the cyclic priority. Nevertheless, first experiments indicate that these drawbacks do not modify the overall behavior of the model.

6.2. Random Processes. The definition of the model includes several random aspects: $\mathcal{B}(p_T)$, $\mathcal{B}(p_A)$, $\mathcal{B}(p_E)$, $\mathcal{R}(\widetilde{N}_c^t)$, $\mathcal{R}(\widetilde{E}_c^t)$. The software implementation uses the same RNG for all nodes and for all Bernoulli laws (see Section 4.3 for a discussion about the required RNG quality). But in this parallel hardware implementation, all streams of stochastic bits must be generated by separate modules, in each node. Since we consider the case, where $p_T = 1$, and since our random selection module (used for both $\mathcal{R}(\widetilde{N}_c^t)$ and $\mathcal{R}(\widetilde{E}_c^t)$) just needs a single LFSR, we finally have to implement $3\ X \times Y$ RNGs. As for the environmental layer, the cost in space for all random aspects of the model is huge.

We choose again to adapt the LFSR-based RNGs of [24]. Experiments in [6, 7] show that both emission and agitation rates do not need a high precision. Therefore we use two 63-bit RNGs adapted from [24], comparing only 8 of their generated bits to p_E and p_A (coded on 8 bits). All RNGs use different seeds (set serially during initialization). Figure 14 depicts one of the used RNGs (to choose other irregularly extracted bits, one has just to use other arrangements of SRAMs and flip-flops, and pick the 8 signals at different places).

The selection module uses a 3-bit random counter to define the main priority choice. Since all 3 bits must be simultaneously accessed, 3 flip-flops are required. Instead of only using 3 bits for the random counter (resulting in an 8 cycle periodicity), we use here an adapted version of the 15-bit random counter of [24] that only needs 3 logic cells to strongly increase the periodicity without requiring more resources.

6.3. General Architecture. Figure 15 describes the general architecture of our implementation. It consists of a grid of 40×30 identical nodes. Border nodes receive constant inputs from their nonexisting neighbors (`exc_in`, `free_in`, and `cell_in` are set to "0" for these nodes).

6.3.1. Initialization. In this implementation, the user defines the desired average number of amoebae. Then the induced ratio (number of amoebae/number of cells) is sent at run time to all nodes, that use it in combination with their 63-bit RNG (threshold compared with the 8 extracted bits), so as to decide whether they initially contain an amoeba or not. This initialization scheme avoids the resource consumption of the large demultiplexer that is required when an external memory defines the exact initial positions of the desired amoebae (this second version has been synthetized but not validated onboard).

6.3.2. Output. In the current version (validated on board) the states of all nodes are sequentially sent as an output to the host PC though the Master bus. This large output is useful for debug, but it requires a significant amount of resources, and it takes time. In the final version (not yet validated onboard), we take advantage of the quantitative criterion BBR (bounding box ratio) that is used in the experimental study of [6, 7] for the evaluation of the aggregation: minimal relative size of an array of nodes that contains all amoebae. Therefore, we implement an OR gate for each row and for each column of nodes, and we compute on-chip the resulting BBR, that is sent to the host PC in real time (i.e., during each clock cycle, the BBR is computed while all node states are updated).

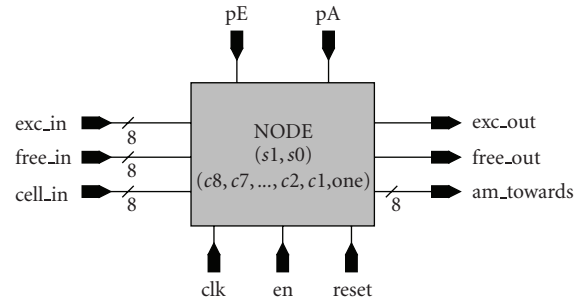


Figure 9: Node module.

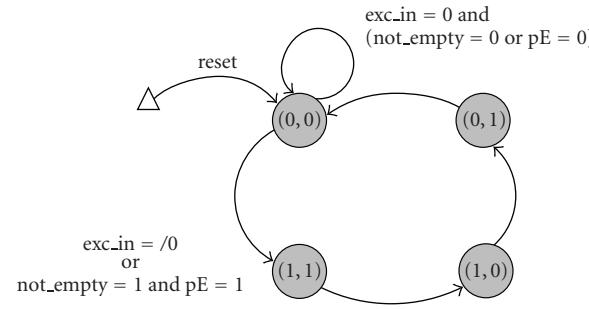


Figure 10: Cell state machine.

Table 3: Synthesis results for a 40×30 grid.

40 × 30 grid hardware resource utilization Xilinx FPGA XC4VLX160ff1513-12	
Number of Slice Flip Flops	31,642/135,168 (23.4%)
Number of 4 input LUTs	113,458/135,168 (83.94%)
Number of occupied Slices	61,727/67,584 (91.33%)
Frequency	130 MHz

6.4. Implementation Results

6.4.1. Resource Consumption. The prototyping platform is the same as in Section 4.4. Each node module requires 57 slices (21 for the different RNGs). Table 3 gives the synthesis results for an environment of 40 × 30 cells, taking advantage of resource optimization. Among the 61,727 used slices, only 794 ones implement the control and I/O handling (though we output all node states in the current version), so that the whole architecture is implemented on 91 of the FPGA resources.

6.4.2. Speedup. Software implementations on a microprocessor-based computer, Pentium 4.2 GHz, require 170 μs per evolution step for a 40 × 30 grid. As for the simulation of the environment alone, the main bottleneck for the software computation time lies in the random number generation (no cache management issue here). The maximum clock frequency of the proposed hardware is 130 MHz. Thus, the implementation on the Virtex-4 provides a speed factor up to 22000×.

Again, it must be pointed out that the used software is not optimized and has been written in Java, and that it does not perform exactly the same computations as the hardware architecture (random number generation, handling of priorities among neighbouring cells). Therefore, we consider that these results only indicate a $\Omega(10^4)$ order of magnitude for the speedup.

6.4.3. Analysis. Depending on the parameter values, the size of the environment and the obstacles, aggregation occurs in the experiments in [6, 7] after up to 20,000 iterations. Therefore the great speedup we obtain becomes particularly interesting if we are able to implement much larger grids.

Such improvements strongly depend on the analysis of the limits of the implementation depicted in this work (which was the main goal of the hardware design of the whole model with amoebae, as explained before). This analysis highlights three major sources of area consumption: coding and handling of populations of amoebae (28%), priority handling (23%), and above all random number generators (37%). Moreover, the implementation of the environment alone shows the great improvements that may be obtained thanks to a block-synchronous approach. But the described implementation would require that we store 11 bits per node (population + state) in the B-RAMs, and most of all, the exchanges of amoebae between nodes at the border could not be performed with sequentially handled blocks (since this handling results from a bidirectional information exchange through the `cell_in` and `am_towards` signals). This is why the current description of the model does not easily fit a block-synchronous version with amoebae.

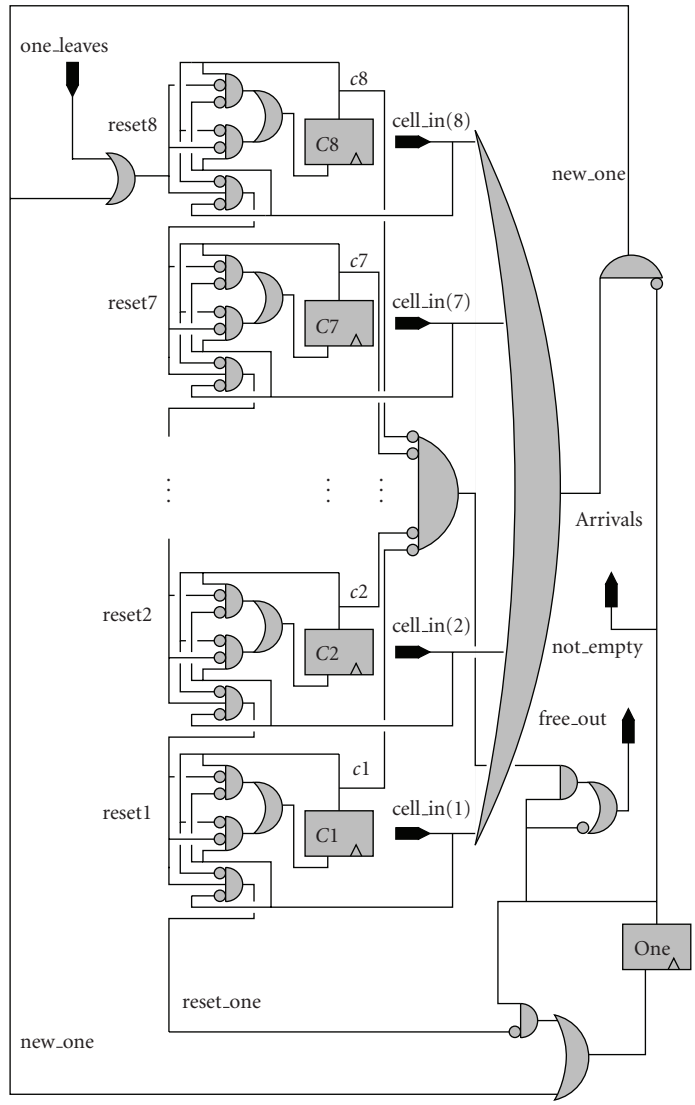


Figure 11: Module to code the population of amoebae.

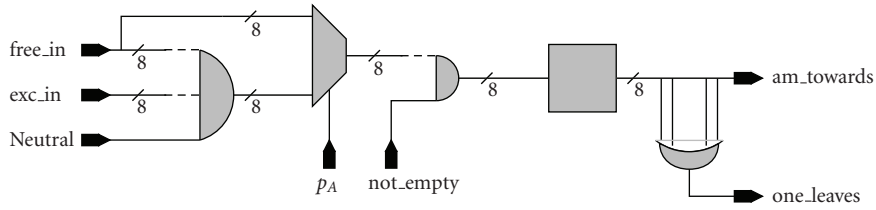


Figure 12: Implementation of the particle layer.

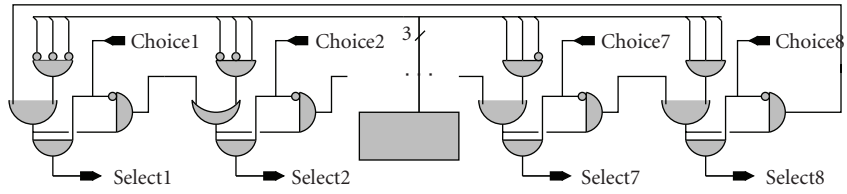


Figure 13: Selection module.

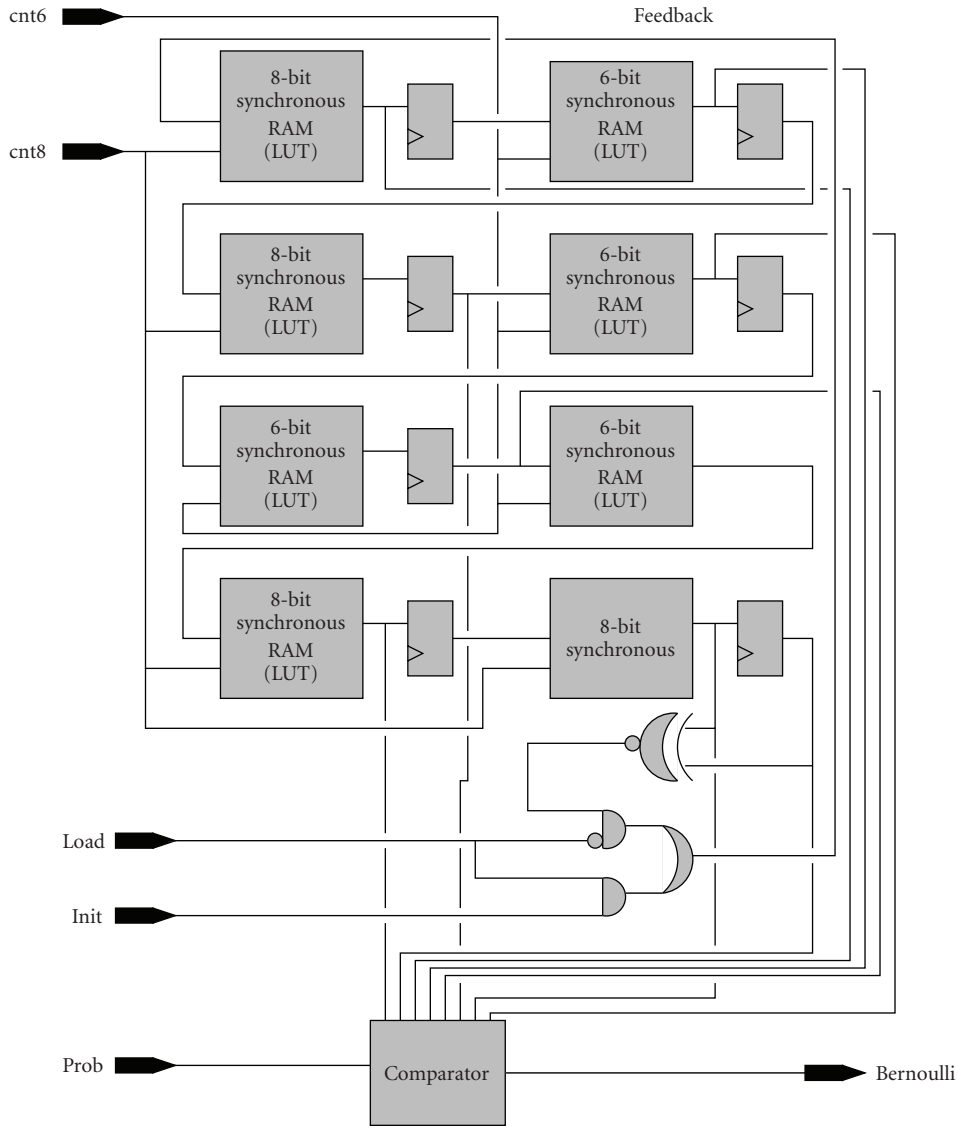


Figure 14: 63-bit random number generator to generate Bernoulli laws.

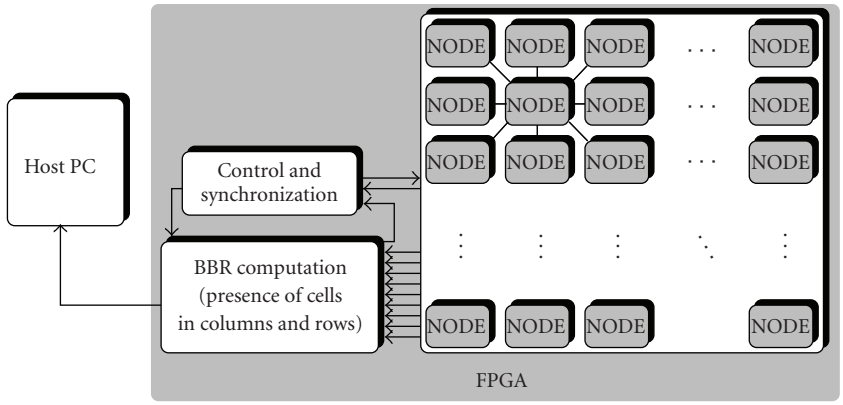


Figure 15: General architecture.

All these issues have led us to explore the definition of a new model for this decentralized gathering process. This new approach is fully based on cellular automata, including the RNGs. Though many theoretical and hardware aspects still need to be studied, it appears to be able to reduce the implementation area drastically: populations are directly handled through the cell state, resulting in a more likely block-synchronous implementation (though a fully parallel implementation corresponds more to the idea of decentralized gathering we explore), and random number resources are spatially mutualized. This is the main current research subject within the Amybia project.

7. Towards Decentralized Gathering of Computational Resources

The context of this work is the definition of innovative schemes of decentralized and massively distributed computing. Recent trends of integrated circuit design investigate various types of alternative computing devices based on multiple generic computing units, possibly distributed in an unknown and irregular way [26]. As stated in the introduction, our work aims at answering one of the problems raised by such new computing paradigms: how to gather enough computing resources to solve a given task. Though this paper describes an upstream work that does not yet pretend to define precisely how the gathering process will be applied to a real system, we may exhibit two possible contexts of use for such decentralized gathering.

7.1. Robot Swarms. Considering a swarm of simple robots that evolve in an environment with very restricted communication possibilities (due to obstacles for example), one may consider a task that alternates exploration and cooperation steps. Exploration is performed by robots that behave as autonomous agents, while cooperation is required when a "target" has been found. Robots that find targets try to attract other robots through decentralized gathering, until a sufficient number of gathered agents are able to perform the task associated to the target. Then robots start again their individual exploration.

As a first experimental setup, we have already implemented our decentralized gathering algorithm with Alice micro-robots (see a demo on http://www.loria.fr/~fates/Amybia/project.html). This application shows the great robustness of our algorithm, since these old robots have only two sensors to detect the light-simulated waves of the environment, and their motions are heterogeneous and almost unpredictable, due to the faulty control of their wheels. In such a context, the study of the properties of our decentralized gathering algorithm is essential, and it may take advantage of rapid simulations on FPGA; whereas an embedded implementation of the whole algorithm has no meaning, since each robot is an agent.

7.2. Task Assignment. Decentralized gathering may also be useful to handle task assignment in a massively distributed and heterogeneous computing device. In such a context, "moving" agents might correspond to transmitting the task assignments between units when using computational resources with fixed locations. In such devices, communication costs depend on the distance between the units, so that the communicating threads should be assigned to neighboring resources if possible. In a multi-task context, when a thread gives birth to other threads, they may be assigned to available computational resources that are not located in the neighborhood. When some resources become idle after having completed some thread, a reassignment process could be useful to gather the resources that handle the threads associated to the same task. A permanent decentralized gathering process might be useful for that if the resources are irregularly distributed and possibly faulty, provided that its cost is negligible with respect to the threads. Other constraints must be studied, such as the cost of context transfer between computational units, or the extension of decentralized gathering to multiple sets of agents to handle multiple tasks. Our preliminary implementation work does not conclude yet about the feasibility of a decentralized gathering process with a negligible cost.

8. Conclusion and Future Work

In this paper, a bioinspired model to solve the decentralized gathering problem is shortly described. It is based on the aggregation properties of the cellular slime mold Dictyostelium discoideum that may live as a monocellular organism, and that is able to behave as a multicellular organism when needed. We model the environment and the individual amoebae by means of cellular automata and reactive agents (simple computational abilities and no memory).

We have designed a hardware parallel implementation of the environment alone, that helps us perform rapid large-scale simulations to study the properties of our model, such as its robustness to noise and obstacles. The implementation results are highly satisfactory in terms of computation speed and environment size. This implementation is currently used so as to perform rapid simulations of phase transitions within a close-to-the-stable-state experimental framework.

Focusing on the whole model (environment and amoebae), we have designed a fully parallel hardware implementation so as to study its ability to provide a massively distributed computational model for decentralized gathering. Despite a great speedup factor, our implementation work points out two main limitations. In terms of embeddability, the area cost of the stochastic aspects of the model is important. Therefore, our theoretical study should evaluate the robustness of our model to low-quality random streams that may also be spatially correlated. In terms of usefulness for large-scale efficient simulations, the grid size we are able to handle does not correspond to interesting experimental environments, and the corresponding software computation time does not justify the use of fast FPGA-based simulations. To significantly increase the grid sizes handled by the FPGA, we currently explore solutions that are based on a block-synchronous approach and a new description of the model

that is fully based on cellular automata. This CA-based approach does not only intend to insert the behaviour of the agents within the state of each cell, but it also applies to the generation of random numbers. We currently consider the definition and design of spatially mutualized CA-based RNGs, that ensure both low-area implementations and a satisfactory spatial independence.

Acknowledgments

The authors wish to thank the other members of the *Amybia* INRIA collaborative research project (http://www.loria.fr/~fates/Amybia/project.html), Nazim Fatès and Hugues Berry, for their useful help and comments.

References

[1] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer, "Gathering of asynchronous robots with limited visibility," *Theoretical Computer Science*, vol. 337, no. 1–3, pp. 147–168, 2005.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, Oxford, UK, 1999.

[3] H. Abelson, D. Allen, D. Coore, et al., "Amorphous computing," *Communications of the ACM*, vol. 43, no. 5, pp. 74–82, 2000.

[4] J. Lawson and D. Wolpert, "Adaptive programming of unconventional nano-architectures," *Journal of Computational and Theoretical Nanoscience*, vol. 3, pp. 272–279, 2006.

[5] T. Bretschneider, B. Vasiev, and C. J. Weijer, "A model for cell movement during *Dictyostelium* mound formation," *Journal of Theoretical Biology*, vol. 189, no. 1, pp. 41–51, 1997.

[6] N. Fatès, "Gathering agents on a lattice by coupling reaction-diffusion and chemotaxis," Tech. Rep., INRIA Nancy Grand-Est, 2008, http://hal.inria.fr/inria-00132266/.

[7] N. Fatès, "Solving the decentralised gathering problem with a reaction-diffusion-chemotaxis scheme - Social amoebae as a source of inspiration," to appear in *Swarm Intelligence*, 2010 http://hal.inria.fr/inria-00132266/.

[8] K. Sugihara and I. Suzuki, "Distributed motion coordination of multiple mobile robots," in *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, vol. 1, pp. 138–143, Philadelphia, Pa, USA, September 1990.

[9] H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita, "Distributed memoryless point convergence algorithm for mobilerobots with limited visibility," *IEEE Transactions on Robotics and Automation*, vol. 15, no. 5, pp. 818–828, 1999.

[10] G. Prencipe, "Impossibility of gathering by a set of autonomousmobile robots," *Theoretical Computer Science*, vol. 384, no. 2-3, pp. 222–231, 2007.

[11] S. Nagano, "Modeling the model organism *Dictyostelium discoideum*," *Development Growth and Differentiation*, vol. 42, no. 6, pp. 541–550, 2000.

[12] A. Marée, *From pattern formation to morphogenesis: multicellular coordination in Dictyostelium discoideum*, Ph.D. thesis, Utrecht University, 2000.

[13] B. Vasiev, F. Siegert, and C. J. Weller II, "A hydrodynamic model for *Dictyostelium discoideum* mound formation," *Journal of Theoretical Biology*, vol. 184, no. 4, pp. 441–450, 1997.

[14] S. Mackay, "Computer simulaion of aggregation in *Dictyostelium discoideum*," *Journal of Cell Science*, vol. 33, pp. 1–16, 1978.

[15] J.-L. Martiel and A. Goldbeter, "A model based on receptor desensitization for cyclic AMP signaling in *Dictyostelium* cells," *Biophysical Journal*, vol. 52, no. 5, pp. 807–828, 1987.

[16] J. R. Weimar, "Cellular automata for reaction-diffusion systems," *Parallel Computing*, vol. 23, no. 11, pp. 1699–1715, 1997.

[17] A. Adamatzky, *Computing in Nonlinear Media and Automata Collectives*, Institute of Physics, Bristol, UK, 2001.

[18] N. Fatès, "Asynchronism induces second order phase transitions in elementary cellular automata," *Journal of Cellular Automata*, vol. 4, no. 1, pp. 21–38, 2009.

[19] H. J. Blok and B. Bergersen, "Synchronous versus asynchronous updating in the "game of Life"," *Physical Review E*, vol. 59, no. 4, pp. 3876–3879, 1999.

[20] H. Hinrichsen, "Nonequilibrium critical phenomena and phase transitions into absorbing states," *Advances in Physics*, vol. 49, no. 7, pp. 815–958, 2000.

[21] B. Girau, A. Boumaza, B. Scherrer, and C. Torres-Huitzil, "Block-synchronous harmonic control for scalable trajectory planning," in *Robotics Automation and Control*, A. lazinica, Ed., pp. 85–110, I-Tech Publications, 2008.

[22] I. Kokolakis, I. Andreadis, and P. Tsalides, "Comparison between cellular automata and linear feedback shift registers bases pseudo-random number generator," *Microprocessors and Microsystems*, vol. 20, no. 10, pp. 643–658, 1997.

[23] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[24] M. George and P. Alfke, "Linear feedback shift registers in virtex devices," *Xilinx Application Note XAPP210 (v1.3)*, 2007.

[25] G. DeYoung, P. B. Monk, and H. G. Othmer, "Pacemakers in aggregation fields of *Dictyostelium discoideum*: does a single cell suffice?" *Journal of Mathematical Biology*, vol. 26, no. 5, pp. 487–517, 1988.

[26] A. Dehon, "Very large scale spatial computing," in *Proceedings of the 3rd International Conference on Unconventional Models of Computation (UMC '02)*, pp. 27–37, Kobe, Japan, October 2002.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 301512, 10 pages
doi:10.1155/2009/301512

Research Article

Software Toolchain for Large-Scale RE-NFA Construction on FPGA

Yi-Hua E. Yang and Viktor K. Prasanna

Department of Electrical Engineering, University of Southern California,Los Angeles, CA 90089-0911, USA

Correspondence should be addressed to Yi-Hua E. Yang, yeyang@usc.edu

Received 15 March 2009; Accepted 19 June 2009

Recommended by Lionel Torres

We present a software toolchain for constructing large-scale *regular expression matching* (REM) on FPGA. The software automates the conversion of regular expressions into compact and high-performance nondeterministic finite automata (RE-NFA). Each RE-NFA is described as an RTL regular expression matching engine (REME) in VHDL for FPGA implementation. Assuming a fixed number of fan-out transitions per state, an n-state m-bytes-per-cycle RE-NFA can be constructed in $O(n \times m)$ time and $O(n \times m)$ memory by our software. A large number of RE-NFAs are placed onto a two-dimensional *staged pipeline*, allowing scalability to thousands of RE-NFAs with linear area increase and little clock rate penalty due to scaling. On a PC with a 2 GHz Athlon64 processor and 2 GB memory, our prototype software constructs hundreds of RE-NFAs used by Snort in less than 10 seconds. We also designed a benchmark generator which can produce RE-NFAs with configurable pattern complexity parameters, including state count, state fan-in, loop-back and feed-forward distances. Several regular expressions with various complexities are used to test the performance of our RE-NFA construction software.

1. Introduction

Regular expression matching (REM) has many applications ranging from text processing to packet filtering. In the narrow sense, each regular expression defines a regular language over the alphabet of input characters. A regular language applies three basic operators on the alphabet: *concatenation* ($\cdot$), *union* ($|$), and *Kleene closure* ($*$), which allow the construction of complex expressions. There are other common operators that also conform to the regular language construct, such as *character classes* ($[\cdots]$), *optionality* (?), and *constrained repetitions* ({a,}, {,b}, {a,b}). All of these operators can be realized by proper arrangements of the three basic ones.

Improving large-scale REM performance has been a research focus in the recent years [1–11]. Since regular languages can be necessarily and sufficiently accepted by finite state automata, a regular expression matching engine (REME) supporting *concatenation*, *union*, *closure*, *repetition*, and *optionality* can always be implemented as either a nondeterministic finite automaton (RE-NFA) or a deterministic finite automaton (RE-DFA). Figure 1 compares side by side the architectures of the two types of automata.

In an RE-NFA approach [2, 4, 7–10], individual regular expressions and their character matching states are processed in parallel with one another. As a result, more than one state in an RE-NFA can be *active* at any time. Optimizations such as input/output pipelining [4], common-prefix extraction [2, 4], multicharacter input [9, 10], and centralized character decoding [2, 12] can be applied to improve throughput and reduce resource requirements of the overall design.

In an RE-DFA approach, several regular expressions are grouped (`union`'d) into a DFA by expanding different combinations of active states into additional *combined states*. In principle, only one combined state in an RE-DFA is active at any time. Various techniques [5, 6, 13, 14] are then applied to improve memory access efficiency and to reduce the total number of states, which usually suffers from quadratic to exponential explosion [11].

Due to the matching power of regular expressions and the complexity of the strings being matched, the REM process can be the slowest bottleneck of a system. To match a regular

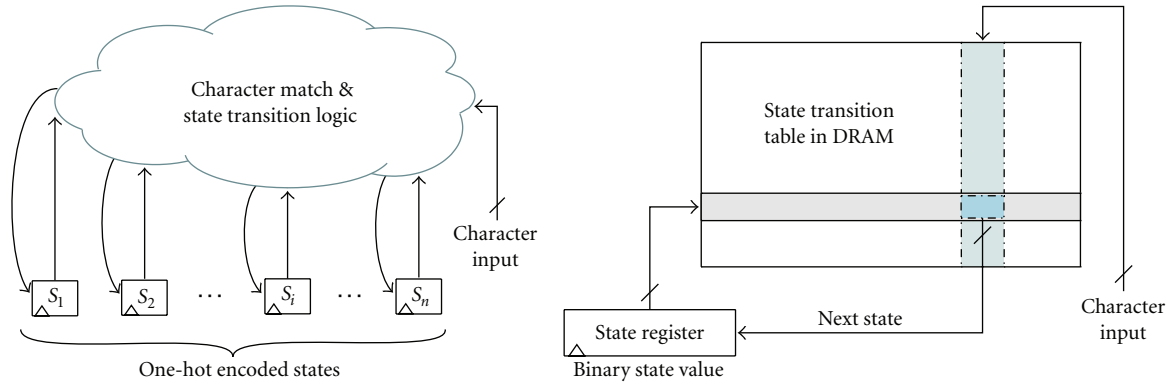


Figure 1: Basic architectures of RE-NFA (a) and RE-DFA (b). Our work focuses on regular expression matching using the RE-NFA architecture.

expression of length n over an alphabet of size Σ can take up to $O(n^2)$ time to process each character (for RE-NFA) or $O(\Sigma^n)$ memory space to store the state transition table (for RE-DFA) [11]. Furthermore, to match K concurrent regular expressions, the overall throughput could be K times slower (for RE-NFA) or take $O(\Sigma^K)$ more memory space (for RE-DFA) in the worst case.

Modern FPGAs offer large amount of reconfigurable logic (LUTs) and on-chip memory (BRAM). We developed a compact and high-performance RE-NFA architecture for REM which utilizes both on-chip logic and memory resources on modern FPGAs [10]. In this study, we focus on the automatic parsing, translation, and construction of regular expressions matching engine (REME) using our RE-NFA architecture for fully automated FPGA implementation. More specifically, we develop an REME construction software with the following components

(1) Automatic conversion from regular expression parse tree [15] to a uniform and modular RE-NFA structure.

(2) Automatic generation of RTL code in VHDL for each RE-NFA. The resulting circuit is spatially stacked a configurable number of times for multicharacter matching.

(3) Allocation of centralized character classification in BRAM for up to 256 REMEs using a simple heuristics.

(4) Automatic construction of up to 16 pipelines in a two-dimensional structure.

(5) A benchmark generator of regular expressions with configurable pattern complexity parameters (*state count*, *state fan-in*, *loop-back,* and *feed-forward* distances).

The rest of this paper is organized as follows. The background and prior work of RE-NFA on FPGA are discussed in Section 2. An overview of our software toolchain is given in Section 3. Section 4 describes REME construction, while Section 5 covers architectural optimization. Section 6 introduces an REME benchmark generator and uses it to evaluate the performance of the REME construction and optimization software. Section 7 concludes the paper.

2. Background and Related Work

Hardware implementation of regular expression matching (REM) was first studied by Floyd and Ullman [15], where an n-state RE-NFA is translated into integrated circuits using no more than $O(n)$ circuit area. Sidhu and Prasanna [8] later proposed an algorithm to implement REM on FPGA in a similar RE-NFA architecture, which has been used by most other RE-NFA implementations on FPGAs [2, 4, 7, 9]. Yang et al. [10] adopted a different approach to translate arbitrary regular expressions to corresponding RE-NFAs with a more modular and uniform circuit structure.

Automatic REME construction on FPGAs was first proposed in [4] using JHDL for both regular expression parsing and circuit generation. In particular, the (J)HDL construction approach used in [4] is in contrast to the *self-configuration* approach done by [8]. Reference [4] also considered large-scale REME construction, where the character input is broadcasted globally to all states in a tree-structured pipeline. Automatic REME construction in VHDL was proposed in [2, 7]. In [2], the regular expression was first tokenized and parsed into a hierarchy of basic NFA blocks, then constructed in VHDL using a bottom-up scheme. In [7], a set of scripts was used to compile regular expressions into op-codes, to convert op-codes into NFA, and to construct the NFA circuits in VHDL.

A multi-character decoder was proposed in [16] to improve pattern matching throughput. While the technique was claimed to be applicable to REM, only the construction of a fixed-string matching circuit was explained. The paper, however, did not describe an automatic mechanism to translate any general pattern into a multi-character matching circuit. An algorithm that extends any single-character matching REME *temporally* into a multi-character matching REME was proposed in [9]. In contrast, the

```
Notations:
n [value]    Content value of node n.
n [left|right|child] Left, right, or only child of node n.
s [next]     Set of next-state transitions of state s.
s[char]      Set of matching characters of state s.
Macros:
s ← CREATE_STATE (T):
              Create a new state s in the state transition table
              T.
p ← CREATE_PSEAUDO():
              Create a special pseudo-state p for later use.
ADD_PSEUDO_NEXT (p, S):
              For every state s ∈ S, add the state set p [next]
              to s [next]. Pseudo-state p is deleted afterward.
PROCEDURE S_out ⇐RE2NFA (n_root, S_pre, T_NFA)
  n_root   Root node of the parse (sub-)tree.
  S_pre    Set of immediate previous states.
  S_out    Set of states transitioning directly outside of n_root.
  T_NFA    The resulting state transition table.
BEGIN
    n_cur ← n_root;
    while n_cur ≠ null
      if n_cur [value] = OP_CONCAT
        S_pre ←RE2NFA (n_cur [left], S_pre, T_NFA);
        n_cur ← n_cur [right];
      else if n_cur [value] = OP_UNION
        S_L ← RE2NFA (n_cur [left], S_pre, T_NFA);
        S_R ← RE2NFA(n_cur [right], S_pre, T_NFA);
      return S_L ∪ S_R;
      else if n_cur [value] = OP_CLOSURE
        p ← CREATE_PSEUO();
        S_tmp ← S_pre ∪ p;
        S_C ← RE2NFA (n_cur [child], S_tmp, T_NFA);
        ADD_PSEUDO_NEXT(p, S_C);
        return  S_C ∪ S_pre;
      else // n_cur = leaf node
        s_new ← CREATE_STATE(T_NFA);
        s_new [char]← n_cur [value];
        foreach s in S_pre
          // add ε-transitions
          s [next]← s [next]∪s_new;
        end foreach
        return s_new;
      end if
    end while
    // error: n_cur [right] cannot be null
END
```

Algorithm 1: Modified McNaughton-Yamada construction (MMY) converting a regular expression parse-tree to an RE-NFA with a modular and uniform structure.

uniform structure of the RE-NFA in [10] allows its circuit to be stacked *spatially* and automatically to process multiple characters per clock cycle.

3. Overview of the Software Toolchain

The main purpose of our software toolchain is to automate the construction and optimization of large-scale RE-NFA circuits on FPGA. The toolchain allows us to generate the whole RTL circuit matching thousands of regular expressions in orders of seconds using a single command. Such a toolchain can help us not only to avoid the tedious and error-prone circuit construction, but also to generate a large-scale regular expression matching engine (REME) for implementation in a small amount of time.

Figure 2 gives an overview of the toolchain. The toolchain consists of two main parts: *REME Construction* and *Architectural Optimization*, briefly described as follows:

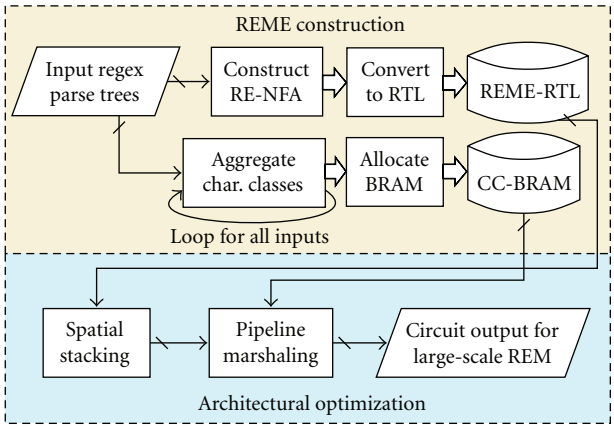


Figure 2: Overview of our toolchain for large-scale REME construction.

(1) *REME Construction*: converts each regular expression into an RE-NFA circuit and collects unique character classes in BRAM across all regular expressions.

(2) *Architectural Optimization*: applies spatial stacking to the individual RE-NFA circuits; marshals RE-NFAs into a 2D staged pipeline to form the final circuit.

In practice, the two paths of REME Construction in Figure 2 are written as a single module interleaving the two tasks for each input regular expression. Conceptually, however, they are independent of each other and can be executed in parallel. In contrast, the two tasks in Architectural Optimization, *spatial stacking,* and *pipeline marshaling* must be performed in serial. The details of the REME Construction part are presented in Section 4, while those of the Architectural Optimization part are in Section 5.

In addition to the basic operators of concatenation, union (|) and Kleene closure ($*$) used to define a regular language, our software also handles most frequently used operators by the Snort IDS [7] such as the repetition (+), optionality (?), constrained repetition ({a,b}), and any character class ([$\cdots$]). Table 1 lists the operators supported by our software. The syntax and semantics of these operators are compatible with the Perl-Compatible Regular Expression [17]. For example, the expression "[0 − 9]{1,3}(\ · [0 − 9]{1,3}){3}[^0 − 9]?" specifies any IP address followed by an optional nonnumerical characters.

4. Automatic REME Construction

The REME Construction is performed in three steps: (1) parse the regular expressions into tree structures, (2) use the *modified* McNaughton-Yamada (MMY) construction (Figure 4, Algorithm 1) to construct the RE-NFAs, (3) map the RE-NFAs into structural VHDL suitable for FPGA implementation.

4.1. From Regular Expression to Parse Tree. The first step is to represent each regular expression as a corresponding parse tree using a standard compiler technique. This step is the same as that described in [15]. Figure 3 shows a parse-tree representation of a regular expression "\ x2f(fn|s)\x3F[^\r\n]*si." This is simplified for the value of illustration from an actual Snort [18] pattern. In particular, a union of any number of single characters is parsed as a single character class (e.g., the [^\r\n] in Figure 3), which can be matched very efficiently in our REM architecture [10].

The resulting parse tree always consists of three types of internal nodes, op_concat, op_union, and op_closure, and a number of leaf nodes equal to the number of individual (and possibly nonunique) character classes in the regular expression.

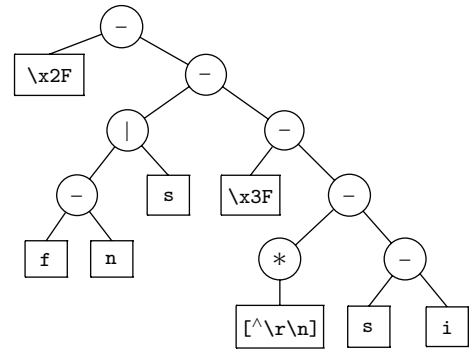


Figure 3: Parse-tree representation of "\ x2F(fn|s)\x3F[^\r\n] *si."

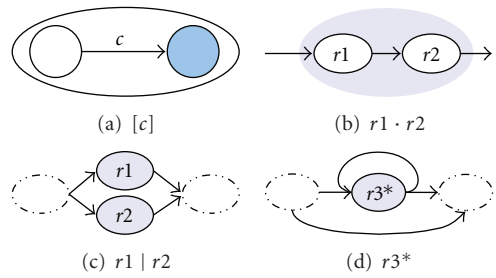


Figure 4: Graphical representation of the modified McNaughton-Yamada (MMY) construction. Note that unlike the original construction, no ϵ-transition-only node is introduced in rules (c) and (d), where the dashed ellipses are *not* part of the current construction.

Table 1: REM operators support by our software.

Op.	Name	Example	Description
-	Concatenation	q_1q_2	q_2 right after q_1
|	Union	$q_1\|q_2$	Either q_1 or q_2
$*$	Kleene closure	$q*$	q zero or more times
+	Repetition	$q+$	q one or more times
?	Optionality	$q?$	q zero or one times
m,n	Constrained rep.	$q\{m,n\}$	q in m to n times
[$\cdots$]	Character class	[a–c]	Either a, b or c
[^$\cdots$]	Inv. char. class	[^\r\n]	Neither \r nor \n
^	Match beginning	^q	q at beginning of input
$	Match ending	$q$$	q at ending of input

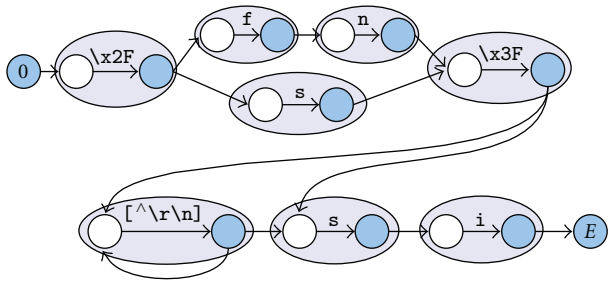


FIGURE 5: A modular NFA for "\ x2F(fn|s)\x3F[^\r\n]*si" constructed using the MMY rules specified in Figure 4.

4.2. From Regular Expression Parse Tree to NFA. Unlike previous work in [15] and later in [8] which use the McNaughton-Yamada (MNY) construction to convert regular expressions into RE-NFAs, we proposed the *modified McNaughton-Yamada (MMY) construction* in [10] to perform the conversion. Figure 4 gives a graphical description of the modified construction rules.

A formal definition of the construction mechanism is given in Algorithm 1. The algorithm takes the regular expression parse tree generated from the previous subsection as input. It is in general a recursive algorithm, where the subtrees of each internal node is processed recursively before the operator of the current node is handled. The only exception is the right child of an `op_concat` node, where for performance reason the tail recursion is performed iteratively. This avoids excessive recursion for a long sequence of `op_concat` operators (which is predominantly the case in real-world patterns).

Two special entities are used in Algorithm 1 for the MMY construction. The first is the set of immediate previous states S_{pre}, which contains the source states of all fan-in transitions to the part of RE-NFA currently under construction. This entity corresponds to the dashed ellipses on the left of Figures 4(c) and 4(d). It allows a long sequence of ϵ-transitions in the original MNY construction to be collapsed into a single ϵ-transition in the MMY construction.

The second entity is the pseudostate p, which works as a placeholder for the source states of an `op_closure`'s feedback loop before the `op_closure` is converted to be part of the RE-NFA. This temporary placeholder is needed to break the circular dependence of an `op_closure` construction on the resulting fan-out states of the very `op_closure` construction.

The MMY construction algorithm produces an NFA extremely modular and easy to map to HDL codes. For example, using the modified construction algorithm, the regular expression "\ x2F(fn|s)\x3F[^\r\n]*si" is converted into a modular NFA with a uniform structure (Figure 5). This conversion is arguably the most complex part of the construction process, taking roughly 350 lines of C code for the automation.

4.3. From RE-NFA to VHDL. To translate the RE-NFA (like Figure 5) into VHDL, each pair of nodes inside a lightly shaded ellipse is mapped to an `entity statebit` with one parameter: the number of input ports, determined by the number of "previous states" that immediately transition to the current state. Inside the `entity statebit`, all inputs aggregate to a single `OR` gate, followed by a character matching via logic `AND` and a state value register. The single-bit output value of the register is connected to the inputs of the immediate "next states."

The REM circuit for Figure 5 is shown in Figure 6. On FPGA devices with 4-input LUTs, a k-input `OR` followed by a 2-input `AND` can be efficiently implemented on a single LUT if $k \leq 3$, or on a single slice of 2 LUTs if $4 \leq k \leq 7$. The mapping takes only about 300 lines of C code to convert any RE-NFA to its RTL structural VHDL description.

4.4. BRAM-Based Character Classification. Our REM architecture in [10] used a 256-bit column of BRAM to match any character class of 8-bit characters. Each bit of the column represents the inclusion of an 8-bit character in the character set. The value of every input characters is used as a row index to BRAM to retrieve the matching result (*true*|*false*) of that character against all character classes (one for each column). Each single-bit result is routed from BRAM to its corresponding correct `entity statebit` as the input to the `AND` gate. As a result, character classification of an n-state RE-NFA can be implemented on a block memory (BRAM) of no more than $256 \times n$ bits.

Furthermore, if two states (either within the same regular expression or across different regular expressions) match the same character class, then they can share the same BRAM column output. We use a two-phase procedure to aggregate the matching outputs of identical character classes.

(i) In phase 1, the software collects the set of unique character classes from a regular expression. Each unique character class is associated with a floating-point *sorting key*:

  (a) if the character class appears only once in the regular expression, then the sorting key is its (only) position index within the regular expression;

  (b) if the charactter class appears multiple times in the regular expression, then the sorting key is the average of all its position indexes within the regular expression;

(ii) In phase 2, the unique character classes are sorted according to their sorting keys and instantiated as BRAM columns. Each BRAM column is also associated with the identifier of the instantiated character class. The output of each BRAM column is then connected to the character matching inputs with the same identifier.

The two-phase procedure allows our software to use the minimum number of BRAM columns for character class matching. It also minimizes routing distance by exploiting the natural ordering (the sorting keys) of the character classes

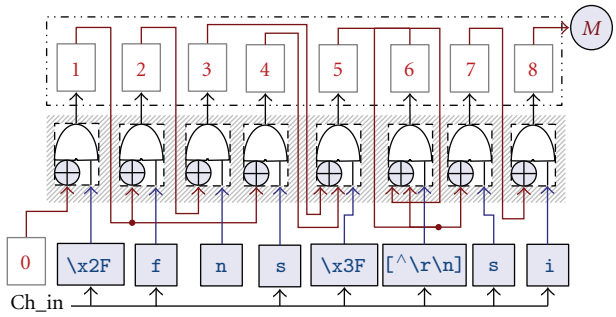


Figure 6: REM circuits constructed by mapping Figure 5 directly to HDL. The ⊕ symbols represent logic OR gates.

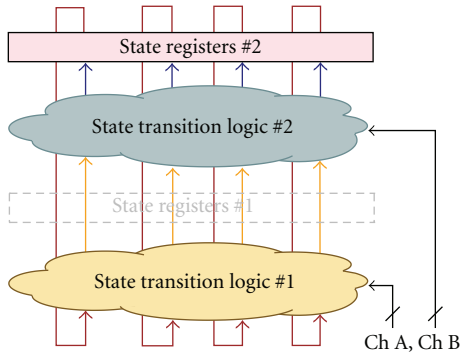


Figure 7: The construction of a 2-character matching circuit.

within the regular expressions. The aggregation of character classes and their distribution to the RE-NFA states take ~250 lines of C code.

5. Automated Architectural Optimizations

After constructing REMEs individually for all regular expressions, the software applies two architectural optimizations [10]. (1) The REMEs are *stacked* to form multi-character matching (MCM) circuits which trade off minimum resource usage for higher performance. (2) The MCM REMEs are grouped into clusters of 16 and marshaled onto a two-dimensional *staged pipeline* structure.

5.1. Circuit Stacking for Multicharacter Matching. In contrast to the NFA-level *temporal extension* used in [9], we adopted a circuit-level *spatial stacking* to construct multi-character matching (MCM) REMEs. Figure 6 shows the basic construction concept of a 2-character matching circuit from two copies of a single-character matching circuit. An algorithm for this spatial stacking approach and the proof of correctness were given in [10]. Benefits of the spatial stacking approach include the following.

Simplicity. The time complexity to construct an n-state, m-character matching REME using spatial stacking is $O(n \times m)$ [10]. In contrast, the time complexity of temporal extension is $O(n^3 \log m)$ [9].

Flexibility. The spatial stacking approach can generate an MCM REME of any natural number m, while the temporal extension approach only generates RE-NFAs with $m = 2^i$.

In practice, n is usually a few tens while m between 2 to 8, making the spatial stacking approach hundreds of times faster than the temporal extension approach. As discussed in Section 6.2, our software can construct thousands of MCM REMEs in ~10 seconds. Also, the optimal value of m with respect to *performance efficiency* (defined in [10]) is usually not a power of 2. For example, the REMEs from Snort rules achieve optimal performance efficiency at $m = 6$ [10].

The program code to construct any m-character matching REME using spatial stacking is simple. Let C be a single-character matching circuit. The program first makes m copies of C, $\{C^{(1)}, \ldots, C^{(m)}\}$, each receiving one of the m consecutive input characters. Then, instead of routing the state outputs back to the state inputs of the same circuit, it removes the state registers of $C^{(k)}$ and connects the (nonregistered) state outputs of $C^{(k)}$ to the state inputs of $C^{(k+1)}$ for $k = 1, \ldots, m-1$. Finally, it connects the (registered) state outputs of $C^{(m)}$ to the state inputs of $C^{(1)}$. The result is an m-character matching circuit for C.

In general, to construct an $(l + m)$-character matching circuit C_{l+m}, we perform the following transformations on every state $i \in \{1, 2, \ldots, n-1\}$ of C_l and C_m:

(1) remove state register i of C_l; forward the AND gate output to its state output,

(2) disconnect state output i of C_l from the state inputs of C_l, and reconnect it to the corresponding state inputs of C_m,

(3) disconnect state output i of C_m from the state inputs of C_l, and reconnect it to the corresponding state inputs of C_l,

(4) the combined circuit receives $(l + m)$ character matching signals per cycle. The first l signals are sent to the C_l part; the last m signals are sent to the C_p part.

5.2. REME Clustering for Staged Pipelining. With a straightforward implementation, the BRAM-based character classifier (Section 4.4) uses 256 bits per state. To implement thousands of REMEs with tens of thousands states, the character classifier would require tens of megabits of BRAM and become the resource bottleneck on FPGA. A second issue in implementing large number of REMEs on FPGA is signal routing. The character matching results from the centralized character classifier in BRAM must be distributed to all REMEs, while the pattern matching result from every REME must be collected and aggregated to the final output. The potentially long routing makes the circuit hard to scale to large number of REMEs.

A *2D staged pipeline* design was proposed in [10] to solve both problems. Figure 8 shows the basic structure of such a staged pipeline. Each stage may contain a cluster of up to 16 REMEs. The horizontal arrows between the pipelines are the signal paths of the input characters. The vertical arrows between pipeline stages are the character matching signals

and the pattern matching results. A priority encoder is used at every stage and pipeline to aggregate the pattern matching results.

Marshaling REMEs into this staged pipeline structure, however, is painstaking and error-prone when done manually. This is mainly due to the buffering and distribution of the character matching signals (the thick vertical arrows in Figure 8). Additionally, different REME grouping can result in different resource usage and routing complexity and give rise to performance variation among REME clusters. To solve these problems, our software use the following heuristic to marshal k REMEs with total N states into p pipelines.

(1) First calculate the average number of states per pipeline, $v = N/p$.

(2) Add any of the k REMEs into a new pipeline. Compute the *compatibility* between the resulting (single-REME) pipeline and each of the rest $k - 1$ REMEs. The *compatibility* between a pipeline and an REME is defined as the number identical character classes in both divided by the number of unique character classes in the REME.

(3) Add the most compatible REME to the pipeline. Recompute the compatibility of all remaining REMEs.

(4) Repeat step 3 until the total number of states in the pipeline is greater than $v - \sigma$, where $\sigma \ll v$ is a design constant.

(5) Go back to step 2 to work on a new pipeline until all REMEs are exhausted.

After marshaling the REMEs into different pipelines, the REMEs within each pipeline are marshaled into different stages in a similar manner. When adding an REME to a pipeline, a function is called to compare each of the character class in the REME to the character classes previously collected in BRAM. If an identical character class is found, then proper connections are made from the BRAM output to the inputs of the respective states.

The time complexity of this procedure is $O(k \times N \times w)$, where w is the number of *distinct* character classes among the N states in the k REMEs. The space complexity is $O(256 \times w)$. In real applications, w grows almost linearly with respect to N for small N, but quickly flats out and grows much slower than $O(\log N)$ when N is moderately large (a few hundred).

Matching outputs from all REMEs are prioritized. Currently, the software assigns higher priority to lower-indexed pipelines and stages, although the priority can be programmed in any other way with little additional complexity.

6. Experimental Results

6.1. Design of Benchmark Generator. We developed a regular expression *benchmark generator* to test how different types of regular expressions affect the performance of the REMEs

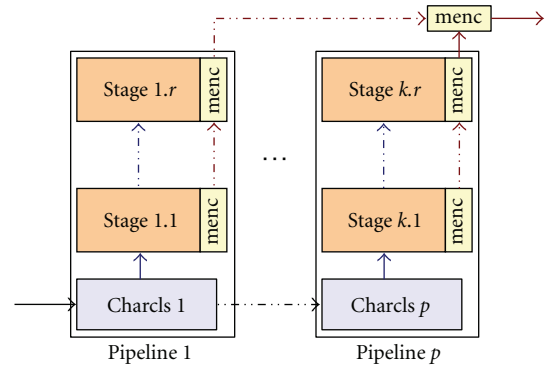


Figure 8: Structure of a 2D staged pipeline with total p pipelines and r stages per pipeline.

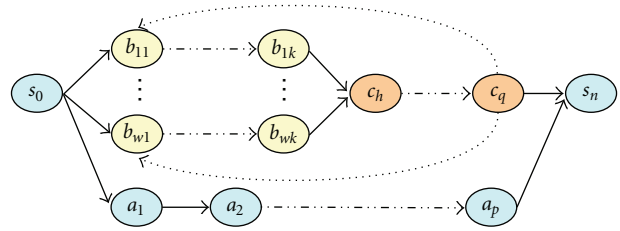


Figure 9: Structure of the regular expressions from the benchmark generator.

constructed by our software. The benchmark generator produced regular expressions of different *state count* (n), *state fan-in* (w), and variable lengths of *loop-back*(q) and *feed-forward* ($q - p$). A general structure of the generated regular expressions is described in Figure 9. (Due to our use of BRAM for character classification, every character class, no matter how simple or complicated it is, takes exactly 256 BRAM bits and is matched by one BRAM access. Since the complexity of character classes does not affect performance, our benchmark generator assigns arbitrary values to the character classes without loss of generality.)

State count represents the total number of states in an RE-NFA. It was used by most related work as the primary metric for REME complexity [2, 4, 7, 9]. We further defined *state fan-in* as the *maximum* number of transitions entering any state [10], since the state machine runs at the speed of the slowest state transition. Both `op_union` and `op_closure` can increase state fan-in, which is the secondary metric for REME complexity.

A state transition *loop-back*is always caused by an `op_closure`, while a state transition *feed-forward* can be caused by unbalanced alternative paths within an `op_union`. Both properties are high-order metrics describing the routing lengths of an REME. According to our experimental experience, however, the actual routing complexity of the REME circuit on FPGA is highly subject to the optimizations done by the place and route software and may not reflect these two metrics closely.

6.2. Performance Evaluation of the Software Toolchain. The time taken to translate a set of parsed regular expressions

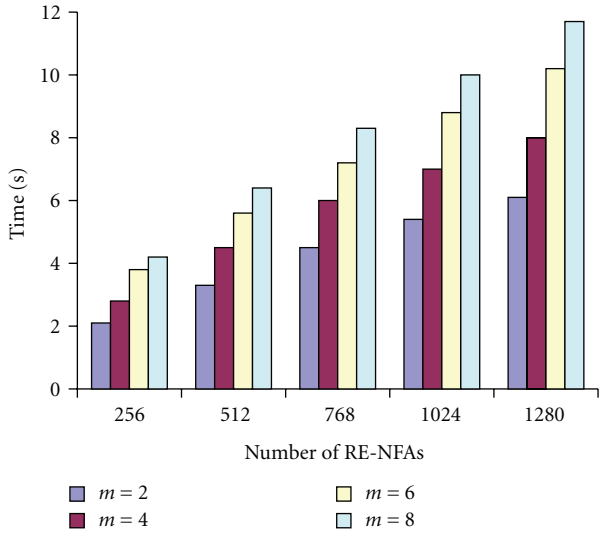


Figure 10: REME construction time for various number of regular expressions and multi-character matching parameters.

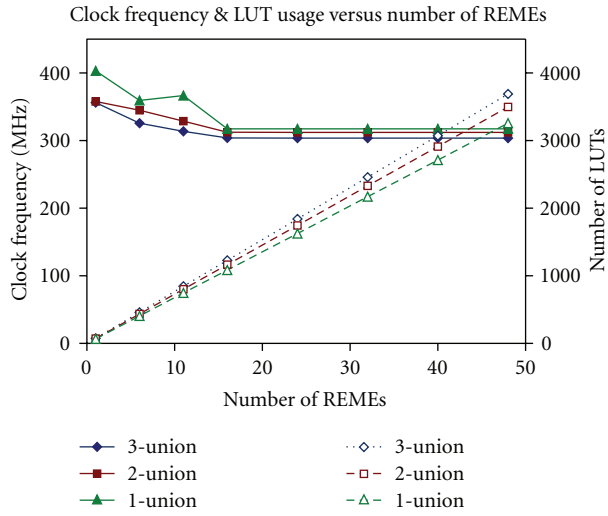


Figure 12: Clock frequency and LUT usage of group of 64-state synthetic REMEs versus number of REMEs implemented. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

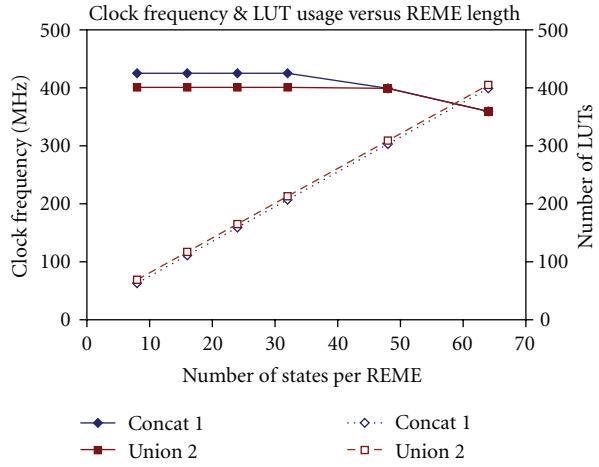


Figure 11: Clock frequency and LUT usage of group of 6 identical synthetic REMEs versus length of every REME. Solid lines (left scale) are clock frequencies; dashed lines (right scale) are number of LUTs.

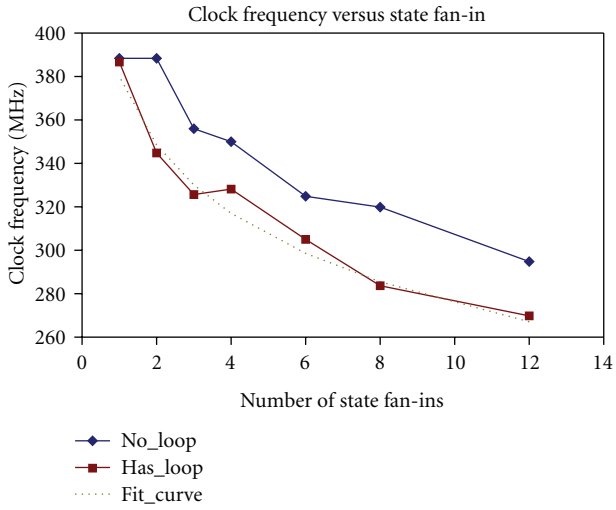


Figure 13: Clock frequency versus state fan-in of the synthetic REMEs.

to VHDL was roughly proportional to the product of the *number of states* (n) and the *size of multi-character input* (m), an observation agreeing with our analysis in Section 5.1. On a 2 GHz Athlon 64 PC, it took between 6 and 12 seconds to translate 1280 Snort REMEs (~28k states) to VHDL, as m increased from 2 to 8. In all cases, about 30% of the time was used for file I/O. Figure 10 illustrates the construction time of various cases in more detail. (Due to the relatively large I/O overhead and the short overall runtime, there is high variance (~15%) among different runs of the same construction. The construction time is also greatly affected by the complexity of regular expressions, especially the *state count* and the *state fan-in* discussed in Section 6.1.)

These results show that the software proposed in this paper is suitable for large-scale REME construction. Since it takes only a few seconds to translate a thousand regular expressions into structural VHDL, the software can be used to reconstruct a large-scale REME quickly in response to dictionary changes. Due to the large number of logic resource used, however, the synthesis and place and route times are in the order of several tens minutes.

6.3. Performance Evaluation of the Constructed REMEs. We first used the benchmark generator described in Section 6.1 to produce synthetic regular expressions of different

numbers and complexities, then use our REME construction software to convert the synthetic regular expressions into 2-character matching REME circuits in VHDL. We synthesized the VHDL into Xilinx NGC targeting the Virtex 4 LX device family and extracted the estimated clock frequency from the timing analysis.

Figure 11 shows clock frequency and LUT usage versus length of REMEs. Series `concat1` was produced by one long sequence of concatenations. Series `union2` was produced by a union of two equal-length concatenations. In each test case, 6 identical REMEs were placed into a single stage.

Series `union2` ran at lower clock frequency than series `concat1` due to the use of the `op_union` operator, which caused series `union2` to have twice the (maximum) state fan-in as `concat1`. The clock rates of both series started to decline gradually with respect to REME length around 32 to 40 states per REME. This decline was due to the longer paths to access the centralized character classification signals from BRAM. This is evidenced by the fact that both `concat1` and `union2` ran at about the same clock rates beyond the length of 40 states, showing a bottleneck elsewhere from the state transitions within the logic slices of FPGA.

In Figure 12, we analyzed the effect of the number of REMEs on achievable clock frequency and total LUT usage. In each test case, 64 states were generated for each REME; 30 states were wrapped inside an `op_closure` ($q = 30$), which was then `op_union`-ed with a sequence of 30 other states ($p = 30$) and concatenated with the last 4 states in sequence. In the w-`union` series, $w = 1, 2, \text{or } 3$, the 30 states inside the `op_closure` were further wrapped by an `op_union` of w operands, each $30/w$ states in length. The purpose was to see how clock rate scaled with respect to number of REMEs for different REME structures and complexities.

As shown in Figure 12, clock frequency declined between 15% to 25% when number of REMEs varied from 1 to 16. All these 16 REMEs are put inside a single stage by our software. Since the added regular expressions were all identical, this decline was again due to longer BRAM access, caused by both longer routes and larger fan-out.

Above 16 REMEs, however, the staged pipeline came into effect, keeping the clock rates at slightly above 300 MHz. This evidently shows that the staged pipeline proposed in [10] was effective in scaling up number of REMEs in a single circuit. LUT usage maintained linear increase with respect to the number of REMEs.

As expected, a higher w value results in a slightly lower achievable clock frequency due to the higher state fan-in of the REMEs.

Figure 13 examines clock frequency versus state fan-in more thoroughly. In each test case, REMEs of 52 states were constructed, with 24 states put inside an `op_union` of w operands, w varying from 1 (single 24-state sequence) to 12 (union of 2-state sequences). For the `has_loop` series, there was also a loop-back transition from the outputs of the 24-state union back to the inputs of the union itself. There was no such loop-back for the `no_loop` series.

The clock frequency was found to decline sublinearly with respect to the state fan-in, at a rate consistent with the findings in Section 6.2. The decline however was not completely smooth because the logic gates on the FPGA device were organized as 4-input LUTs-fan-ins of size multiples of 4 tend to perform better than the overall trend. The loop-back transition around the `op_union` (in the `has_loop` series) connected every state output of the union operator to every input state of that operator. This resulted in more complex routing and further impacted the clock frequency.

Overall our experiments show that the REME construction algorithms proposed in [10] generated FPGA circuits with high clock frequency and high LUT efficiency for large number of highly complex regular expressions.

7. Conclusions

We presented a software toolchain which automates the construction and optimizations of regular expression matching engines (REMEs) on FPGA. The software accepts a potentially large number of regular expressions as input and generates RTL codes in VHDL as output, which could be accepted directly by FPGA synthesis and implementation tools. The automated REME optimizations include centralized character classifications, multi-character matching, and staged pipelining. We also developed a benchmark generator to produce REMEs of configurable pattern complexities to evaluate the performance of the software.

On a 2 GHz Athlon 64 PC, our software generates a compact and high-performance REME circuit matching over a thousand regular expressions in just a few seconds. Extensive studies showed that the two-dimensional staged pipeline effectively localized signal routing and achieved a clock rate over 300 MHz while processing hundreds of REMEs in parallel.

Acknowledgment

This work was supported by U.S. National Science Foundation under Grant CCR-0702784.

References

[1] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 3rd International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '07)*, pp. 1–12, New York, NY, USA, December 2007.

[2] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 119–126, Bangkok, Thailand, December 2006.

[3] B. C. Brodle, R. K. Cytron, and D. E. Taylor, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA '06)*, pp. 191–202, Boston, Mass, USA, June 2006.

[4] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware,"

in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '02)*, pp. 1–111, 2002.

[5] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[6] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '07)*, pp. 155–164, 2007.

[7] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '07)*, pp. 127–136, New York, NY, USA, 2007.

[8] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 227–238, 2001.

[9] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL '08)*, pp. 131–136, August 2008.

[10] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08)*, November 2008.

[11] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '06)*, pp. 93–102, San Jose, Calif, USA, December 2006.

[12] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 2, pp. 12–17, European Design and Automation Association, Leuven, Belgium, 2006.

[13] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '07)*, pp. 145–154, 2007.

[14] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM '08)*, pp. 207–218, Seattle, Wash, USA, August 2008.

[15] R. W. Floyd and J. D. Ullman, "The compilation of regular expressions into integrated circuits," *Journal of the ACM*, vol. 29, no. 3, pp. 603–622, 1982.

[16] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 249–257, Napa, Calif, USA, April 2004.

[17] "PCRE: Perl Compatible Regular Expression," http://www.pcre.org.

[18] "Snort network instrusion detection," http://www.snort.org.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 572860, 13 pages
doi:10.1155/2009/572860

Research Article

A Hardware Filesystem Implementation with Multidisk Support

Ashwin A. Mendon, Andrew G. Schmidt, and Ron Sass

Reconfigurable Computing Systems Lab, University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, NC 28223-0001, USA

Correspondence should be addressed to Ashwin A. Mendon, aamendon@uncc.edu

Received 16 March 2009; Accepted 4 August 2009

Recommended by Cesar Torres

Modern High-End Computing systems frequently include FPGAs as compute accelerators. These programmable logic devices now support disk controller IP cores which offer the ability to introduce new, innovative functionalities that, previously, were not practical. This article describes one such innovation: a filesystem implemented in hardware. This has the potential of improving the performance of data-intensive applications by connecting secondary storage directly to FPGA compute accelerators. To test the feasibility of this idea, a Hardware Filesystem was designed with four basic operations (open, read, write, and delete). Furthermore, multi-disk and RAID-0 (striping) support has been implemented as an option in the filesystem. A RAM Disk core was created to emulate a SATA disk drive so results on running FPGA systems could be readily measured. By varying the block size from 64 to 4096 bytes, it was found that 1024 bytes gave the best performance while using a very modest 7% of a Xilinx XC4VFX60's slices and only four (of the 232) BRAM blocks available.

1. Introduction

Many FPGA devices available today are rich in special-purpose blocks. These fixed-function cores, implemented on the die, add capabilities to the ever-growing capacity of these devices. The Virtex-4 FX devices, for example, in addition to the conventional programmable logic and flip-flops, include processors, Block RAMs, DSP48 cores, and high-speed serial transceivers. The Multigigabit Transceiver (MGT) cores are especially interesting because they allow for a wider range of high-speed serial peripherals, such as disk drives, to be directly connected to the devices.

These advances enable a designer to implement highly integrated computing systems. For example, it is feasible to integrate video, networking interfaces, disk controllers [1], and other conventional peripherals onto a single Platform FPGA device running a mainline Linux kernel. In fact, several high-performance computing researchers are currently investigating the feasibility of using Platform FPGAs as the basic compute node in parallel computing machines [2–4]. If successful, there is an enormous potential for reducing the size, weight, and cost while increasing the scalability of parallel machines.

Within the context of parallel computing, the integration of disk drives is especially interesting because of its potential to speed up data-intensive parallel applications. In particular, out-of-core applications (MPI-IO) needing tightly integrated secondary storage or streaming very large data sets would benefit. Tight integration, specifically, is important to these high-performance computing applications for a number of reasons. It allows FPGA computational cores to consume data directly from disk without interrupting the processor (or traversing the operating system's internal interfaces). It also allows the introduction of simple striped multidisk controllers (without the cost or size of peripheral chipsets). Finally, it is possible to coordinate disks attached to multiple discrete FPGA devices—again, without depending on the processor.

Filesystems are typically implemented in *software* as part of the operating system. This paper describes the implementation of a *hardware* filesystem. Figure 1 illustrates this concept. Figure 1(a) is the traditional organization with the filesystem and device driver implemented in software, Figure 1(b) is the filesystem migrated into hardware. The simplest filesystems organize the sequential fixed-size disk sectors into a collection of variable-sized *files*. Of course,

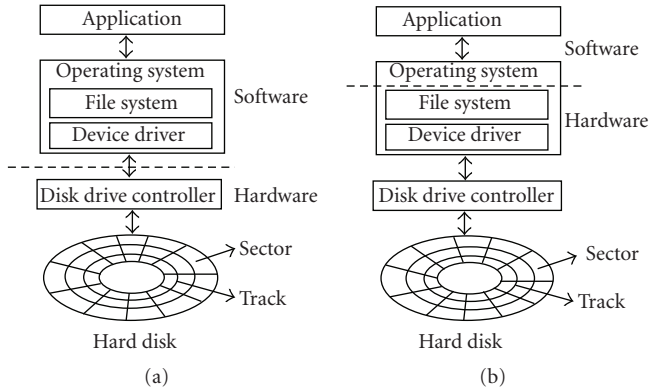


Figure 1: (a) Traditional filesystem implementation. (b) Filesystem migrated into programmable logic.

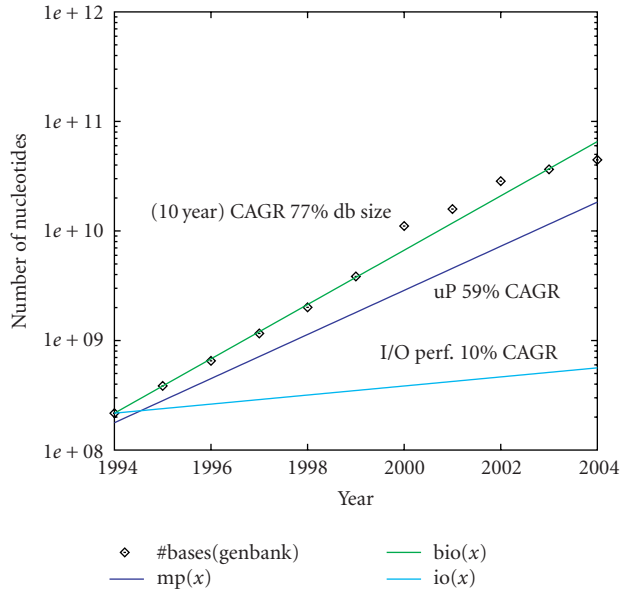


Figure 2: Compound annual growth rate of problem size, single processor performance, and I/O subsystem performance.

most modern filesystems are much more complex and also include a large amount of metainformation and further organize files into a hierarchy of directories. The design presented here, however, is narrowly defined to support high-performance computing. This is not a particularly serious weakness since SRAM-based FPGA devices can be reprogrammed to incorporate new features in hardware.

For some scientific applications, these features are extremely valuable. For example, in some cases, the resolution of experiment or simulation is limited by the main memory available to store the data structures. In order to increase the detail of the simulation, computational scientists are forced to code their algorithms so that data are explicitly moved between secondary storage and main memory. (These so-called out-of-core applications are far more efficient than simply relying on the OS to swap memory to secondary storage.)

Alternatively, if part of the computation is performed by accelerators implemented in the programmable logic of an FPGA, then the data do not necessarily have to go through all of the traditional layers of an OS (device driver, filesystem interface) just to have the application then forward it to the core. Instead, the core can simply open the file and access it directly. This frees the processor from handling I/O requests and avoids the use of off-chip memory bandwidth to buffer disk data as in other approaches [5]. It also reduces the number of interrupts which has been shown to negatively impact very large parallel systems [6, 7]. Finally, by migrating part of the filesystem operations to hardware, it becomes feasible to handle remote disk access directly in hardware. (Again lowering the number of interrupts the processor sees and avoids wasting memory bandwidth to buffer data between disk and network subsystems.) In short, this approach has the potential of increasing the bandwidth from disk to core, lowering the latency, and reducing the computational load on the processor for a large number of FPGA devices configured for high-performance computing.

In a third case, some applications are facing datasets that are growing faster than computer speeds. Consider processor speeds and the size of bioinformatic databases. Suppose that single processor performance continues to double every 18 months and biological databases are growing even faster. Figure 2 shows both growth rates of between 1994 and 2004 on a semilog graph. The nucleotide data points come from GenBank [8], a public collection of sequenced genomes. A line fitted to this data shows a compound annual growth rate of 77% (compared to the 59% annual growth rate of processors). Now consider the performance gains of I/O subsystems (disk and interface). Secondary storage is not keeping pace with processor speeds, let alone the growth rate of the biological databases. The most aggressive estimates [9] suggest a 10% compound annual growth rate in performance while others [10] suggest a more modest 6% growth rate. Regardless, the consequence is profound: the same question (e.g., *is this sequence similar to any known gene*?) will take longer and longer every year. In short, the problem size is growing so fast; the bottleneck is simply I/O bandwidth. A filesystem implemented in hardware will not directly address

this issue; however, it does offer a first step towards alleviating high I/O bandwidth needs.

Our approach to this investigation is broken down into three steps: first, a software implementation as a basic proof of concept of how the filesystem in hardware will operate, second, a simulation of the hardware filesystem, and third, a synthesized implementation running on an FPGA.

The rest of this paper is organized as follows. The next Section is a background section on filesystems, specifically the Unix filesystem. In Section 3, the three designs (software, simulation, and synthesized) implementations are presented. Section 4, explains the testing methodology followed by the presentation and analysis of the results. The paper concludes with a brief summary and future directions.

2. Background

2.1. Disk Subsystem. The main purpose of a computing system is to create, manipulate, store, and retrieve data. As such, filesystems have been central to most modern computing systems. Filesystems are responsible for managing and organizing files on a nonvolatile storage medium, such as a Winchester-type disk drive (also known as a hard disk or hard drive). Files are composed of bytes and the filesystem is responsible for implementing byte-addressable files on block-addressable physical media, such as disk drives. Key functions of a filesystem are (1) efficiently use the space available on the disk, (2) efficient run-time performance, and (3) perform basic file operations like create file, read, write, and delete. Of course most filesystems also provide many more advanced features such as file editing, renaming, user access permission, and encryption to name a few.

The hardware filesystem implemented is loosely modeled after the well-known UNIX filesystem (UFS) [11]. UFS uses logical blocks of 512 bytes (or larger multiples of 512). Each logical block may consist of multiple disk sectors. Logical blocks are organized into a filesystem using an Inode structure that includes file information (such as file length), a small set of direct pointers to data blocks, and a set of indirect pointers. The indirect pointers point to logical blocks that consist entirely of pointers. UFS uses a mutilevel indexed block allocation scheme which includes a collection of direct, single indirect, double indirect, and triple indirect pointers in the Inode. The filesystem layout is as shown in the Figure 3.

Normally, the filesystem is designed to be independent of the disk controller. The disk controller is typically a device driver in an operating system that is responsible for communicating with the physical media and responds to block transfer commands from the filesystem. For expediency, the work here focuses on the most common, commodity drives available today: Serial ATA (SATA). SATA provides a 4-wire point-to-point configuration, supporting one device per controller connection. Each device has dedicated bandwidth and there are no master/slave configuration jumper issues as with parallel ATA drives. The pin count is reduced from 80 pins to 7 pins having 3 ground lines interspersed between 4 data lines to prevent crosstalk. Several FPGA devices include high-speed serial transceivers. For example, the Xilinx Virtex II, Virtex-4, and Virtex-5 device families have members that include multigigabit transceiver cores. These cores can be configured to communicate via the SATA protocol at the physical layer. There are commercial IP cores available to do this.

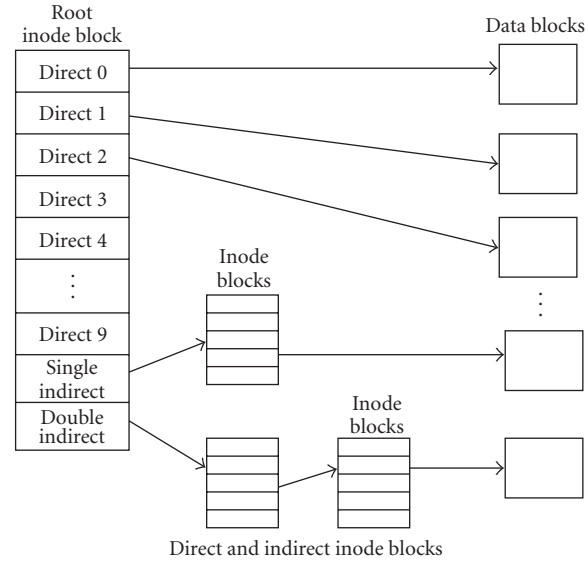


Figure 3: UNIX Inode structure.

2.2. Related Work. The hardware filesystem architecture described in this paper is, to the authors' knowledge, novel and unique. However, there are several research efforts pursuing related goals. These efforts are described below.

Work at the University of California, Berkeley, describes BORPH's kernel filesystem layer [5] which enables hardware cores to access disk files by plugging into the software interface of the operating system via a hardware system call interface. However, the cores still have to traverse the software stack of the OS. The approach proposed here allows the hardware cores direct access to disk by implementing the filesystem directly in hardware.

The Reconfigurable parallel disk system implemented in the RDisk project [12] provides data filtering near disks for bioinformatics databases by using a Xilinx Spartan 2 FPGA board. While this is relevant for scan algorithms which read in large datasets, it does not provide the capabilities of a filesystem such as writing and deleting files.

Using FPGAs to mitigate the I/O bandwidth bottleneck has been of interest commercially among server vendors such as Netezza [13] and Bluearc [14]. Netezza database storage servers have a tight integration of storage and processing for SQL-type applications by having FPGAs chips in parallel Snippet Processing Units (SPUs). These provide initial query filtering to reduce the I/O and network traffic in the system. Bluearc's Titan 3000 network storage server uses a hardware-based filesystem to speed up the I/O interface.

Finally, well-known RAID storage solutions have either hardware or software controller managing data across multiple disks. However, these solutions operate on a single

I/O channel or bus [15] and still traverse the operating system's software stack. While this can be used to improve disk performance, it does not necessarily improve disk to compute accelerator performance. Moreover, the approach proposed here has the ability to be directly integrated into the network subsystem of a parallel machine—allowing multiple I/O channels in a parallel filesystem implementation.

3. Design

To mitigate risk, the design and implementation of the Hardware Filesystem (HWFS) was staged. The first stage focused on a software reference design and hardware simulations to judge the feasibility. This was reported in [16]. The work here describes a design that correctly synthesizes for an FPGA, includes a RAM Disk core to emulate the behavior of a disk controller, and has support for multiple disks in a RAID0 configuration. Below is a high-level description of the Hardware Filesystem followed by a description of the RAM Disk core and multidisk extensions.

3.1. Hardware Filesystem Core. As mentioned in the previous section, the layout of the UNIX filesystem was the initial starting point for the HWFS described here. However, UFS was designed to be general-purpose whereas the aim of this work is more narrowly focused on feeding streams of data to computational accelerators. This has led to a number of differences. First, the Hardware Filesystem uses only direct and single indirect pointers in its inodes and the last indirect logical block points to another block of indirect pointers. Essentially, after the initial pointers in the inode are exhausted, the system reverts to a linked-list structure for very large files. This layout is shown in Figure 4. A second difference in the Hardware Filesystem is that the file names are merged into the *Super Block* along with filesystem metadata such as freelist head and freelist index. The UFS supports a hierarchy of directories and subdirectories but the HWFS described here is flat.

A high-level block diagram of the HWFS is shown in Figure 5. It consists of a single large state machine, a buffer for the *Super Block*, a buffer for the currently open *Inode*, and a *Freelist* buffer. A compute accelerator presents commands and parameters to the core; the core is responsible for making disk controller block requests and delivering that data associated with a compute accelerator's request.

Specifically, the state machine implements the Open, Read, Write, and Delete file operations. The *operation* port is driven by the compute accelerator to select from the four main operations. The state machine asserts *request* and *new blknum* signals to issue a new block request to data from the disk controller. A *command* signal is used to distinguish between read and write block requests. The logical block numbers are issued from the *blocknum* port to address the appropriate memory location. The HWFS core waits on a *cmd ack* signal before issuing the next block request. On completion of a block transaction, the memory interface asserts the *blk xfer done* signal.

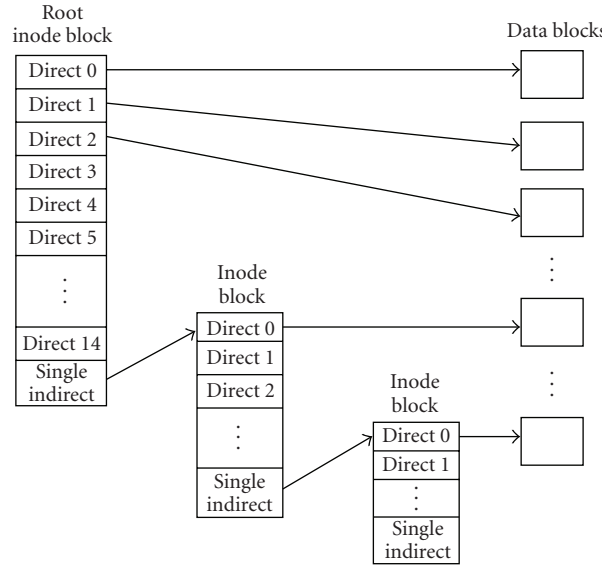


Figure 4: HWFS Inode Structure.

The HWFS core has an additional functional improvement over what was reported in [16]. Support for multiple disks has been provided through the use of split transactions on disk controller. This allows multiple block requests to be issued to the memory subsystem. An internal counter in the FSM keeps track of outstanding block transactions. Subsection 3.2 discusses the multidisk controller and integration with the HWFS in more detail.

The rest of this subsection describes the various operations and the relevant states in the top-level state machine.

Open File. Open File takes in a file name along with an open command and returns the file's *Root Inode* location. The open command is required before issuing the first read or write command for a file. After the file is open, any number of reads and writes can be issued.

The state machine starts from the *Read Super Block* state and reads blocks 0–3 from *disk* into the *Super Block* buffer 4 bytes at a time. After reading 4 blocks, it transitions to the *Match Filename State* which starts a linear search for the 8 byte filename. The FSM sequences through the Super Block buffer, starting at the first file slot and reads the BRAM contents into a 64-bit equality comparator. If a match is found with the required filename, the comparator sends a *found* signal to the FSM which transitions to the *Find Inode* state. If the search is unsuccessful, the file does not exist in the filesystem Super Block and a *file not found* signal is asserted. In *Find Inode* state, the state machine captures the file's *Root Inode* location from the filename-inode mapping in the Super Block. Figure 6 depicts the open operation.

Read File. Once the file is opened, Read File uses the file's root inode block location to read in the file contents into a Read FIFO. Read File (Figure 7) begins with the *Read Inode* state. The *Root Inode* block of the file is first fetched from *disk* into the *Inode* buffer using the inode location

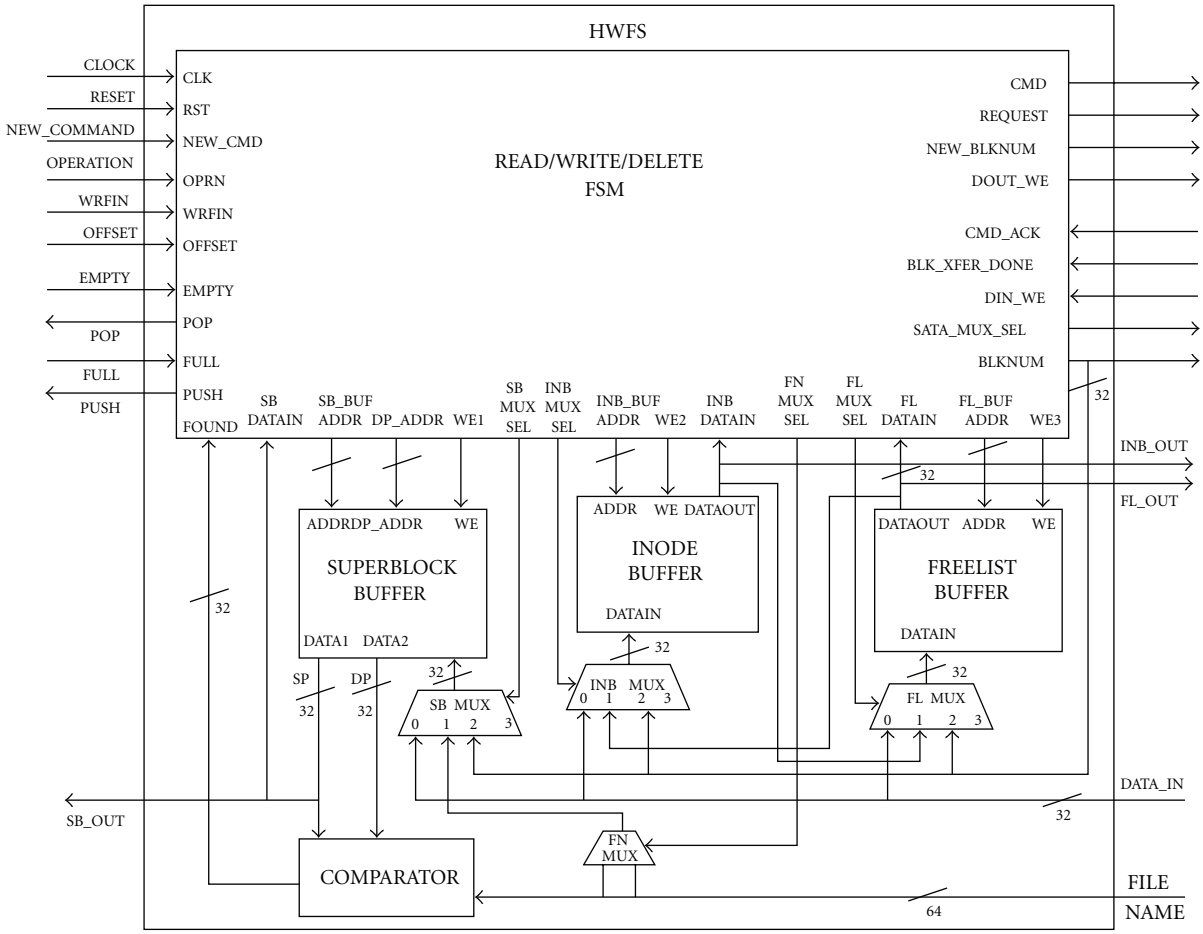


Figure 5: Hardware filesystem core: block diagram.

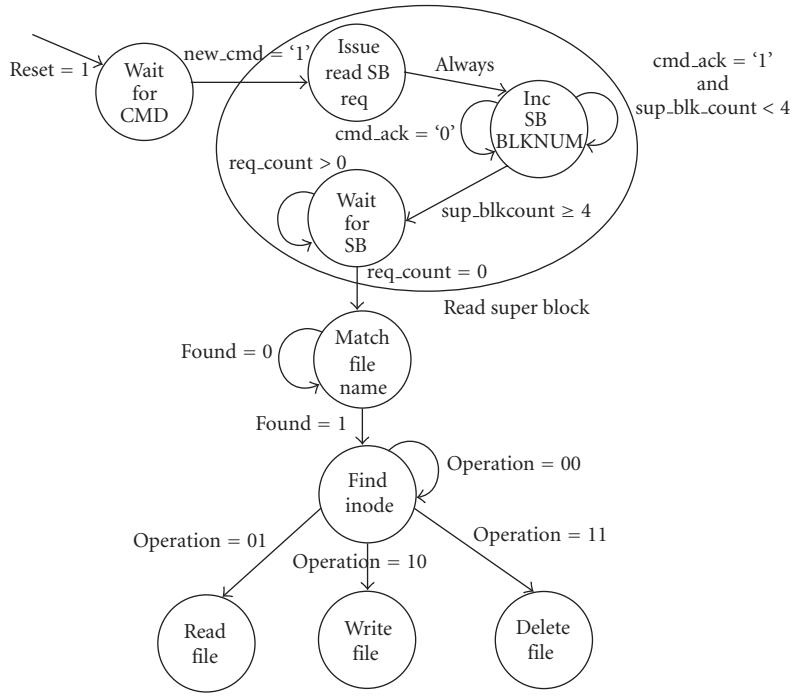


Figure 6: Open file state machine.

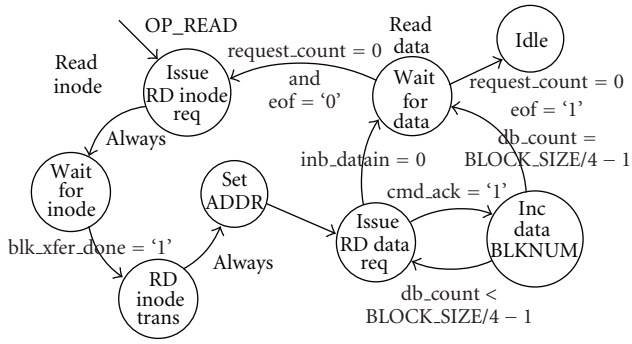


Figure 7: Read file state machine.

number as a blockid. The state machine then transitions to the *Read Data* state via an intermediate state: *Set Address* which accounts for the delay between setting the BRAM address and reading the output blockid. In *Read Data* state, the inode block is read sequentially, 4-bytes at a time. The output inodes are fed back to the state machine and used as blockids for fetching data blocks from *disk* into the Read FIFO. The last inode in the root inode block links to the next inode block of the file. The FSM uses this link inode for fetching the next inode block of the file by cycling back to the *Read Inode* state. The data blocks are read until an inode 0 is found which signals the end of file. The state machine then returns to the *idle* state.

Write File. Write File shown in Figure 8 either creates a file if it does not exist on the disk or appends data to an existing file. Once the file has been opened, the state machine extracts the filesystem metadata from the *Super Block* buffer in *Read FSMD* state. The *Read Free Block* state reads the first block from the freelist into the *Freelist* buffer. The FSM then goes to *Write Data Block* state where the data blocks are written from the *Write FIFO* out to disk using free blockids from the freelist buffer. The blockid is also simultaneously stored as an *Inode* in the *Inode* buffer. In *Write Inode Block*, the inode buffer's contents are written to disk and a *Root Inode* block for the file is created. As the file size increases, subsequent inode blocks of the file are created and added as a linked list to the root inode block. The file's inode blocks thus exist in the form of a linked list interspersed among the data blocks and freelist blocks. After the data blocks and inode blocks are written, the filesystem metadata and super blocks are written to close the file so that HWFS is ready for the next operation.

Delete File. In Delete file, shown in Figure 9, the first block from the freelist is read into the *Freelist* buffer and the *Root Inode* block is read into the *Inode* buffer. The blockids from *Inode* buffer are then transferred to *Freelist* buffer starting from the freelist index, until it is full of free blockids. Next, the contents of the freelist buffer are stored on disk. The FSM reads the Super Block to delete the file name and update the filesystem metadata. The new *freelist head* and *freelist index* are written to the Super Block buffer and then transferred to *disk*.

3.2. Multidisk and RAM Disk Support. To further explore the feasibility and functionality of the HWFS core, a synthesized and operational design was required. However, commercial SATA disk controller cores are expensive and difficult to justify for a feasibility study. To make experiments—especially experiments with multiple disks—more feasible, a RAM Disk core was developed.

Figure 10 illustrates a high-level block diagram of the system using the hardware filesystem and a SATA disk. The processor and computation core are both capable of interfacing with the hardware filesystem across the system bus. While the HWFS is targeted for a SATA hard disk, the HWFS core itself is designed with a generic interface to increase the number of devices that can be potentially interfaced with, beyond a hard disk. The Xilinx ML-410 FPGA board [17] provides interfaces for both ATA and SATA disks; however, to focus on the HWFS development the more complex ATA and SATA interfaces have been replaced with a RAM Disk.

Purpose of the RAM Disk. When presenting a hardware filesystem, it would be assumed that the data would be stored on a hard disk. In these tests we have opted to use a specially designed RAM Disk in place of the SATA hard disk. One might ask why to read about a disk-less hardware filesystem. To this seemingly simple question, we would like to explain our reasoning for the lack of a hard disk in the currently implemented design. First and foremost is the cost of the SATA IP core. While SATA IP cores are currently for sale [18], they are prohibitively expensive to purchase outright without any indication that the money would be well spent. Second is the design complexity of having to both create a hardware filesystem and integrate it with the SATA core in order to test even the simplest of file operations. Finally, while SATA may currently be the forerunner in the market, trends may soon shift to alternative disks and interfaces which could cause another redesign of the system.

Our implementation attempts to minimize initial cost and risk by focusing first on the design of the hardware filesystem. In simulation creating a simple SATA stub, which mimics some of the simple functionality of the SATA interface, enables a more rapid development of the hardware filesystem. In hardware there is no SATA stub; instead a fake disk must be created. External SDRAM presented itself as the ideal candidate with its easy and well-documented interface. This RAM Disk is not targeted to be competitive with an actual hard disk, nor is it the long term goal of the Hardware Filesystem to include the RAM Disk. It simply provides an interface to large, off-chip storage that would allow for better testing of the Hardware Filesystem running on an actual FPGA. The data stored within the RAM Disk—super, inode, data, and free blocks—are the same as the data that would be stored on that of a SATA disk. The key differences being the on-chip controller's interface and the data being stored in DDR2 instead of a physical disk.

As a result of the RAM Disk interface, we are now able to support any storage device by bridging the Hardware

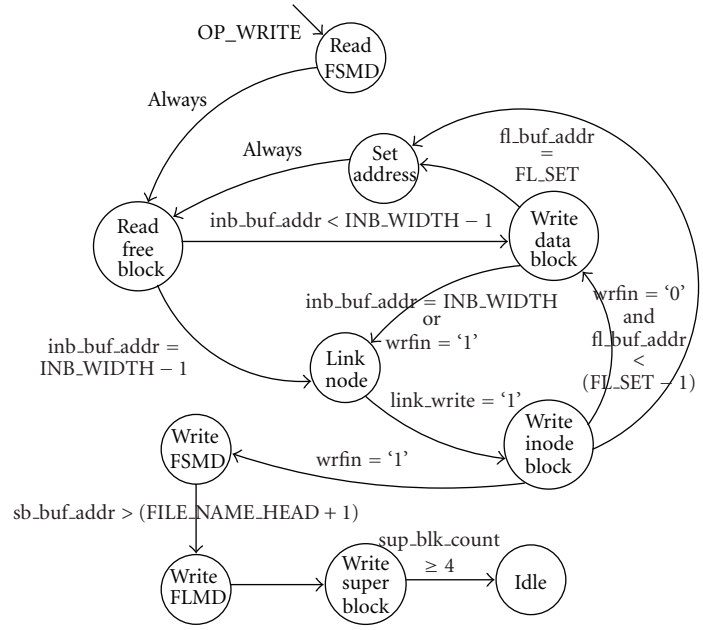


Figure 8: Write file state machine.

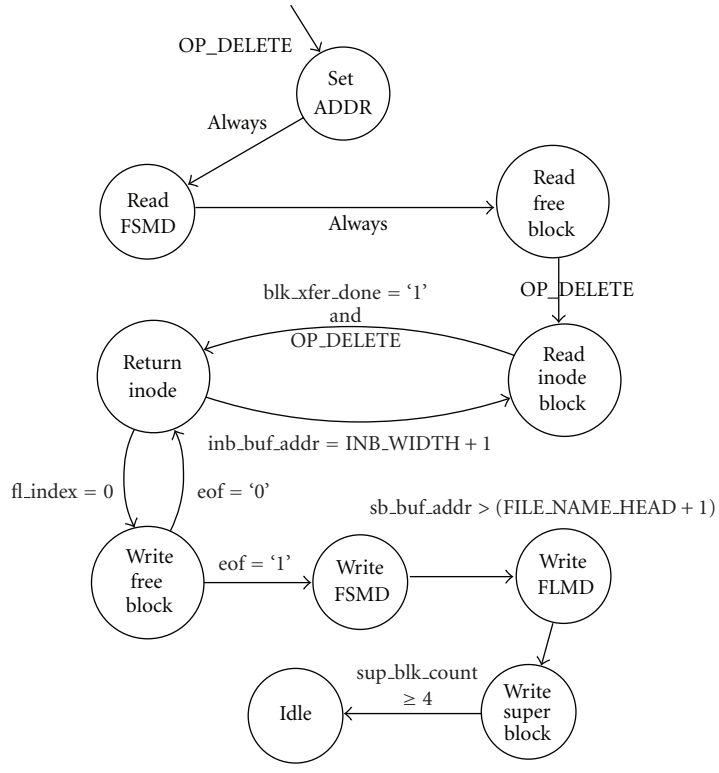


Figure 9: Delete file state machine.

Filesystem's interface with the storage device's interface. This can be seen in Figure 11. While the complexity of the interfaces might be difficult to design, it should not be impossible, merely time consuming. The advantage of such an approach is that with a working hardware filesystem the disk interface would take focus, reducing the number of unknowns in the design.

Finally, we do not aim to use the RAM Disks for performance. It should be obvious that the time to access a hard disk (rotational delay + seek time) will be constant between both a typical operating system's filesystem and the hardware filesystem. Therefore, a straightforward test between the two filesystems is not immediately possible. Instead, what we show is the efficiency of the hardware filesystem.

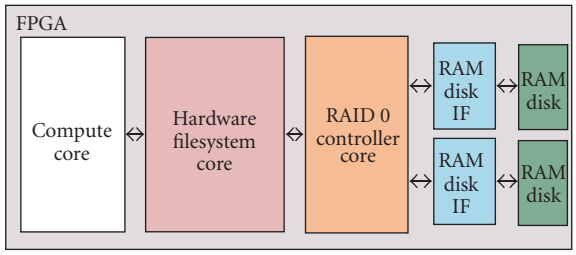


Figure 10: Hardware system level interface between the SATA core and hardware filesystem.

System Implementation. A hardware base system has been created consisting of a processor, system bus, on-chip memory, external memory, and the HWFS core. Linux 2.6 has been compiled and configured to run with this hardware base system; however, it is important to note that Linux is being used as the test bench and is not using the HWFS core. Eventually, considering the results from these tests, Linux will use the HWFS core in place of its current software file system.

Figure 11(a) depicts the high-level interface between the HWFS and the RAM Disk. Between the HWFS and the RAM Disk lies the Native Port Interface (NPI) to provide a custom, direct interface to the memory controller. The memory controller is a conventional soft IP core which communicates with the external memory. Requests from the HWFS are in the form of block transfers and it is the NPI which converts those block transfers into physical memory transfers.

Figure 11(b) highlights the flexibility of the HWFS core's design. Purchasing a SATA controller IP core and creating a simple interface between the HWFS and the SATA controller all that is necessary to port the RAM Disk implementation to a SATA implementation. Likewise, for any additional secondary storage the same process would apply.

Adding Multiple Disk Support. To support multiple disks a Redundant Array of Independent Disks (RAID) [19] Level 0 controller has been designed and synthesized for the FPGA. RAID 0 stripes data across n number of disks but does not offer fault-tolerance or parity. RAID 0 was chosen for this design as a first-order proof of concept to investigate the question how hard is it to add multiple disk support to the current Hardware Filesystem design? The initial design of the Hardware Filesystem core only supported access to a single disk, not a limitation, but instead a design choice to focus on the HWFS's internal functionality.

To provide support to multiple disks a handshaking protocol was established between the HWFS and the RAID 0 controller. Since the number of disks in the RAID system is unknown to the HWFS, requests should be issued as generically as possible. The handshaking protocol requires the HWFS to wait for a request acknowledge from the RAID controller before issuing subsequent requests. Initial designs with a single disk did not require this handshaking since only one request was in process at any given moment.

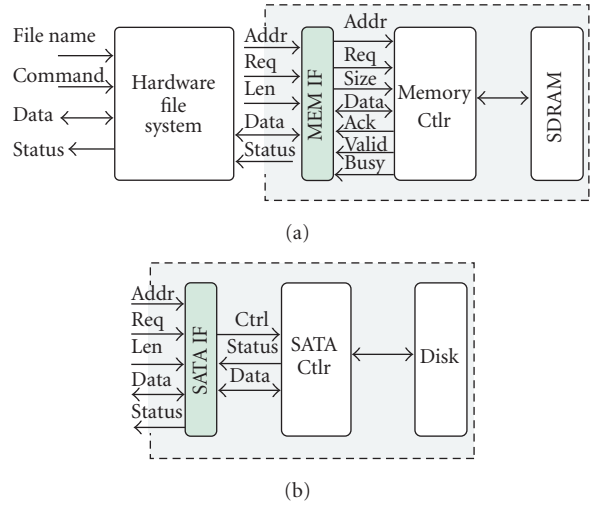


Figure 11: (a) System interface with RAM disk (b) Modular interface with SATA.

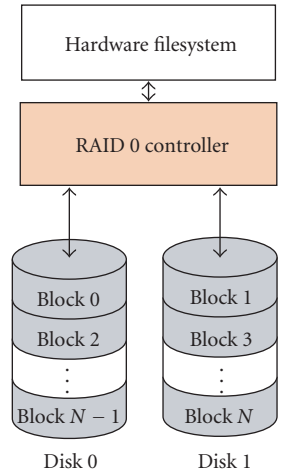


Figure 12: Hardware Filesystem connected to the RAID 0 Controller which stripes the data blocks across two disks.

To illustrate the RAID 0, Figure 12 shows the Hardware Filesystem connected to the RAID 0 controller which is connected to two disks—this paper presents support for N disks, but only two RAM Disks have been tested running on the FPGA at the time of this writing. The stripe size in this design is one full block, but subblocks could be just as easily used. The RAID controller has been designed with a generic interface to allow easy support of any number of disks; limitations on the Xilinx ML-410 forced physical tests on the FPGA to two disks. More extensive tests of systems with greater than two disks have been performed and verified in simulation.

For a RAID controller with multiple disks, each read or write transaction could be to the same disk or to a different disk. For requests to the same disk the transactions are serialized, requiring the first transaction to complete before the second transaction can commence. For two requests to

two separate disks, both requests can be issued in parallel. On a read request the RAID controller must also make sure that the blocks are returned in the correct order since it is possible for two concurrent requests to be returned out of order.

With the successful integration of the RAID 0 controller, it is feasible to integrate more sophisticated controllers which offer parity, fault-tolerance, and mirroring of data in future designs. These higher RAID levels would still likely use the same interface to the HWFS core as the RAID 0; the difference would be the functionality within the RAID controller core itself.

4. Experimental Setup and Results

To establish whether implementing a filesystem directly in hardware provides sufficient improvements in latency and bandwidth while utilizing limited chip resources, we simulated and synthesized the VHDL design code and ran the design on a Xilinx ML-410 FPGA Development board. The experimental setup, results obtained, and analysis follows.

4.1. Simulation Setup and Results. The functionality of the design was first verified in simulation with a VHDL testbench and a *satastub* behavioral model. The testbench instantiates the top level structural VHDL module of the design. It then creates a 100 MHz master clock signal to synchronize the design and provides a reset pulse to initialize the state machine. Next, a test process generates an input test sequence to exercise the design. This includes a 64-bit file name for opening the required file from the disk and a 2-bit operation signal to select from one of the four operations: open, read, write, and delete. The state machine transitions to the idle state and asserts the stop-simulation signal on completing the operations. The testbench checks for this signal and reports a "Testbench Successful" message along with the iteration time. ModelSim verification environment, version 6.3b, running on a Linux Workstation was used for simulation and debugging.

To evaluate the amount of overhead induced by the filesystem itself the execution times of sequential read and write operations were measured in simulation with an ideal disk for file sizes ranging from 1 kilobytes to 5 gigabytes shown in Table 1.

The filesystem's efficiency was computed as the ratio of the time taken to transfer raw data blocks of a file between the HWFS and disk to the total transfer time with the filesystem's processing overhead:

$$\text{eff} = \frac{\text{raw block transfer time}}{\text{filesystem block transfer time}},$$

$$\text{raw block transfer time} = \frac{\text{file size} \times \text{clock cycle time}}{\text{bytes per clock cycle}}. \quad (1)$$

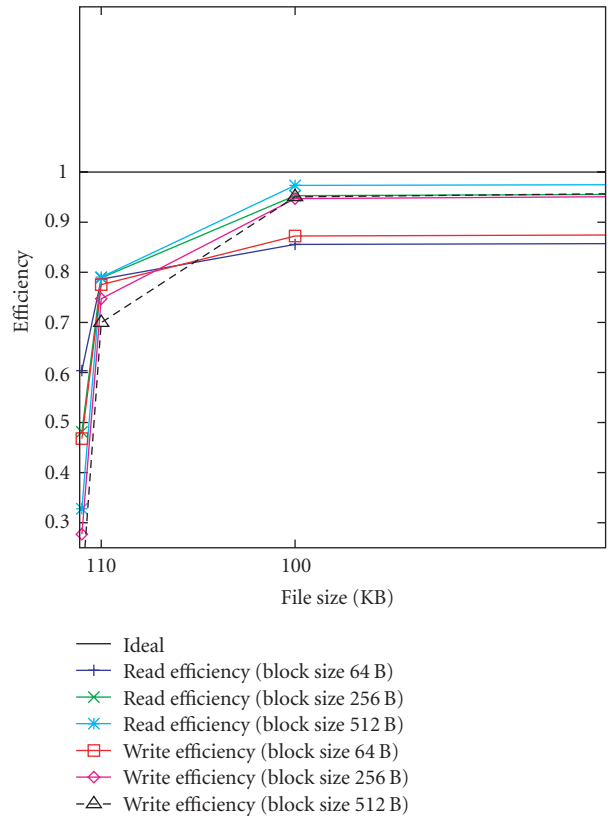


Figure 13: HWFS sequential read/write efficiency in simulation plotted against different file sizes.

The overhead includes the time taken to read the Super Block, find a file name match, get its root inode block (open file operation), read the inode blocks of the file (read file operation), and read/write free blocks and inode blocks (write file operation). Figure 13 shows a plot of the sequential read and write efficiencies for 64 B, 256 B, and 512 B sized blocks. It is observed that for small files (1 KB to 10 KB) the efficiency is below 80%. It increases to 95% for 100 KB files and saturates for very large files (shown by a flattening of the plot for file sizes beyond 100 KB). This is due to the overhead having little effect on the execution times for large files thereby achieving efficient run-time performance (to emphasize the transition in efficiency, the x-axis is restricted to 250 KB in the figure).

4.2. Synthesis Setup and Results. The setup for the system running on the ML-410 builds upon the description given in Section 4.1. The test is running on a Linux-based system which requires a device driver to allow the test application to communicate with the HWFS core. The test begins with the PowerPC initializing the RAM Disk with the empty root filesystem. The PowerPC communicates directly with the RAM Disk since it is a volatile storage device and it is necessary to format the RAM Disk. Once the RAM Disk has been initialized, the PowerPC's test application exercises the HWFS via the device driver. The test application simply issues multiple *open*, *read*, *write*, and *delete* commands to the

Table 1: HWFS sequential read/write execution time in simulation with different block sizes.

	Read			Write		
File Size (Bytes)	64 B	256 B	512 B	64 B	256 B	512 B
1 KB	4.24 μs	5.32 μs	7.81 μs	5.47 μs	9.24 μs	15.52 μs
10 KB	32.56 μs	32.45 μs	32.39 μs	33.02 μs	34.26 μs	36.6 μs
100 KB	299.2 μs	268.6 μs	263 μs	293.4 μs	270.3 μs	269.3 μs
1 MB	3.03 ms	2.71 ms	2.67 ms	2.96 ms	2.7 ms	2.65 ms
10 MB	30.12 ms	26.8 ms	26.6 ms	29.6 ms	26.5 ms	26.5 ms
100 MB	300.4 ms	266.4 ms	264.3 ms	295 ms	263 ms	262.5 ms
1 GB	2.98 s	2.7 s	2.69 s	2.96 s	2.65 s	2.63 s
5 GB	14.6 s	13.6 s	13.5 s	14.4 s	13.2 s	13 s

Table 2: Hardware filesystem with a single disk resource utilization synthesized for the XC4VFX60.

Block Size	Slices	LUTs	F/Fs	BRAMs
64 B	1378	2546	871	2
128 B	1302	2442	872	3
256 B	1254	2350	874	3
512 B	1335	2515	882	3
1024 B	1357	2559	887	4
4096 B	1317	2483	904	14

Table 3: Hardware filesystem with multiple disk resource utilization with Block Size 1024 synthesized for the XC4VFX60.

Resources	HWFS	RAID Ctlr	RAM Disk IF
Slices	1357	343	678
F/Fs	887	255	601
LUTs	2559	626	1253
BRAMs	4	0	2

HWFS core. After the test finishes, the PowerPC reads the RAM Disk to verify the successful completion of the test.

The VHDL design description was synthesized for varying block sizes between 64 and 1024 bytes using the Xilinx Synthesis Tool (XST) available in the Xilinx ISE design suite, version 10.1, for the target device XC4VFX60-11ff1152 from the Virtex-4 family to generate the Xilinx specific NGC files. Table 2 shows the resource utilization statistics with these varying block sizes. Since the super block, inode, and freelist buffers are mapped onto BRAMs, the logic resource (slice) utilization is independent of block size. A slight variation in slice count is observed due to the BRAM buffer's address width variations and the synthesis tool's speed optimization efforts. Based on the synthesized resource utilization results, the largest block size without excessive BRAM usage is 1024 Bytes. At a block size of 4096 Bytes a total of 14 BRAMs are used. In a filesystem with a large number of small files, 4096 Byte blocks would possibly introduce fragmentation; however, the HWFS focuses on opening relatively few large files, orders of magnitude greater than the block size. As a result the block size is less of a restriction as the BRAM resource utilization.

Table 3 shows the resource utilization breakdown for the Hardware Filesystem, RAID Controller, and RAM Disk interface with a block size of 1024 Bytes accessing two disks. In this design the RAID Controller connects to two RAM Disk interface cores which each connect to two external SDRAM DIMMs. In this configuration the HWFS and RAID Controller use a modest 7% of the slices while only using four BRAMs. The RAM Disk interface uses two BRAMs to buffer sending and receiving data between the HWFS and RAM.

Single RAM Disk Results. Table 4 gives the execution measurements for read/write operations with a single RAM Disk synthesized and run in hardware. Unlike the simulation tests, the RAM Disk is not an ideal disk and the execution times increase accordingly. For a real SATA hard disk these numbers would again increase; however, the importance of this test is to show that running in actual hardware produces similar trends to simulation when taking into account the storage media's access times.

Table 5 is presented to highlight the time taken by the filesystem to process data in comparison with the RAM Disk memory transaction time. For a write operation the execution time of the Hardware Filesystem is 5.54 microseconds compared to the simulation time of 5.47 microseconds (Table 1). This shows that the Hardware Filesystem is able to maintain the same performance with a RAM Disk as with the simulation's ideal disk. The same holds true for the read operation.

The efficiency of the Hardware Filesystem with a single RAM Disk is shown in Figure 14. The HWFS stalls until both of the block requests to and from memory are satisfied. Due to this added memory transaction latency, the efficiency graph shows a dip in performance as compared to the simulation efficiency in Figure 13.

Multiple RAM Disks Results. The split transactions implemented for multidisk support provides an improvement over the single disk efficiency. Test results and the efficiency graph for read/write operations over two RAM Disks are shown in Table 6 and Figure 15. The limiting factor on the number of RAM Disks in the multiple RAM Disk test is based on the Xilinx ML-410 development board consisting of two

Table 4: Hardware filesystem read/write execution time with single RAM Disk synthesized for the XC4VFX60.

	Read			Write		
File Size (Bytes)	64 B	512 B	1024 B	64 B	512 B	1024 B
1 KB	9.28 μs	12.54 μs	19.62 μs	9.4 μs	28.3 μs	51.77 μs
10 KB	73.59 μs	45.8 μs	51.28 μs	52.3 μs	59.55 μs	83.02 μs
100 KB	709.84 μs	380.97 μs	366.32 μs	483 μs	391.56 μs	396.65 μs
1 MB	7.18 ms	3.76 ms	3.55 ms	4.9 ms	3.57 ms	3.54 ms
10 MB	71.8 ms	37.44 ms	35.35 ms	48.97 ms	35.48 ms	34.82 ms
100 MB	717.93 ms	374.33 ms	353.32 ms	489.65 ms	354.53 ms	347.69 ms

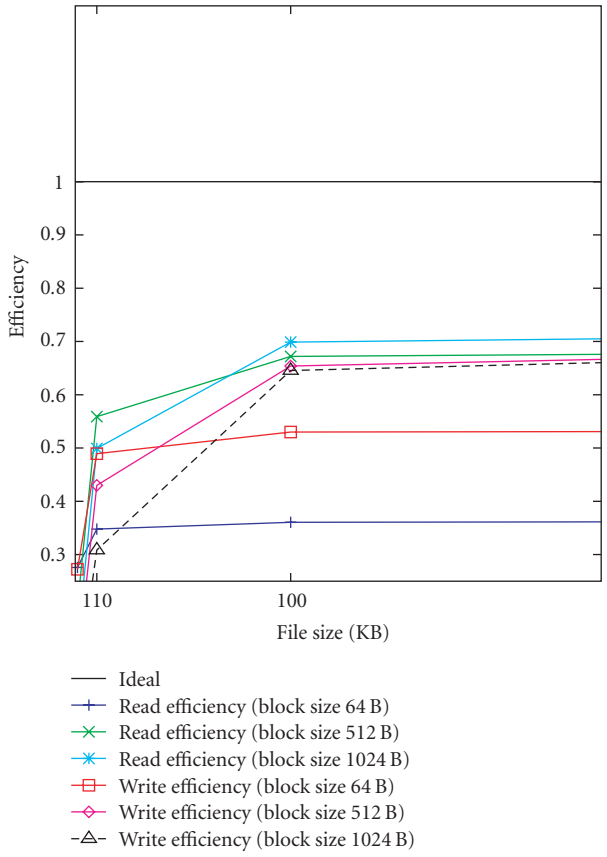


Figure 14: HWFS sequential read/write efficiency with single RAM disk plotted against different file sizes.

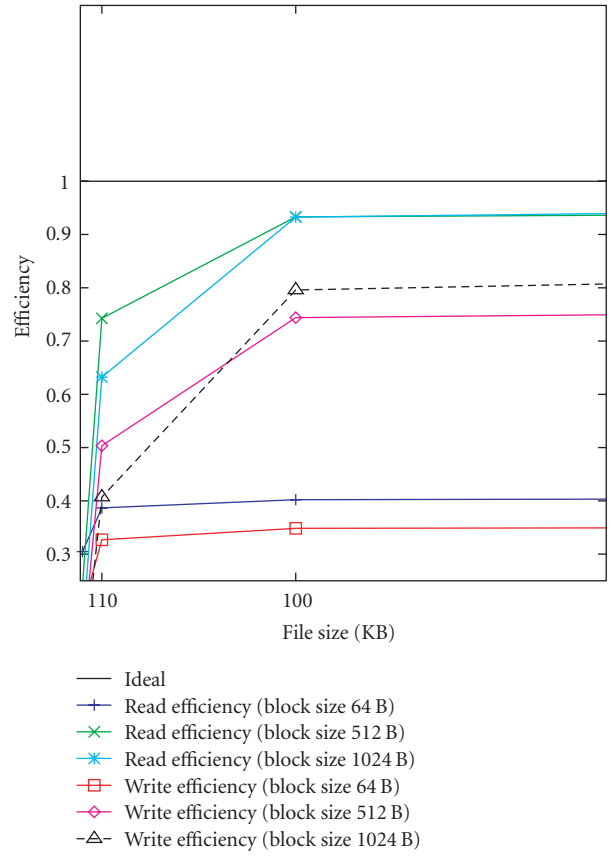


Figure 15: HWFS sequential read/write efficiency with two RAM disks plotted against different file sizes.

Table 5: Hardware filesystem execution time for a 1 KB file with a single ram disk with 64 byte block size.

Operation	Total	HWFS	RAMs
Write	9.29 μs	5.54 μs	3.75 μs
Read	9.16 μs	4.32 μs	4.84 μs
Delete	5.27 μs	2.66 μs	2.61 μs

external memory channels. While it would be possible to further model multiple disks by subdividing up the external memory, two disks provide sufficient results to motivate the research to investigate the HWFS with actual SATA disks.

For 64 byte blocks, the memory channel bandwidth is underutilized. Ideal transactions would be bursts of 128 bytes or larger. It is observed from Figure 15 that the efficiency increases with the size of the block for the same file size. This is due to using larger blocks which improve the data transfer bandwidth. Using large block sizes increases the BRAM usage for the core's metadata buffers without providing any substantial improvement in efficiency. Adding multiple disk support allowed two transactions to be processed in parallel, increasing overall efficiency. Given these trade-offs, 1024 B blocks prove to be ideal for this design.

Table 6: Hardware filesystem read/write execution time with two RAM disk synthesized for the XC4VFX60.

	Read			Write		
File Size (Bytes)	64 B	512 B	1024 B	64 B	512 B	1024 B
1 KB	8.4 μs	10.69 μs	17.05 μs	13.08 μs	21.87 μs	36.86 μs
10 KB	66.17 μs	34.47 μs	40.48 μs	78.33 μs	50.85 μs	62.87 μs
100 KB	636.63 μs	274.35 μs	274.45 μs	734.69 μs	344.09 μs	321.7 μs
1 MB	6.4 ms	2.75 ms	2.69 ms	7.4 ms	3.37 ms	3.02 ms
10 MB	64.3 ms	27.47 ms	26.7 ms	74.7 ms	33.54 ms	29.86 ms
100 MB	643.28 ms	274.68 ms	267.84 ms	746.8 ms	335.3 ms	298.38 ms

5. Conclusion and Future Work

This paper evaluates the feasibility, functionality, and performance of a hardware filesystem implemented on an FPGA device. The HWFS core provides a generic interface to storage media and was evaluated with a RAM Disk. The design was synthesized and run on an ML410 developer board (Xilinx Virtex-4 device). By adding a RAID controller, split transactions to multiple RAM Disks are also supported, yielding additional performance benefits by allowing concurrent requests in parallel to separate disks.

Synthesis results show that the HWFS and RAID cores in total use ≈7% of the slices for an XC4VFX60 device. The design correctly implements the four basic filesystem operations: open, read, write, and delete. The filesystem, which was designed for situations that require relatively few very large files provides efficient run-time performance for file sizes greater than 100 KB as the metadata overhead has little effect on the access times for files larger than that threshold. The sequential read/write efficiencies improve with larger disk block sizes due to higher data transfer rates and smaller overhead.

The novel architecture proposed and implemented in this project has the potential of increasing the disk to core bandwidth by bypassing the sequential software stack of the OS, avoiding the use of main memory bandwidth and reducing the processor's computational load.

Current results are limited to just RAM Disks but once a SATA IP core is acquired, a simple interface can be created to port to support SATA drives. This will allow the hardware filesystem to be evaluated with actual File I/O performance using HPC I/O benchmarks. Thus, this filesystem core is an important first step in testing a parallel hardware filesystem for coordinating file access from multiple, distributed disks.

Acknowledgment

This project was supported in part by the National Science Foundation under NSF Grant CNS 06-52468 (CRI). The opinions expressed are those of the authors and not necessarily those of the foundation.

References

[1] S. Tam and L. Jones, "Embedded serial ATA storage system," Tech. Rep. XAPP716(v1.0), Xilinx, San Jose, Calif, USA, October 2006.

[2] R. Sass, W. V. Kritikos, A. G. Schmidt, et al., "Reconfigurable Computing Cluster (RCC) Project: investigating the feasibility of FPGA-Based petascale computing," in *Proceedings of the International Symposium on Field Programmable Custom Computing Machines (FCCM '07)*, April 2007.

[3] C. Pedraza, E. Castillo, J. Castillo, et al., "Cluster architecture based on low cost reconfigurable hardware," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 595–598, September 2008.

[4] M. Saldana, E. Ramalho, and P. Chow, "A message-passing hardware/software co-simulation environment to aid in reconfigurable computing design using TMD-MPI," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 265–270, December 2008.

[5] H. K.-H. So and R. Brodersen, "File system access from reconfigurable FPGA hardware processes in BORPH," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 567–570, September 2008.

[6] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: observing the effects of kernel operation on parallel application performance," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '07)*, 2007.

[7] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '08)*, 2008.

[8] NCBI user services, "Genbank overview," August 2005, http://www.ncbi.nlm.nih.gov/Genbank.

[9] T. Agerwala, "System trends and their impact on future microprocessor design," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '02)*, IEEE Computer Society Press, Los Alamitos, Calif, USA, November 2002.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1996.

[11] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[12] D. Lavenier, S. Guyetant, S. Derrien, and S. Rubini, "A reconfigurable parallel disk system for filtering genomic banks," in

Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '03), pp. 154–163, 2003.

[13] Netezza, "Netezza data warehouse appliances," http://www.netezza.com/data-warehouse-appliance-products/dw-appliance.aspx.

[14] Bluearc, The bluearc file system technology, http://www.bluearc.com/html/products/file_system.shtml.

[15] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: high-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, 1994.

[16] A. A. Mendon and R. Sass, "A hardware filesystem implementation for high-speed secondary storage," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 283–288, December 2008.

[17] Xilinx Inc., "Xilinx ml410 board," http://www.xilinx.com/ml410-p.

[18] ASICS World Services, "Serial ATA Host IP Core," December 2007, http://www.asics.ws/.

[19] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 109–116, ACM, 1988.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 501672, 8 pages
doi:10.1155/2009/501672

Research Article

Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings

Knut Wold and Chik How Tan

NISlab, Department of Computer Science and Media Technology, Gjøvik University College, 2802 Gjøvik, Norway

Correspondence should be addressed to Knut Wold, knutw@hig.no

Received 26 February 2009; Accepted 19 June 2009

Recommended by Lionel Torres

A true random number generator (TRNG) is an important component in cryptographic systems. Designing a fast and secure TRNG in an FPGA is a challenging task. In this paper, we analyze the TRNG designed by Sunar et al. (2007) based on XOR of the outputs of several oscillator rings. We propose an enhanced TRNG with better randomness characteristics that does not require postprocessing and passes the statistical tests. We have shown by experiment that the frequencies of the equal length oscillator rings in the TRNG are not identical. The difference is due to the placement of the inverters in the FPGA and the resulting routing between the inverters. We have implemented our proposed TRNG in an Altera Cyclone II FPGA. Our implementation has passed the NIST and DIEHARD statistical tests with a throughput of 100 Mbps and with a usage of less than 100 logic elements in the FPGA. The restart experiments have shown that the output from our TRNG behaves truly random and not pseudorandom.

1. Introduction

Traditionally, a high assurance implementation of cryptographic algorithms has been done in application specific integrated circuit (ASIC). During the recent years, more and more of these implementations are done in field programmable gate array (FPGA). There are several reasons for this development. The FPGA can be reprogrammed, leading to more flexibility for modification of algorithms, changing algorithms, and fixing bugs. The development of an algorithm in an FPGA is easier and faster as compared to an ASIC design, resulting in a shorter time-to-market. In addition, the latest FPGA devices are manufactured with the state-of-the-art technology.

It is well known that a true random number generator (TRNG) is an important component of today's cryptographic systems. Typically a TRNG can be used for generating keys, initialization vectors, random sequences for cryptographic challenges-responses, and so forth. In a cryptographic system, a private or secret parameter is normally generated by a TRNG and is an interesting property to an attacker. Therefore, the generation of a random bit sequence is important and should be unpredictable to an attacker. One common method for generating a truly random sequence is to amplify the thermal noise in a diode [1]. The disadvantage of this method is the use of external components. This approach enables an attacker to manipulate and read the random bit sequence from the device and consequently violate the security of the entire cryptographic system. If the TRNG is implemented entirely inside the FPGA, an attacker will have difficulties in retrieving and manipulating the random bit sequence. The challenge is to design a TRNG in an FPGA passing all statistical tests and at the same time using as few resources as possible and achieving a high throughput of random bits.

In this paper, we examine more closely the TRNG based on oscillator rings proposed by Sunar et al. [2]. We show that the TRNG described in [2] is not random without postprocessing. We propose an enhancement of the proposal from [2] and experimentally show improved performance with respect to FPGA resource usage and throughput. We also show that our TRNG has no bias and, therefore, no need for complicated postprocessing. We experimentally demonstrate that the frequencies of the oscillator rings are different due to the placement and routing of the inverters inside the FPGA.

We have implemented our proposal in an Altera Cyclone II FPGA [3]. Our implementation of the TRNG based on oscillator rings passes the NIST and DIEHARD statistical tests with a throughput of 100 Mbps and the usage of less than 100 logic elements in the FPGA. Repeated restarts of the TRNG from the same reset state have shown that the output of our random generator behaves truly random and not pseudorandom. The standard deviation has been calculated from 1000 traces recorded after reset. A short startup period should be omitted in order to obtain good quality of the randomness, but after the TRNG output stabilizes, the standard deviation becomes constant and in accordance with the theoretical values.

The rest of this paper is organized as follows: in Section 2, we briefly examine the previous work on TRNG in FPGA. In Section 3, we analyse the TRNG of [2]. In Section 4, we propose an enhancement of the TRNG to achieve better randomness on the output sequence. The analysis of the randomness of our proposed TRNG and the investigation of distribution of frequencies on oscillator rings are discussed in Sections 5 and 6, respectively. In Section 7, we describe in detail an implementation of our proposed TRNG. In Section 8, we investigate the behavior of our TRNG after repeated restarts from known reset state, and finally we make a conclusion in Section 9.

2. Related Work

Several implementations of TRNG in FPGA have been proposed during the recent years. The common entropy source used is jitter on clock signals. Jitter can be viewed as timing deviation from the theoretically correct position due to electronic or thermal noise [4]. The random jitter will typically follow a Gaussian distribution characterized by a certain standard deviation (σ). Usually, jitter is an unwanted property in a system, but this behavior is useful when generating random signals in a TRNG.

In 2002, Fischer et al. [5] used the jitter in analogue phase-locked loop (PLL) in FPGAs from Altera as entropy source in a TRNG. The strategy was to create different clock signals with jitter from the PLL and sample one of the clock signals with the other. This method is restricted to FPGAs containing such analogue components. Later, Kohlbrenner and Gaj [6] used a similar technique, but the clocks are generated by oscillator rings containing two transparent latches, a buffer and an inverter. Since the frequencies of the two oscillator rings have to be almost equal, the oscillator rings have to be correctly matched. Tkacik [7] proposed a TRNG using a linear feedback shift register (LFSR) and a cellular automaton shift register (CASR) clocked by two independent oscillator rings. Selected outputs from the LFSR and CASR are combined by an XOR generating the final random signal. The disadvantage of this scheme is that the TRNG has memory and is, therefore, not stateless as pointed out in [8]. In 2006, Golić [9] proposed a TRNG using a Galois ring oscillator (GARO) and a Fibonacci ring oscillator (FIRO). These LFSR-like structures use inverters as delay elements instead of register elements. The outputs from one GARO and one FIRO are combined by means of an XOR and the random sequence is generated by sampling with a D flip-flop. This design was further investigated by Dichtl and Golić [10]. The output signal from these FIRO/GARO structures has a noisy analogue behavior, making them more susceptible to cross-talk from other signals inside the FPGA than ordinary digital signals. In 2007, Sunar et al. [2] gave a theoretical proposal of a TRNG based on several equal length oscillator rings made up of an odd number of inverters (see Figure 1). The outputs from the oscillator rings are XORed together and sampled with a D flip-flop. To compensate for the imbalance between the number of zeros and ones in the random signal, a postprocessing stage is present on the output of the D flip-flop. Schellekens et al. [11] implemented this scheme in a Xilinx FPGA, but with a large number of rings in order to make the random sequence output pass the statistical tests. In 2008, Vasyltsov et al. [12] proposed a TRNG based on a 5-stage metastable ring oscillator, where each stage contains only one inverter. The result is a fast and small implementation of a TRNG in an FPGA or ASIC, but optimization in the synthesis process causes difficulties in the FPGA implementation. Recently, Danger et al. [13] proposed a fast TRNG based on creating metastability in open loop structures in FPGAs.

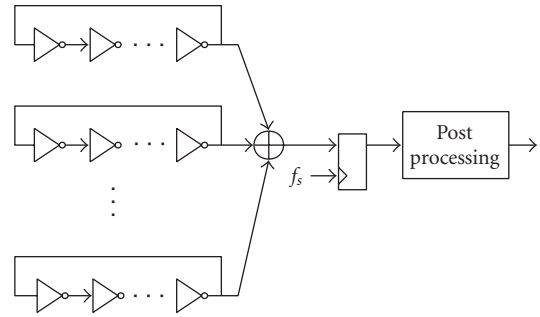


Figure 1: TRNG based on oscillator rings [2].

3. TRNG Based on Oscillator Rings

Since our proposed enhancement is based on the TRNG of Sunar et al. [2], we take a closer look on the design from [2], see Figure 1. The TRNG consists of several equal length oscillator rings connected to an XOR tree. The output from the XOR tree is sampled by a D flip-flop, and the output signal of the D flip-flop is then postprocessed in order to increase the entropy and remove bias from the random signal. The proposed postprocessing in [2] is a resilient function implemented as a BCH-code. The suggested design of the TRNG consists of 114 oscillator rings where each ring consists of 13 inverters. The suggested sampling frequency is 40 MHz and the postprocessing is a [256, 16, 113] extended BCH code. The resulting throughput of the TRNG in [2] is 2.5 Mbps.

The entropy source of the TRNG is the jitter created by each oscillator ring. The jitter has a Gaussian distribution around each clock transition between logic low and logic high level. This jitter will create an accumulated phase drift in each ring so that the transitions will be at different times

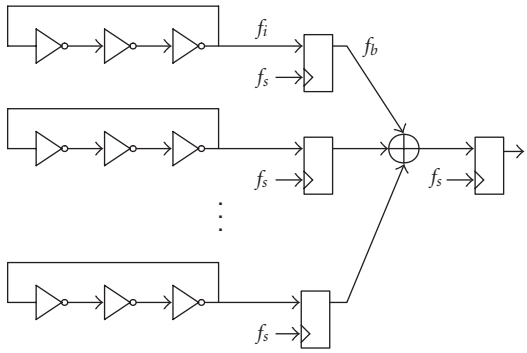


Figure 2: Our proposal.

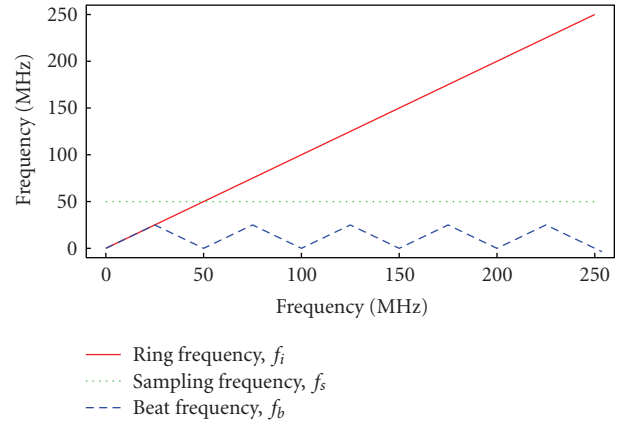


Figure 3: Beat frequency.

in the sampling period. Due to the jitter, the unpredictable transition region is assumed to be uniformly distributed in the sampling period. The number of rings needed can then be calculated based on the coupon collector's problem, that is, the number of uniform random selections of N urns such that all urns are selected at least once. The number of urns is determined by the proportion of the jitter size compared to the frequency of the oscillator ring. Because the number of rings grows exponentially when filling up the last urns, a lower fill rate than 100% is selected. To compensate for this, a BCH-code is used for postprocessing. The resulting random number throughput is reduced by a factor of 16 due to this postprocessing scheme.

In [10], some weaknesses of this implementation were mentioned. The main concern of the authors of [10] is that the XOR-tree and the sampling D flip-flop cannot handle the high number of transitions from the oscillator rings. The frequency of an oscillator ring is approximately the same or higher than the sampling frequency. With many oscillator rings in parallel, the number of transitions during a sampling period will be so high that the setup- and hold-times for the lookup table (LUT) and the internal register element in the FPGA will be shorter than specified for the device.

The analysis of the TRNG from [2] is shown in Sections 5 and 6 as it is better to present a comparison with the proposed enhanced TRNG.

4. Our Proposed Enhancement

To cope with the problem with many transitions in the sampling period, we suggest an enhancement of the TRNG based on the oscillator rings in [2] by adding an extra D flip-flop after each ring (Figure 2). As we will show, this configuration will improve the randomness of the TRNG. The randomness of the configuration relies on the jitter variations of the oscillator rings. Adding these extra flip-flops will not alter the collection of the randomness of each ring, but improve the overall randomness at the output.

The frequency of the oscillator ring (f_i) is dependent on the odd number of inverters in the ring. The frequency will increase with the decreasing number of inverters. In order to have a fast and small TRNG, the number of inverters should be as low as possible making the frequency of the rings become high as compared to the sampling frequency (f_s). The advantage of our enhancement is that the signals on the input of the XOR will now be synchronous with the sampling clock and only updated once in the sampling period. Due to this reduction in transitions on the input to the XOR tree, the setup- and hold-times for the internal logic in the FPGA will now be within acceptable limits.

The frequency of the beat signal (f_b) after the extra flip-flop will always be less than half of the sampling frequency and lie in the interval $[0, f_s/2]$ (Figure 3). The sampling frequency f_s should be chosen such that the beat frequency f_b at the input of the XOR is as high as possible, and avoid the frequency of the oscillator rings be a multiple of the sampling frequency resulting in a beat frequency near zero or no transitions (in the worts case) in the beat signal.

The result of adding the extra D flip-flop, is that the switching activity on the input to the XOR-tree is significantly reduced. The XOR calculation becomes deterministic while the randomness is collected by the sampling of the free running oscillator rings. The sampling of a free running oscillator ring could lead to metastability in the flip-flop causing the output of the flip-flop neither to be logic low or logic high for a short time period. This phenomenon can arise when a transition occurs during the setup and hold-time of the flip-flop, which is the case for the extra D flip-flops in our proposed TRNG. To avoid the metastable state to propagate into the XOR tree, the output of the oscillator ring could be sampled by one additional D flip-flop.

If a large number of rings is needed, the logic of the XOR tree will be deep and contain many logic levels. The result could be violating the timing inside the FPGA because the time delay through the XOR tree is longer than the sampling period. In this case, one or more register levels can be inserted into the XOR tree. This will not affect the throughput, but it will increase the latency of the TRNG output and increase the resources used in the FPGA.

5. Bias in TRNG

One of the basic statistical tests of random number generators is the frequency test of ones and zeros. For a good random bit sequence the probability of a zero or a one should

be equal to 1/2. In other words, there should be no bias in the random bit sequence.

Let X and Y be two random bit sources with expected values $E(X) = E(Y) = \mu$, respectively, and let ρ be their correlation. Then the expected value of the XOR of the two sequences $(X \oplus Y)$ is given by (see e.g., [14]).

$$E(X \oplus Y) = \frac{1}{2} - 2\left(\mu - \frac{1}{2}\right)^2 - 2\rho\mu(1-\mu). \quad (1)$$

If μ is close to 1/2, (1) can be written as

$$E(X \oplus Y) \approx \frac{1}{2}(1-\rho). \quad (2)$$

It can be seen that correlation between the two sequences will generate bias in the output from the XOR of two random bit sequences. If X and Y are linearly independent, then $\rho = 0$ and $E(X \oplus Y) \approx 1/2$.

If there are n independent bits, each with expected value μ, then the expected value of XOR of all these bits will be given by

$$\frac{1}{2} + (-2)^{n-1}\left(\mu - \frac{1}{2}\right)^n = \frac{1}{2}(1 + (-2\varepsilon)^n), \quad (3)$$

where $\varepsilon = \mu - 1/2$. Since $\mu \in (0,1) \Rightarrow |2\varepsilon| < 1$, the expected value in (3) will converge to 1/2 for increasing number of sequences, n. In other words, adding more oscillator rings in the TRNG design should improve the bias if the rings are independent.

We have carried out some experiments on the randomness of the TRNG in [2] (without any postprocessing) and our proposal in Figure 2. The experiments are carried out on a Starter Development Board from Altera containing a Cyclone II FPGA. This device has a core voltage of 1.2 V and is fabricated in 90 nm technology. Quartus II WebEdition 6.1 is used for synthesis and Place and Route (P&R). The sequences of random bits generated inside the FPGA are stored in an external SRAM and transmitted to a PC for analysis through an asynchronous serial connection. The result is a number of blocks of subsequent random number bits from the TRNG where each block has a maximum size of 4 Mbit. The sampling frequency used in this experiment is 50 MHz. No constrains have been put on the P&R tool regarding the placement of the inverters in the FPGA.

We have implemented the two configurations of TRNG (Figures 1 and 2), recorded 10 blocks of 1 Mbit of random data from each configuration, and determined the frequency of ones in all blocks. We have performed the experiment with oscillator rings of lengths 3 and 13, and with varying number of rings. The results are shown in Figure 4. They indicate that the design in [2] has a bias after the XOR of the oscillator rings. The tendency is that the bias increases with the increasing number of rings and there is a majority of zeros in the output. Comparing these observations against (1)–(3) shows that there is some dependency or correlation in the random sequences creating a bias. It seems that this bias is due to the problem with the high number of transitions at the input of the XOR tree and the sampling flip-flop. For our configuration (Figure 2), it is seen that

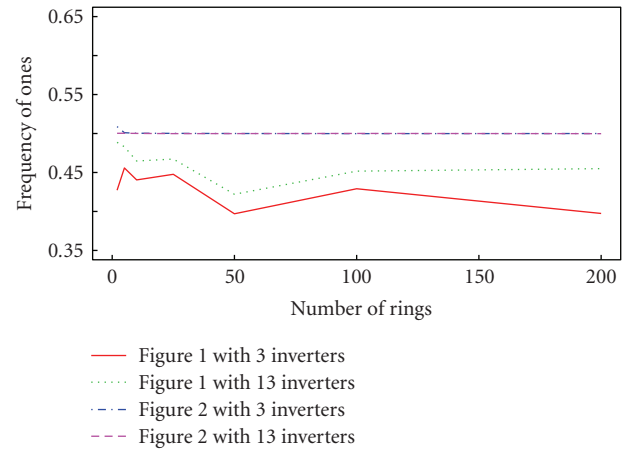


Figure 4: Bias of the TRNG on Figures 1 and 2.

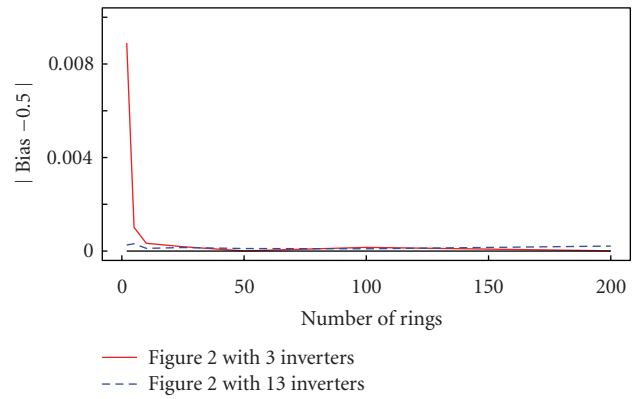


Figure 5: |Bias − 1/2| of the TRNG on Figure 2.

the expected value is close to 1/2 for increasing number of oscillator rings. Figure 5 shows a closer view of the curves, where the absolute value of the bias from the ideal 0.5 level is shown for our configuration with 3 and 13 inverters in the oscillator rings. It is seen that the bias converges to 0, and that our enhanced TRNG behaves according to the theory of XOR of independent random sequences.

6. Distribution of Ring Frequencies

According to [2], the assumption of randomness is that the equal length oscillator rings will have the same frequency while the phase drift related to the jitter causes the drifting of the transition regions. We believe that the frequencies of the oscillator rings will be different from each other. We have carried out an experiment where we have implemented 64 oscillator rings in the Altera Cyclone II FPGA and tapped out the signal from each of these rings to I/O-pins on the FPGA. These frequencies were measured with an oscilloscope.

Figure 6 shows the histograms of the frequencies of oscillator rings with 5 and 31 inverters, respectively. (For oscillator rings with 3 inverters, the measured signals are outside the specification of the I/O-pins for our Cyclone II FPGA (maximum frequency of 300 MHz). However, the

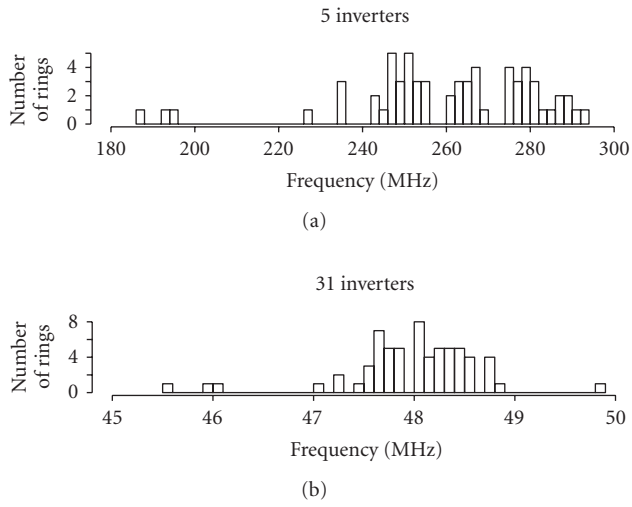


Figure 6: Histogram of ring frequencies.

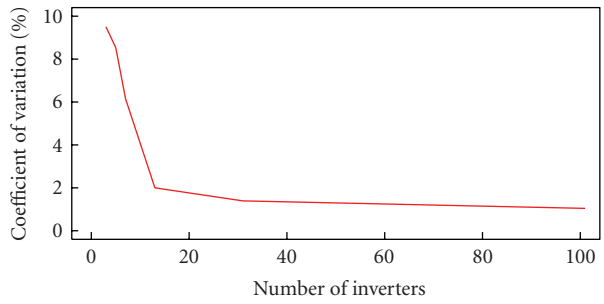


Figure 7: Dispersion of frequencies.

frequencies are measurable and the measurements with 3 inverters gave a similar histogram as shown in Figure 6 with 5 inverters.) From this experiment, it is observed that the distribution of the frequencies for short rings does not follow a Gaussian distribution and the frequencies are clustered in groups. For longer rings, the clustering is not so obvious and the distribution is approaching Gaussian with only some values far from the mean.

When examining similar histograms for other lengths, it is observed that the dispersion is decreasing with increasing number of inverters. In Figure 7, the dispersion is measured by the coefficient of variation defined as the percentage of σ/μ, where σ is the standard deviation and μ is the mean of the measured frequencies. It can be seen that the dispersion is high for short rings and decreasing with longer oscillator rings. Based on this observation, using oscillator rings with only 3 inverters will give the highest dispersion in the frequencies.

To explain the behavior of the frequency distribution, the architecture of the Altera Cyclone II FPGA [3] has to be examined. This FPGA consists of a matrix with logic elements (LEs), each containing a programmable register and an LUT for implementing any logic function of four inputs. 16 of these LEs are then grouped into a logic array block (LAB). All the LEs and LABs are connected together via different routing resources depending on the distance between them inside the FPGA. When running P&R for the design in an FPGA, the inverters in the oscillator rings are located at physical LEs. Depending on the placement, the routing delay between the LEs will differ. If all the inverters are placed in LEs inside one LAB, the routing delay will be short. If the inverters are placed in LEs in different LABs, the routing delay will be increased resulting in a lower frequency of the oscillator ring. In addition, there will also be a variation in the delay of each LUT, typically following a Gaussian distribution. All these variations in the routing delays cause the distribution of the oscillator ring frequencies and the clustering for short rings. For short oscillator rings, the inverters of some of the rings are placed in the same LAB, but for some of the other rings, the inverters are placed in two or more LABs, resulting in routing delays with large variations. For long oscillator rings, the difference between the routing delays of each ring will be smaller due to the fact that the inverters have to be placed in more than one LAB. From Figure 7, this can be seen indirectly. For small number of inverters, the dispersion is high, and decreasing until the rings contains more than 16 inverters and therefore filling up more than one LAB. For more than 16 inverters, the dispersion is constant, indicating that the variation is only due to the natural timing variation between the difference logic elements in the FPGA. For other FPGAs with similar architecture, the oscillator ring frequencies will result in similar distributions.

Due to the observed distribution of frequencies of equal length oscillator rings, the transition regions will quickly be spread out over the sampling time period, much faster than if only the accumulation of the oscillator ring jitter was contributing. In order to examine the effect of this frequency distribution on the randomness, we carried out an experiment where a model of the TRNG was made in MATLAB without involving the jitter in the generation of the output sequence. 100 blocks of 1 Mbit each were recorded, and for each block a new set of oscillator ring frequencies was generated from a Gaussian distribution (same as generating one block of data from 100 different TRNGs). The resulting output sequence was tested by the NIST randomness test suite [15], and it showed that with 50 or more oscillator rings in the MATLAB TRNG model, the quality of the generated random sequences was good enough to pass the NIST test suite. This experiment shows that a TRNG combining several equal length oscillator rings outputs where there is a dispersion between the frequencies, generates bit sequences that have good qualities even though they are deterministic. The jitter introduced in the oscillator rings contributes with the unpredictable behavior that is necessary to have a true random source.

7. TRNG Implementation

We have implemented our proposed TRNG from Figure 2. In order to have a fast and small TRNG, the number of inverters in the oscillator ring is selected to be 3. A sampling frequency of 100 MHz is selected, resulting in a throughput of 100 Mbps since our TRNG does not use any postprocessing. In most of real TRNG designs in cryptographic systems,

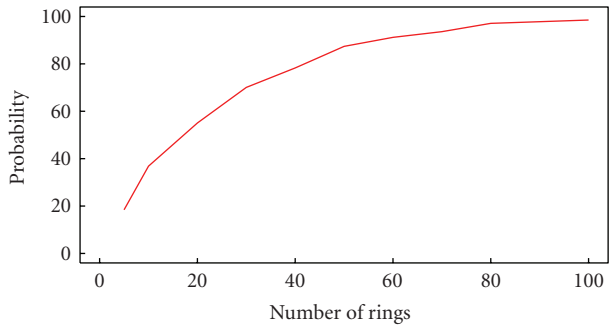


Figure 8: Simulated probability of hitting a transition region.

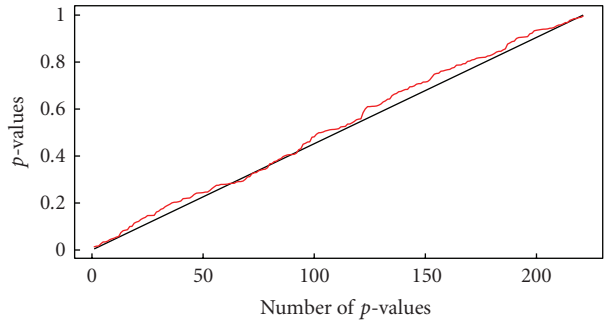


Figure 10: Results from DIEHARD.

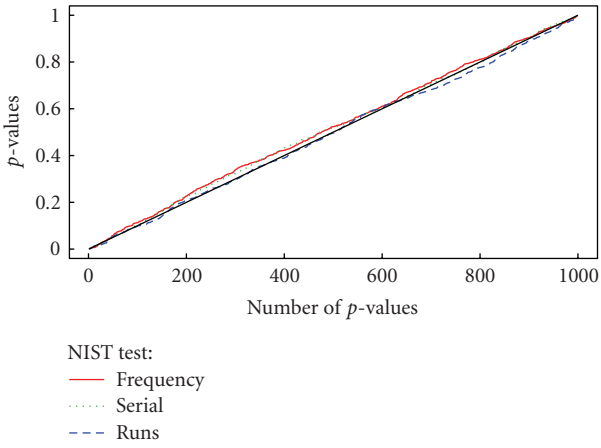


Figure 9: Selected results from the NIST suite.

Table 1: Resources used in the Altera FPGA.

Oscillator rings	LUT only LEs	LUT/Register LEs	Total LEs
25	57	26	83
50	116	51	167

using postprocessing is recommended in order to improve the randomness by increasing the entropy and removing bias. But, for our TRNG, a postprocessing is not needed to pass the statistical tests, and, therefore, an additional post-processor can be of a simple type like an XOR of two subsequent bits or a von Neumann corrector.

The required number of rings is estimated based on the probability to hit the transition region with the sampling. Sunar et al. [2] computed this by using a combinatorial approach (coupon collector's problem). An alternative way is to make a statistical model of the TRNG and perform simulations in order to decide how many oscillator rings are needed to achieve a high probability such that at least one ring is sampled in the transition region. When the size of the jitter is small compared to the sampling period, the simulations show that the number of rings in the transition region follows a Poisson distribution with a parameter $\lambda = k \cdot r$ where r is the number of rings and k is a constant depending on the size of the jitter compared to the sampling period. The probability of sampling in at least one of the transition regions versus the number of rings is shown in Figure 8. It shows that the probability increases rapidly for small number of rings, but many oscillator rings are needed to get a 100% certainty.

We have carried out an experiment where we have used 50 oscillator rings with 3 inverters, a sampling frequency of 100 MHz and no postprocessing. A total of 1000 blocks of 1 Mbit (a total of 1 Gbit) of random data have been captured from the TRNG. The data was tested by using the statistical tests of NIST (SP 800-22) [15] and DIEHARD [16]. The random data passed both tests. We also performed the same experiment with only 25 oscillator rings. The random data also passed both the NIST and the DIEHARD tests (Figures 9 and 10). From these figures it can be observed that the sorted P-values from the tests follow the ideal diagonal line. These experiments indicate that it is probably not necessary to have almost 100% certainty to hit at least one transition region in order to pass the NIST and DIEHARD statistical tests for this kind of TRNG.

Table 1 shows the amount of resources used for our TRNG in the Altera Cyclone II FPGA. For 25 oscillator rings, the number of LEs is less than 100 (<1% of the total number of LEs in our medium size FPGA). For comparison, the original design in [2] occupies more than 1800 LEs.

A TRNG design based on several oscillator rings is robust because the placement of the inverters inside the FPGA is not critical and no constraints on the P&R tool are necessary. As the experiments show, having different delays of the inverter chains contributes to the quality of randomness. However, if there are interactions between the oscillator rings making the rings oscillate with the same frequency and phase, the output bit sequence will naturally not be random. We have not seen any sign of interaction between the oscillator rings in our experiments.

All the tests were performed at the room temperature. The effect of varying the temperature is beyond the scope of this study, but in general, changing the temperature will influence the oscillator ring frequencies. An increase in temperature will decrease the oscillator ring frequencies and vice versa. But since all the rings will be influenced in the same manner, there will be a shift in all the frequencies and the dispersion between the frequencies will remain approximately unchanged.

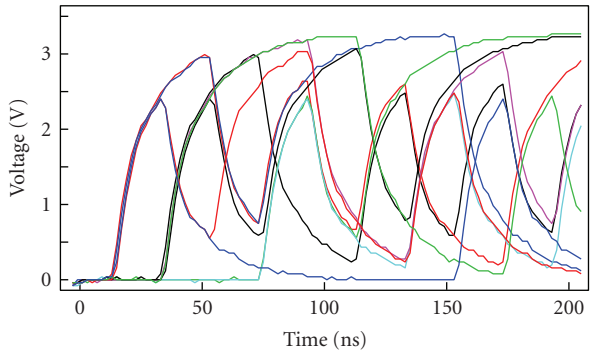


Figure 11: 10 restarts with 25 rings.

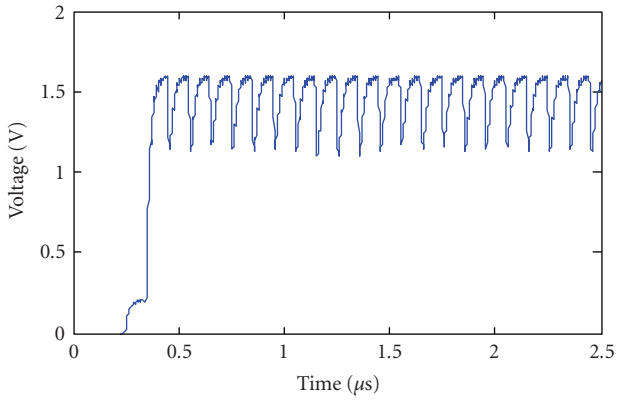


Figure 12: Standard deviation of 1000 traces, sampling frequency 10 MHz and 25 rings.

8. Restart Experiment

In order to examine the randomness of our TRNG after startup, an oscilloscope was used to capture the random output when restarting the TRNG several times from the same reset state. While the reset is active, the oscillator ring outputs are kept at zero or low level. When the reset is deactivated, the oscillator rings start to oscillate. In Figures 11 and 10 restart sequences from the output of the TRNG are captured where the oscilloscope is triggered on a clocked version of the reset signal at the origin of the graph. The sampling frequency is 50 MHz. Because of the bandwidth limitation in the oscilloscope, the measured outputs are not square signals. It can be seen that all the outputs start at zero, but there is a deviation after the first clock period of 20 ns. This experiment shows that our TRNG outputs randomness quickly after a restart. The experiment also shows that since the traces are deviating from each other, the output contains true randomness and not pseudorandomness. If the random signal had been only pseudorandom, the restart experiment should have given equal traces when repeatedly starting the TRNG from the same reset state.

The restart experiment was expanded to capture 1000 restarts. The standard deviation for all these traces was calculated and is shown in Figure 12 for a sampling frequency of 10 MHz. The form of the curve is very regular because all the traces are aligned with the sampling frequency, and

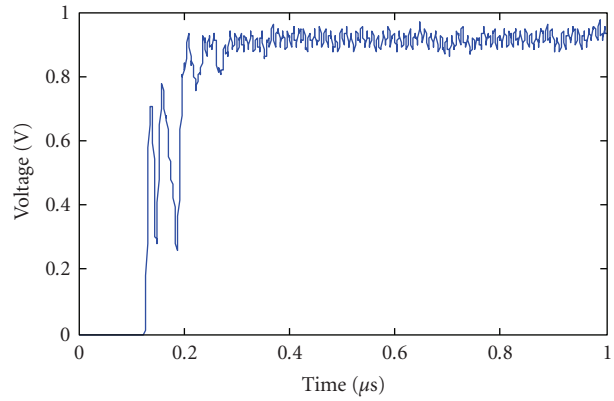


Figure 13: Standard deviation of 1000 traces, sampling frequency 100 MHz and 25 rings.

because the random signal is digital with voltage level of either +3.3 V or 0 V. Theoretically, the standard deviation can be calculated by looking at the probability of a voltage level of logic zero and logic one, and the probability of a transition between the two logic levels. A good random signal should have equal probability of zeros and ones, and also equal probability of a transition or no transition. The mean value of the voltage with these conditions, is then $\mu = 1.65$ V. The standard deviation can then be calculated for the two cases: (1) the signal is in the middle of the sampling period, and (2) the signal is at the sampling point. For case (1), the standard deviation is $\sigma = 1.65$ V, and for case (2) the standard deviation is $\sigma = 1.17$ V. From Figure 12 we can see that the theoretical values match the measured data when the starting period is omitted. It is also observed that the standard deviation is stable after a short startup period.

Figure 13 shows the standard deviation with a sampling frequency of 100 MHz. Due to the band limitation of the oscilloscope, the values of the standard deviation differ from the theoretical values. It is observed that in a short startup period, the quality of the randomness is not optimal because the standard deviation is not stabilized. From Figures 12 and 13, it is seen that this startup period is constant regarding the sampling frequency or the throughput bit-rate. For the case of a TRNG with 25 oscillator rings with 3 inverters in each ring, this startup time is about 300ns. This indicates that the first data bits should be omitted in order to have good quality of the random sequence.

9. Conclusion

We have analyzed the TRNG in [2] and have proposed an enhancement of a TRNG based on oscillator rings. By adding an extra flip-flop after each inverter ring before the XOR tree, we have shown that the performance is much better than [2] regarding the random signal. We have also shown that the frequencies of each ring are not equal but have some kind of distribution. Smaller rings will have higher dispersion in the distribution and therefore also better potential for fast generation of randomness after restart.

We have implemented the TRNG from Figure 2 and carried out statistical tests on the resulting random bit sequences. We have shown that our TRNG passes both the NIST and DIEHARD tests without postprocessing. The throughput of the TRNG is 100 Mbps and the resources used in the FPGA are less than 100 logic elements in an Altera Cyclone II FPGA.

The restart experiments show that the output of the TRNG behaves truly random and not pseudorandom since the traces differ when restarted from the same reset state. These experiments also show that the standard deviation of the traces is in accordance with the theory, and that it is stable after a short startup period. Due to this startup period, the first bits should be omitted in order to have good quality of the randomness.

Acknowledgment

This paper is an extended version of the conference paper [17] presented at ReConFig08. This work was carried out while Chik How Tan was at Gjøvik University College.

References

[1] B. Jun and P. Kocher, "The Intel Random Number Generator," White paper prepared for Intel Corporation, April 1999.

[2] 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*, vol. 56, no. 1, pp. 109–119, 2007.

[3] Altera Corporation, "Cyclone II Device Handbook," June 2006, http://www.altera.com.

[4] Tektronix Inc, "A Guide to Understanding and Characterizing Timing jitter," 2003, http://www.tektronix.com/jitter.

[5] V. Fischer and M. Drutarovský, "True random number generator embedded in reconfigurable hardware," in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '03)*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 415–430, Springer, 2003.

[6] P. Kohlbrenner and K. Gaj, "An embedded true random number generator for FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '04)*, pp. 71–78, ACM, 2004.

[7] T. E. Tkacik, "A hardware random number generator," in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '03)*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 450–453, Springer, 2003.

[8] M. Bucci and R. Luzzi, "Design of testable random bit generators," in *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '05)*, vol. 3659 of *Lecture Notes in Computer Science*, pp. 147–156, Springer, 2005.

[9] J. D. Golić, "New methods for digital generation and postprocessing of random data," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1217–1229, 2006.

[10] 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)*, vol. 4727 of *Lecture Notes in Computer Science*, pp. 45–62, Springer, 2007.

[11] D. Schellekens, B. Preneel, and I. Verbauwhede, "FPGA vendor agnostic true random number generator," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, IEEE, 2006.

[12] 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)*, vol. 5154 of *Lecture Notes in Computer Science*, pp. 164–180, Springer, 2008.

[13] J.-L. Danger, S. Guilley, and P. Hoogvorts, "High speed true random number generator based on loop structures in FPGAs," *Microelectronics Journal*. In press.

[14] R. Davies, "Exclusive Or (XOR) and Hardware Random Number Generators," February 2002, http://www.robertnz.net.

[15] NIST Special Publication 800-22, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," October 2000.

[16] G. Marsaglia, "DIEHARD: A Battery of Tests of Randomness," 1996, http://stat.fsu.edu/pub/diehard.

[17] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in FPGA based on oscillator rings," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 385–390, 2008.