Journal of Electrical and Computer Engineering

ESL Design Methodology

Guest Editors: Deming Chen, Kiyoung Choi, Philippe Coussy, Yuan Xie, and Zhiru Zhang

ESL Design Methodology

Journal of Electrical and Computer Engineering

ESL Design Methodology

Guest Editors: Deming Chen, Kiyoung Choi, Philippe Coussy, Yuan Xie, and Zhiru Zhang

Editorial Board

The editorial board of the journal is organized into sections that correspond to the subject areas covered by the journal.

Circuits and Systems

Communications

Signal Processing

Contents

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 358281, 2 pages
doi:10.1155/2012/358281

Editorial
ESL Design Methodology

Deming Chen,[1] Kiyoung Choi,[2] Philippe Coussy,[3] Yuan Xie,[4] and Zhiru Zhang[5]

[1] *Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*
[2] *School of Electrical Engineering, Seoul National University, Seoul 151-742, Republic of Korea*
[3] *Department of Sciences and Techniques, Lab-STICC, Université de Bretagne-Sud, Lorient 56321, Cedex, France*
[4] *Department of Computer Science and Engineering, Pennsylvania State University at University Park, University Park, PA 16802-1294, USA*
[5] *High-level Synthesis Department, Xilinx Inc., San Jose, CA 95124, USA*

Correspondence should be addressed to Deming Chen, dchen@illinois.edu

Received 15 May 2012; Accepted 15 May 2012

1. Introduction

ESL (*electronic system level*) design is an emerging design methodology that allows designers to work at higher levels of abstraction than typically supported by register transfer level (RTL) descriptions. Its growth has been driven by the continuing complexity of IC design, which has made RTL implementation less efficient.

ESL methodologies hold the promise of dramatically improving design productivity by accepting designs written in high-level languages such as C, System C, C++, and MATLAB, and so forth, and implementing the function straight into hardware. Designers can also leverage ESL to optimize performance and power by converting compute intensive functions into customized cores in System-on-Chip (SoC) designs or FPGAs. It can also support early embedded-software development, architectural modeling, and functional verification.

ESL has been predicted to grow in both user base and revenue steadily in the coming decade. Meanwhile, the design challenges in ESL remain. Some important research challenges include effective hardware/software partitioning and co-design, high-quality high-level synthesis, seamless system IP integration, accurate and fast performance/power modeling, and efficient debugging and verification, and so forth.

With the invitation of *Journal of Electrical and Computer Engineering* of the Hindawi Publishing Corporation, we started the effort of putting together a special issue on ESL design methodology. After call for papers, we received submissions from around the globe, and after a careful review and selection procedure, eight papers are accepted into this special issue. These papers cover a wide range of important topics for ESL with rich content and compelling experimental results. We introduce the summaries of these papers next. They are categorized into four sections: high-level synthesis, modeling, processor synthesis and hardware/software co-design, and design for error resilience.

2. High-Level Synthesis

In the paper "*Parametric yield-driven resource binding in-high-level synthesis with multi-Vth/Vdd library and device sizing*" Y. Chen et al. demonstrated that the increasing impact of process variability on circuit performance and power requires the employment of statistical approaches in analyses and optimizations at all levels of design abstractions. This paper presents a variation-aware high-level synthesis method that integrates resource sharing with Vth/Vdd selection and device sizing to effectively reduce the power consumption under given timing yield constraint. Experimental results demonstrate significant power yield improvement over conventional worst-case deterministic techniques.

D. Menard et al. present in the paper "*High-level synthesis under fixed-point accuracy constraint*" a new method to integrate high level synthesis (HLS) and word-length optimisation (WLO). The proposed WLO approach is based on analytical fixed-point analysis to reduce the implementation cost of signal processing applications. Authors demonstrate that area savings can be obtained by iteratively performing WLO and HLS, in the case of a latency constrained application, by taking advantage of the interactions between these two processes.

In "*Highlevel synthesis: productivity, performance, and software constraints*" by Y. Liang et al., a study of HLS targeting FPGA in terms of performance, usability, and productivity was presented. For the study, the authors use an HLS tool called AutoPilot and a set of popular-embedded benchmark kernels. To evaluate the suitability of HLS on real-world applications, they also perform a case study using various stereo matching algorithms used in computer vision research. Through the study, they provide insights on current limitations of mapping general purpose software to hardware using HLS and some future directions for HLS tool development. They also provide several guidelines for hardware friendly software design.

3. Modeling

In "*Task-level data model for hardware synthesis based on concurrent collections*" by J. Cong et al., a task-level data model (TLDM), which can be used for task-level optimization in hardware synthesis for data processing applications, was proposed. The model is based on the Concurrent Collection model that can provide flexibility in task rescheduling. Polyhedral models are embedded in TLDM for concise expression of task instances, array accesses, and dependencies. The authors demonstrate examples to show the benefits of the proposed TLDM specification in modeling task level concurrency for hardware synthesis in heterogeneous platforms.

A. Barreteau et al. in the paper "*A state-based modeling approach for efficient performance evaluation of embedded system architectures at transaction level*" address the important topic of performance evaluation for SoC based on transaction-level modeling (TLM). The authors propose a generic execution model and a specific computation method to support hardware/software architecture evaluation. The benefits of the proposed approach are highlighted through two case studies.

4. Processor Synthesis and Hardware/Software Codesign

The paper "*Automated generation of custom processor core from c code*" by J. Trajkovic et al. presents a novel solution to constructing a processor core from a given application C code. The proposed methodology starts with an initial data path design by matching code properties to hardware elements and iteratively refines it under given design constraints. The experimental results show that the technique scales very well with the size of the C code, and demonstrate the efficiency of the technique on wide range of applications, from standard academic benchmarks to industrial size examples like the MP3 decoder.

The paper "*Hardware and software synthesis of heterogeneous systems from dataflow programs*" by G. Roquier et al. stresses that sequential programming model does not naturally expose potential parallelism to target heterogeneous platforms. Therefore, this work presents a design method that automatically generates hardware and software components and their interfaces, from a unique high-level description of the application, based on the dataflow paradigm. The work targets heterogeneous architectures composed by reconfigurable hardware units and multicore processors. Experimental results using several video coding algorithms show the effectiveness of the approach both in terms of portability and scalability.

5. Design for Error Resilience

The paper "*Selectively fortifying reconfigurable computing device to achieve higher error resilience*" by M. Lin. et al. introduces the concept of Selectively Fortified Computing (SFC) for mission-critical applications with limited inherent error resilience. The SFC methodology differs from the conventional approaches that use static temporal and/or spatial redundancy and complicated error prediction or estimation techniques. It selectively allocates hardware redundancy for the key components of a design in order to maximize its overall error resilience. The experimental results from a 720P H.264/AVC encoder prototype implemented with a Virtex 5 device demonstrated the effectiveness of SFC operating under a wide range of error rates.

6. Concluding Remarks

ESL design area is a fast evolving field. We hope this special issue would provide a snapshot of the current research activities in ESL design area and offer useful references for researchers who are interested in this exciting field. Finally, we would like to thank all the 29 reviewers for this special issue wholeheartedly, whose effort has made this special issue successful.

Deming Chen
Kiyoung Choi
Philippe Coussy
Yuan Xie
Zhiru Zhang

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 105250, 14 pages
doi:10.1155/2012/105250

Research Article

Parametric Yield-Driven Resource Binding in High-Level Synthesis with Multi-V_{th}/V_{dd} Library and Device Sizing

Yibo Chen,[1] Yu Wang,[2] Yuan Xie,[1] and Andres Takach[3]

[1] *Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA*
[2] *Department of Electronics Engineering, Tsinghua University, Beijing 100084, China*
[3] *Design Creation and Synthesis, Mentor Graphics Corporation, Wilsonville, OR 97070, USA*

Correspondence should be addressed to Yibo Chen, yxc236@cse.psu.edu

Received 3 August 2011; Revised 4 January 2012; Accepted 15 January 2012

Academic Editor: Zhiru Zhang

The ever-increasing chip power dissipation in SoCs has imposed great challenges on today's circuit design. It has been shown that multiple threshold and supply voltages assignment (multi-V_{th}/V_{dd}) is an effective way to reduce power dissipation. However, most of the prior multi-V_{th}/V_{dd} optimizations are performed under deterministic conditions. With the increasing process variability that has significant impact on both the power dissipation and performance of circuit designs, it is necessary to employ statistical approaches in analysis and optimizations for low power. This paper studies the impact of process variations on the multi-V_{th}/V_{dd} technique at the behavioral synthesis level. A multi-V_{th}/V_{dd} resource library is characterized for delay and power variations at different voltage combinations. Meanwhile, device sizing is performed on the resources in the library to mitigate the impact of variation, and to enlarge the design space for better quality of the design choice. A parametric yield-driven resource binding algorithm is then proposed, which uses the characterized power and delay distributions and efficiently maximizes power yield under a timing yield constraint. During the resource binding process, voltage level converters are inserted between resources when required. Experimental results show that significant power reduction can be achieved with the proposed variation-aware framework, compared with traditional worstcase based deterministic approaches.

1. Introduction

Integrating billions of transistors on a single chip with nanoscale transistors has resulted in great challenges for chip designers. One of these challenges is that the pace of productivity gains has not kept up to address the increases in design complexity. Consequently, we have seen a recent trend of moving design abstraction to a higher level, with an emphasis on *electronic system level (ESL)* design methodologies. A very important component of ESL is raising the level of abstraction of hardware design. High-level synthesis (HLS) provides this component by providing automation to generate optimized hardware from a high-level description of the function or algorithm to be implemented in hardware. HLS generates a cycle-accurate specification at the register-transfer level (RTL) that is then used in existing ASIC or FPGA design methodologies. Commercial high-level synthesis tools [1] have recently gained a lot of attention as evidenced in recent conference HLS workshops (DATE2008, DAC2008, and ASPDAC2009), conference panels, and publications that track the industry. While high-level synthesis is able to quickly generate implementations of circuits, it is not intended to replace the existing low-level synthesis. The major benefit coming from high-level synthesis is the high design efficiency, the ability to perform fast prototyping, functional verification, and early-stage design space exploration, which in turn provide guidance on succeeding low-level design steps and help produce high-quality circuits.

Power consumption and process variability are among other critical design challenges as technology scales. While it is believed that tackling these issues at a higher level of the design hierarchy can lead to better design decisions, a lot of work has been done on low-power-high-level synthesis [2–4] as well as process-variation-aware-high-level synthesis [5–8]. These techniques have been successfully implemented but most of the existing work focuses on one side of the issues

in isolation. Recently, Srivastava et al. [9] explore the multi-$V_{th}/V_{dd}/T_{ox}$ design space with the consideration of process variations at the gate level. Nevertheless, variation-aware-low power exploration for behavioral synthesis is still in its infancy.

Multiple threshold and supply voltages assignment (multi-V_{th}/V_{dd}) has been shown as an effective way to reduce circuit power dissipation [2, 3, 10, 11]. Existing approaches assign circuit components on critical paths to operate at a higher V_{dd} or lower V_{th}, and noncritical portions of the circuit are made to operate at lower V_{dd} or higher V_{th}, respectively. The total power consumption is thus reduced without degrading circuit performance. However, nowadays circuit performance is affected by process variations. If the variations are underestimated, for example, using nominal delays of circuit components to guide the design, non-critical components may turn to critical ones due to the variations, and circuit timing constraints may be violated. On the other hand, in existing corner-based worst-case analysis, variations are overestimated resulting in design specs that are hard to meet, and this consequently increases design effort and degrades circuit performance.

Device sizing is a well-studied technique for performance and power tuning at gate- or circuit-level [12]. To improve performance, upsizing of a high-V_{th} transistor, which increases switching power and die area, can be traded off against using a low-V_{th} transistor, which increases leakage power. Therefore, combining multi-V_{th} assignment and device sizing as integrated problem, can increase the design flexibility and further improve the design quality. Meanwhile, in terms of mitigating process variations, it is possible that increasing the size of transistors can reduce the randomness of the device parameters through averaging.

This paper presents a variation-aware power optimization framework in high-level synthesis using simultaneous multi-V_{th}/V_{dd} assignment and device sizing. Firstly, the impact of parameter variations on the delay and power of circuit components is explored at different operating points of threshold and supply voltages. Device sizing is then performed to mitigate the impact of variations and to enlarge the design space for better quality of the design choice. A variation-characterized resource library containing the parameters of delay and power distributions at different voltage "corners" and different device sizes, is built once for the given technology, so that it is available for high-level synthesis to query the delay/power characteristics of resources. The concept of parametric yield, which is defined as the probability that the design meets specified constraints such as delay or power constraints is then introduced to guide design space exploration. Statistical timing and power analysis on the data flow graph (DFG) is used to populate the delay and power distributions through the DFG and to estimate the overall performance and power yield of the entire design. A variation-aware resource binding algorithm is then proposed to maximize power yield under a timing yield constraint, by iteratively searching for the operations that have the maximum potential of performance/power yield improvement, and replacing them with better candidates in the multi-V_{th}/V_{dd} resource library. During the resource binding process, voltage level converters are inserted for chaining of resource units having different V_{dd} supplies.

The contribution of this paper can be summarized as follow:

(i) first, this is the first work to apply multi-V_{th}/V_{dd} techniques during high-level synthesis under the context of both delay and power variations. A flow for variation-aware power optimization in multi-V_{th}/V_{dd} HLS is proposed. This flow includes library characterization, statistical timing and power analysis methodologies for HLS, and resource binding optimization with variation-characterized multi-V_{th}/V_{dd} library;

(ii) combined multi-V_{th}/V_{dd} assignment and device sizing for high-level synthesis are performed at the granularity of function unit level, to improve the design quality and at the same time to reduce the design complexity;

(iii) voltage level conversion is explored during the resource binding in high-level synthesis, enabling the full utilization of multi-V_{dd} components for parametric yield maximization.

2. Related Work

Prior research work tightly related to this paper mainly falls into two categories: (1) gate level power minimization by simultaneous multi-V_{th} assignment and gate sizing; (2) low-power high-level synthesis using multi-V_{th} or multi-V_{dd}; (3) process variation aware high-level synthesis.

Several techniques were proposed to consider V_{th} allocation and transistor sizing as an integrated problem [13–16]. Wei et al. [14] presented simultaneous dual-V_{th} assignment and gate sizing to minimize the total power dissipation while maintaining high performance, while Karnik et al. [16] improved the simultaneous V_{th} allocation and device sizing using a Lagrangian Relaxation method. However, all of the reported techniques focus on tuning at transistor level or gate-level. While the fine granularity can yield optimal results, it also lead to high design complexity.

Shiue [2] proposed low-power scheduling schemes with multi-V_{dd} resources by maximizing the utilization of resources operating at reduced supply voltages. Khouri and Jha [3] performed high-level synthesis using a dual-V_{th} library for leakage power reduction. Tang et al. [4] formulated the synthesis problem using dual-V_{th} as a maximum weight-independent set (MWIS) problem, within which near-optimal leakage power reduction is achieved with greatly reduced run time. Very recently, Insup et al. explored optimal register allocation for high-level synthesis using dual supply voltages [17]. However, all of these techniques were applied under deterministic conditions without taking process variation into consideration.

Process variation-aware high-level synthesis has recently gained much attention. Jung and Kim [6] proposed a timing yield-aware HLS algorithm to improve resource sharing and reduce overall latency. Lucas et al. [8] integrated timing-driven floorplanning into the variation-aware high-level

design. Mohanty and Kougianos's work [18] took into account the leakage power variations in low-power high-level synthesis; however, the major difference between [18] and our work is that, the delay variation of function units was not considered in [18], so the timing analysis during synthesis was still deterministic. Recently, Wang et al. [19] proposed a joint design-time optimization and postsilicon tuning framework that tackles both timing and power variations. Adaptive body biasing (ABB) was applied to function units to reduce leakage power and improve power yield.

3. Multi-V_{th}/V_{dd} Library Characterization under Process Variations

Scheduling and resource binding are key steps during the high-level synthesis process. The scheduler is in charge of determining the sequencing the operations of a control/data flow graph (CDFG) in control steps and within control steps (operator chaining) while obeying control and data dependencies and cycle constraints while optimizing for area/power/performance. The binding process binds operations to hardware units in the resource library to complete the mapping from abstracted descriptions of circuits into practical designs. This section presents the characterization of the variation-aware multi-V_{th}/V_{dd} resource library, including the delay and power characterization flow and the selection of dual threshold and supply voltages.

3.1. Variation-Aware Library Characterization Flow. In order to facilitate the design space exploration while considering process variations, the resource library of functional units for HLS has to be characterized for delay/power variations. As shown in Figure 1, under the influence of process variations, the delay and power of each component are no longer fixed values, but represented by probability density functions (PDFs). Consequently, the characterization of function units with delay and power variations requires statistical analysis methodologies.

Process variations come from a set of sources, including random doping fluctuation (RDF) [20] and geometric variations of the gate (primarily on channel length) [21]. Since both RDF and channel length variations manifest themselves as fluctuations on the effective threshold voltage of the transistor [22], their effects can be expressed by the variations of V_{th}. Since this work focuses on demonstrating the effectiveness of variation-aware synthesis, rather than a comprehensive modeling of all variation effects, we try to focus on V_{th} variations with a simplified assumption of normal distribution of V_{th} variations, rather than covering all physical-level variation factors with different distributions. The magnitude of V_{th} variations in real circuits can be obtained via on-chip sensing and measurement. In this work, we use NCSU FreePDK 45 nm technology library [23] for all the characterization and experiments. We set the standard deviation σ of V_{th} to be 50 mV, which is projected from the silicon measurement data in [24].

We then use a commercial gate-level statistical timing analysis tool, Synopsys PrimeTime *VX* to perform the

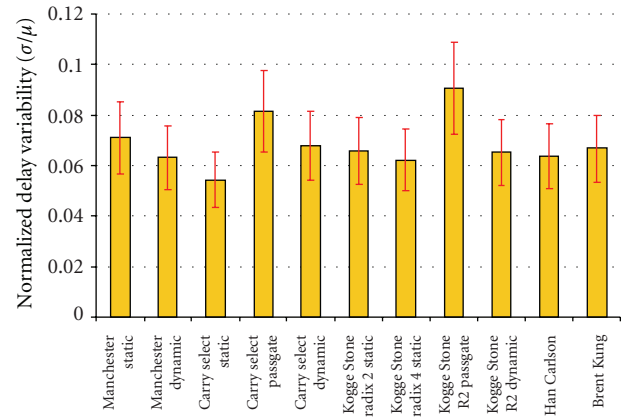


Figure 1: The delay variation for 16-bit adders in IBM Cu-08 technology (courtesy of IBM).

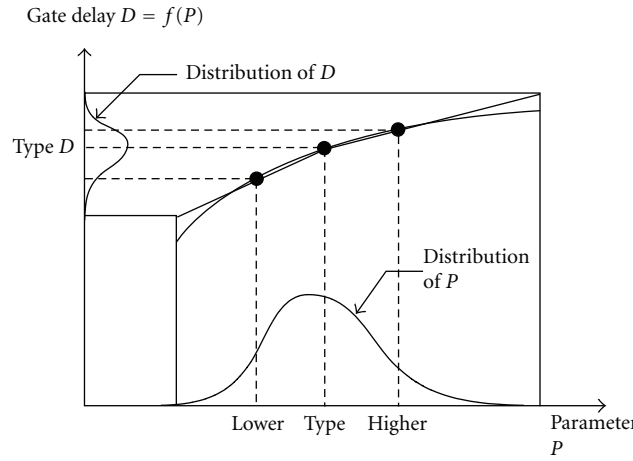


Figure 2: Calculating the delay distribution as a function of parameter P.

characterization. This variation-aware tool increases the accuracy of timing analysis by considering the statistical distribution of device parameters such as threshold voltage and gate oxide thickness. Given the distribution of a device parameter P, PrimeTime *VX* calculates the distribution of gate delay continuously throughout the range of values, using linear interpolation and extrapolation at the library-defined functional operating points, as shown in Figure 2. Validation against SPICE Monte Carlo statistical analysis [25] shows that PrimeTime *VX* analysis holds the similar accuracy but significantly reduces the running time.

The characterization flow takes as input the statistical distributions of process parameters (V_{th} in this work) and generates the statistical distributions of delay and power for each resource in the library. To characterize the delay of function units under the impact of process variations, the following steps are performed:

(1) all the standard cells in a technology library are characterized using variation-aware analysis, and the

results including parameters of cell delay distributions are collected to build a variation-aware technology library;

(2) the function units used in HLS are then synthesized and linked to the variation-aware technology library;

(3) statistical timing analysis for the function units is performed using PrimeTime *VX*, and the parameters of delay distributions are reported.

Statistical power characterization for function units in the resource library can be done using Monte Carlo analysis in SPICE. The power consumption of function units consists of dynamic and leakage components. While dynamic power is relatively immune to process variation, leakage power is greatly affected and becomes dominant as technology continues scaling down [26]. Therefore, in this paper only leakage power is characterized using statistical analysis. However, this do not mean that considerations for dynamic power can be omitted. In fact, dynamic power optimization in high-level synthesis has been a well-explored topic [1]. Our variation-oriented work emphasizing leakage power optimization can be stacked on or integrated into existing power optimization approaches in high-level synthesis, to further reduce the total power consumption of circuits.

According to Berkeley short-channel BSIM4 model [27], higher threshold voltages can lead to exponential reduction in leakage power, which is given by:

$$I_{\text{leakage}} = I_0 \exp\left(\frac{V_{\text{gs}} - V_{\text{th}}}{nV_t}\right)\left(1 - \exp\left(\frac{-V_{\text{ds}}}{V_t}\right)\right), \tag{1}$$

$$I_0 = \mu_0 C_{\text{ox}} \frac{W}{L}(n-1)V_{\text{th}}^2, \tag{2}$$

where I_{leakage} is the gate leakage current, V_{gs} and V_{ds} are the gate voltages, V_t is the thermal voltage, and n is the subthreshold swing factor. Since we assume that the device parameter V_{th} follows normal distribution, I_{leakage} follow log-normal distribution. Therefore, the leakage power of a function units is the sum of a set of log-normal distributions, which describe the leakage power of each library cell. According to [28], the sum of several log-normal random variables can be approximated by another log-normal random variable, as shown in (3):

$$P_{\text{FU}} = P_1 + P_2 + \cdots + P_n = k_1 e^{V_1} + k_2 e^{V_2} + \cdots + k_n e^{V_n}, \tag{3}$$

where P_{FU} describes the leakage power distribution of the function unit; while P_n, k_n, and V_n are the corresponding variables for library cells that build up the function unit. The mean and deviation of P_{FU} can be estimated via iterative moment matching out of the leakage power distributions of library cells [28].

The power characterization flow is stated as follows. Process variations are set in the MOS model files, and 1000 runs of Monte Carlo iterations are performed for each library cell. After the characterization, the parameters of the leakage power distributions of library cells are extracted.

Note that in our work we only characterize subthreshold leakage, since it starts dominant for technology nodes of 45 nm and below. The gate leakage can also be characterized with similar methods.

3.2. Multi-V_{th}/V_{dd} Library Characterization. Previous implementations using multiple threshold and supply voltages in conjunction have shown a very effective reduction in both dynamic and leakage power [11]. Therefore, our approach considers the combination of dual threshold and dual supply voltages, and characterizations are performed at the four "corners" of voltage settings, namely $(V_{\text{th}}^L, V_{\text{dd}}^H)$, $(V_{\text{th}}^H, V_{\text{dd}}^H)$, $(V_{\text{th}}^L, V_{\text{dd}}^L)$, and $(V_{\text{th}}^H, V_{\text{dd}}^L)$, where $(V_{\text{th}}^L, V_{\text{dd}}^H)$ is the nominal case and the other three are low-power settings. Note that although only 4 voltage settings are discussed in this paper, it is natural to extend the approach presented here to deal with more voltage settings. To reduce the process technology cost, in this paper, the multi-V_{th}/V_{dd} techniques are applied at the granularity of function units. That means, all the gates inside a function unit operate at the same threshold and supply voltages. Voltages only differ from function units to function units.

The selection of appropriate values of threshold and supply voltages for power minimization has been discussed under deterministic conditions [11]. Rules of thumb are derived for the second V_{dd} and V_{th} as functions of the original voltages [11]:

$$\begin{aligned} V_{\text{dd}}^L &= 0.43V_{\text{dd}}^H + 0.82V_{\text{th}}^H + \frac{0.72}{K} - \frac{0.55}{K^2} - 0.2, \\ V_{\text{th}}^L &= -0.024V_{\text{dd}}^H + 1.14V_{\text{th}}^H + \frac{0.72}{K} - \frac{0.49}{K^2} - 0.18, \end{aligned} \tag{4}$$

where K stands for the initial ratio between dynamic and static power. While the empirical models in [11] are validated on actual circuit benchmarks [29], they may not be accurate under the impact of process variations. A refined model taking into account the process variations is presented in [9]. As shown in Figure 3, the total power reduction with variation awareness is plotted under different combinations of $V\text{th}_2$ (V_{th}^H) and $V\text{dd}_2$ (V_{dd}^L), and this guides the optimal value selection in this work.

The characterization results (which will be further discussed in Section 6) show that, power reduction is always achieved at the cost of delay penalties. Moreover, larger delay variations are observed for slower units operating at high-V_{th} or low-V_{dd}, which means larger probability of timing violations when they are placed on the near-critical paths. This further demonstrates the necessity of statistical analysis and parametric yield-driven optimization approaches.

3.3. Device Sizing for the Resource Library. Conventionally, device sizing is an effective technique to optimize CMOS circuits for dynamic power dissipation and performance. In this work, we show that device sizing may also be utilized to mitigate the impact of process variations. As previously mentioned, the sources of process variations mainly consists of random doping fluctuation (RDF) [20] and geometric variations (GVs). GV affect the real V_{th} through the drain

induced barrier lowering (DIBL) effect. Originally, both RDF and GV have almost the equal importance in determining the V_{th} variance. As we propose to use low-V_{dd} and high-V_{th} resource units in the design, the difference between supply voltage and threshold voltage diminishes, and this reduces DIBL effect. As a result, the uncertainty in V_{th} arising from GV rapidly falls as V_{dd}. On the other hand, the RDF-induced V_{th} variation is independent of V_{dd} changes and solely a function of channel area [30]. Therefore, V_{th} variation resulting from RDF becomes dominating as V_{dd} approaches V_{th}.

Due to the independent nature of RDF variations, it is possible to reduce their impact on circuit performance through averaging. Therefore, upsizing the device can be an effective way for variability mitigation with enlarged channel area. According to [31], V_{th} variance $\sigma_{V_{th}}$ resulting from RDF is roughly proportional to $(WL)^{-1/2}$, which means we can either increase the transistor width or channel length or both. Conventional sizing approaches focus on tuning the transistor width for performance. In terms of process variability mitigation, the measurement data of V_{th} variation for 4 different device sizes is plotted in Figure 4, which shows that increasing transistor width is a more effective way to reduce the V_{th} variance [24]. Although larger transistor width means larger leakage power, the fluctuations on leakage power are reduced, and the design space for resource binding is significantly enlarged, thus using resources with larger size in the design may still be able to improve the parametric yield.

In this work, we upsize all the function units in the resource library to generate alternatives for power tuning and variability mitigation. The sizing is performed on all the gates with two different settings: the basic size (1W1L) and the double-width size (2W1L). We then perform the variation characterization for the upsized function units under all the four voltage "corners" presented in the previous section. The characterization results will be presented in Section 6.

4. Yield Analysis in Statistical High-Level Synthesis

In this section, a parametric yield analysis framework for statistical HLS is presented. We first show the necessity of statistical analysis by a simple motivational example and then demonstrate the statistical timing and power analysis for HLS as well as the modeling and integration of level converters for multi-V_{dd} HLS.

4.1. Parametric Yield. To bring the process-variation awareness to the high-level synthesis flow, we first introduce a new metric called parametric yield. The parametric yield is defined as the probability of the synthesized hardware meeting a specified constraint Yield $= P(Y \leq Y_{max})$, where Y can be delay or power.

Figure 5 shows a motivational example of yield-aware analysis. Three resource units $R1$, $R2$, and $R3$ have the same circuit implementation but operate at different supply or threshold voltages. Figure 5 shows the delay and power distributions for $R1$, $R2$, and $R3$. In this case the mean power follows up $\mu_P(R3) < \mu_P(R2) < \mu_P(R1)$, and the mean delay follows $\mu_D(R1) < \mu_D(R2) < \mu_D(R3)$, which is as expected since power reduction usually comes at the cost of increased delay. The clock cycle time T_{CLK} and the power consumption constraint P_{LMT} (e.g., the TDP (thermal design power) of most modern microprocessors) are also shown in the figure. If the variation is disregarded and nominal-case analysis is used, any of the resource units can be chosen since they all meet timing. In this case, $R3$ would be chosen as it has the lowest power consumption. However, under a statistical point of view, $R3$ has a low timing yield (approximately 50%) and is very likely to cause timing violations. In contrast, with corner-based worst-case analysis only $R1$ can be chosen under the clock cycle time constraint (the worst-case delay of

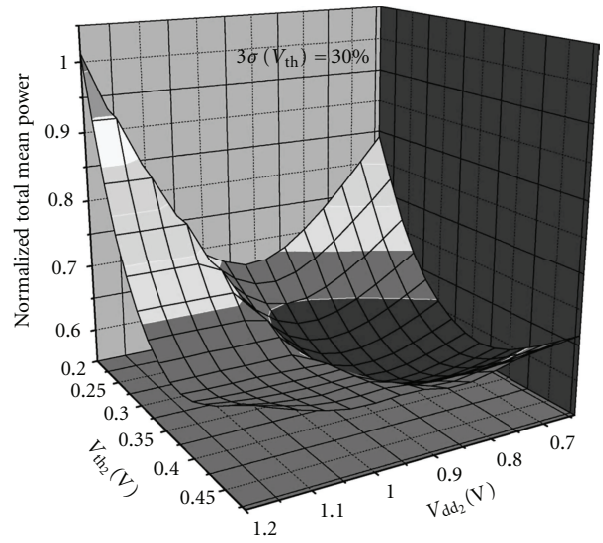


Figure 3: Optimal selection of dual threshold and supply voltages under process variation.

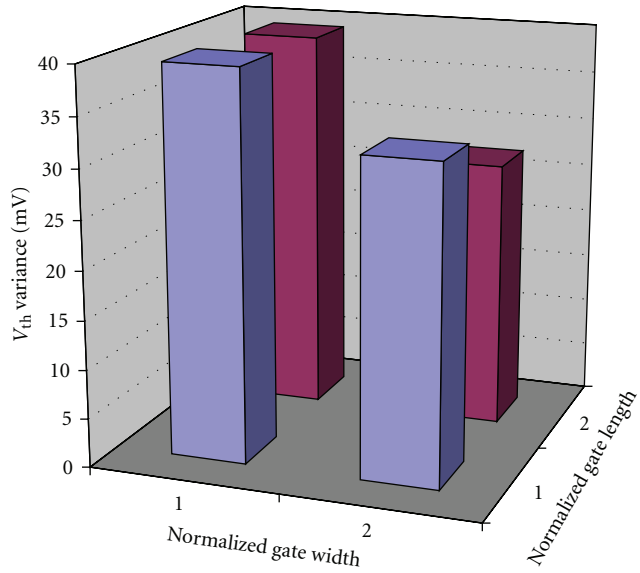


Figure 4: V_{th} variations with difference device sizes (normalized to minimal size).

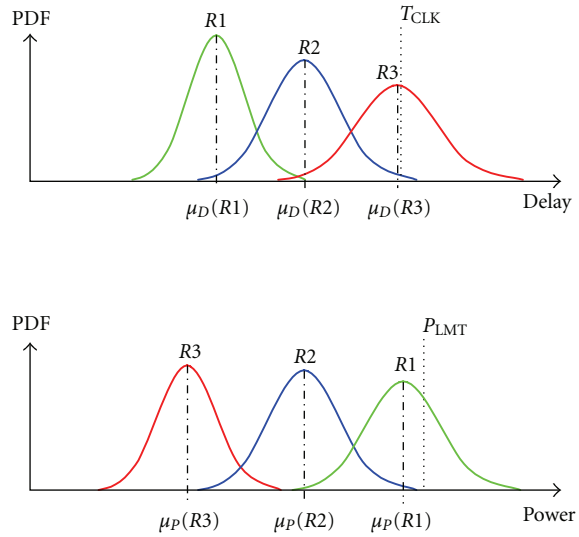


Figure 5: Motivating example of yield-driven synthesis.

$R2$ slightly violates the limit), whereas $R1$ has a poor power yield. In fact, if we set a timing yield constraint instead of enforcing the worst-case delay limitation, $R2$ can be chosen with a slight timing yield loss but a well-balanced delay and power tradeoff. Therefore, a yield-driven statistical approach is needed for exploring the design space to maximize one parametric yield under other parametric yield constraints.

4.2. Statistical Timing and Power Analysis for HLS. High-level synthesis (HLS) is the process of transforming a behavioral description into register level structure description. Operations such as additions and multiplications in the DFG are scheduled into control steps. During the resource allocation and binding stages, operations are bound to corresponding function units in the resource library meeting type and latency requirements.

Given the clock cycle time T_{CLK}, the timing yield of the entire DFG, Yield_T is defined as

$$\text{Yield}_T = P(T_1 \le T_{CLK}, T_2 \le T_{CLK}, \ldots, T_n \le T_{CLK}), \tag{5}$$

where $P()$ is the probability function, $T_1, T_2, \ldots, T_n$ are the arrival time distributions at control step $1, 2, \ldots, n$, respectively.

The arriving time distribution of each clock cycle can be computed from the delay distributions of function units bound at that cycle. Two operations, sum and max, are used to compute the distributions:

(i) sum operation is used when two function units are chained in cascade within a clock cycle, as shown in CC1 and CC2 of Figure 6. The total delay can be computed as the "sum" of their delay distributions (normal distribution assumed);

(ii) max operation is used when the outputs of two or more units are fed to another function unit at the same clock cycle, as shown in CC1 of Figure 6. The "maximum" delay distribution can be computed out of the contributing distributions using tightness probability and moment matching [19].

With these two operations, the arriving time distribution of each clock cycle is computed, and the overall timing yield of the DFG is obtained using (5).

The total power consumption of a DFG can be computed as the sum of the power consumptions of all the function units used in the DFG. Given a power limitation P_{LMT}, the power yield of the DFG Yield_P is computed as the probability that total power P_{DFG} is less than the requirement, as expressed in (6):

$$\text{Yield}_P = P(P_{DFG} \le P_{LMT}). \tag{6}$$

Since dynamic power is relatively immune to process variations, it is regarded as a constant portion which only affects the mean value of the total power consumption. Therefore, the total power is still normally distributed, although statistical analysis is only applied to the leakage power. As aforementioned in Section 3, our proposed yield-driven statistical framework can be stacked on existing approaches for dynamic power optimization, to further reduce the total power consumption of circuits.

4.3. Voltage Level Conversion in HLS. In designs using multi-V_{dd} resource units, voltage level convertors are required when a low-voltage resource unit is driving a high-voltage resource unit. Level conversion can be performed either synchronously or asynchronously. Synchronous level conversion is usually embedded in flip-flops and occurs at the active clock edge, while asynchronous level converters can be inserted anywhere within the combinational logic block.

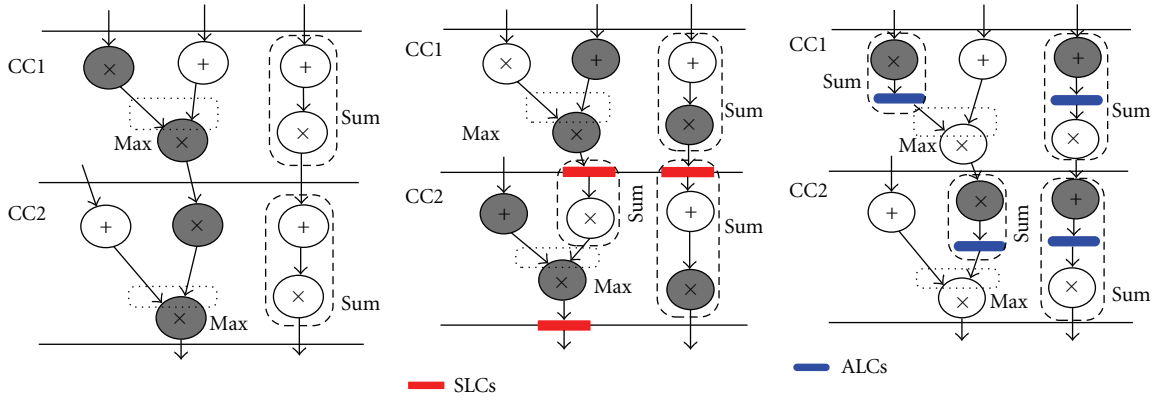
When process variations are considered, asynchronous level converters are even more favorable, because they are not bounded by clock edges, and timing slacks can be passed through the converters. Therefore, time borrowing can happen between low-voltage and high-voltage resource units. As slow function units (due to variations) may get more time to finish execution, the timing yield can be improved, and the impact of process variations is consequently reduced.

While many fast and low-power level conversion circuits have been proposed recently, this paper uses the multi-V_{th} level converter presented in [32], taking the advantage that there is no extra process technology overhead for multi-V_{th} level converters, since multi-V_{th} is already deployed for function units. The proposed level converter is composed of two dual V_{th} cascaded inverters. Its delay and power are then characterized in HSPICE using the listed parameters [32].

The delay penalty of a level converter can be accounted by summing its delay with the delay of the function unit it is associated to. The power penalty can be addressed by counting the level converters used in the DFG and adding the corresponding power to the total power consumption.

5. Yield-Driven Power Optimization Algorithm

In this section, we propose our yield-driven power optimization framework based on the aforementioned statistical



(a) No level conversion (b) synchronous conversion (c) synchronous conversion

FIGURE 6: Timing yield computation in multi-V_{dd} high-level synthesis with different level conversions. Shadowed operations are bound to function units with low-supply voltages, and the bars indicate the insertion of level converters.

timing and power yield analysis. During the high-level synthesis design loop, resource binding selects the optimal resource instances in the resource library and binds them to the scheduled operations at each control step. A variation-aware resource binding algorithm is then proposed to maximize power yield under a preset timing yield constraint, by iteratively searching for the operations with the maximum potential of timing/power yield improvement, and replacing them with better candidates in the multi-V_{th}/V_{dd} resource library.

5.1. Variation-Aware Resource Binding Algorithm Overview. Our variation-aware resource binding algorithm takes a search strategy called variable depth search [19, 33, 34] to iteratively improve the power yield under performance constraints. The outline of the algorithm is shown in Algorithm 1, where a DFG is initially scheduled and bound to resource library with nominal voltages (V_{th}^{L}, V_{dd}^{H}). A lower-bound constraint on the timing yield is set, so that the probability of the design can operate at a given clock frequency, will be larger than or equal to a preset threshold (e.g., 95%). In the algorithm, a move is defined as a local and incremental change on the resource bindings. As shown in the sub routine GENMOVE in Algorithm 1, the algorithm generates a set of moves and finds out a sequence of moves that maximizes the accumulated *gain*, which is defined as $\alpha * \Delta\text{Yield}_D + \Delta\text{Yield}_P$, where α is a weighting factor to balance the weights of timing and power yield improvements. The optimal sequence of moves is then applied to the DFG, and the timing and power yields of the DFG are updated before the next iteration. The iterative search ends when there is no yield improvement or the timing yield constraint is violated.

Note that our worst-case resource binding algorithm uses the same search strategy (variable depth search) [19, 33, 34] as the variation-aware resource binding algorithm. The key difference is that, instead of iteratively improving the power yield under performance constraints, the worst-case resource binding algorithm iteratively reduces the power consumption under specified performance constraints, where both the power consumption calculation and performance constraints are specified as deterministic numbers, rather than using the concept of power yield and performance yield.

5.2. Voltage Level Conversion Strategies. Moves during the iterative search may result in low-voltage resource units driving high-voltage resource units. Therefore, level conversion is needed during resource binding. However, if resources are selected and bound so that low-voltage resource units never drive high-voltage ones, level conversion will not be necessary, and the delay and power overheads brought by level converters can be avoided. This reduces the flexibility of resource binding for multivoltage module combinations, and may consequently decrease the attainable yield improvement. The tradeoff in this conversion-avoidance strategy, can be explored and evaluated within our proposed power optimization algorithm.

We also incorporate other two strategies of level conversions in the power optimization algorithm for comparison. All the three strategies are listed as follows:

(i) level conversion avoidance: resource binding is performed with the objective that low-voltage resources never drive high-voltage ones. As shown in Figure 6(a), no dark-to-light transition between operations is allowed (while dark operations are bound to low-V_{dd} units), so that level conversion is avoided. This is the most conservative strategy;

(ii) synchronous level conversion: voltage level conversion is done synchronously in the level-converting flip-flops (SLCs). As shown in Figure 6(b), the dark-to-light transition only happens at the beginning of each clock cycles. The flip-flop structure proposed in [35] is claimed to have smaller delay than the combination of an asynchronous converter and a conventional flip-flop. However, as discussed previously, synchronous level conversion may reduce the flexibility of resource binding as well as the possibility

```
VABinding(DFG, ResLib, Constraints, LCStrategy)
   ▷ Initialization
(1)   Scheduling using ASAP strategy
(2)   Initial Binding to (V_th^L, V_dd^H) resources
   ▷ Variation-aware resource binding
(3)   while ΔYield_P > 0 AND Yield_D ≥ Constraint
(4)      do for i ← 1 to MAXMOVES
(5)            do Gain_i ←GenMove(DFG, ResLib, LCStrategy)
(6)               Append Gain_i to Gain_List;
(7)         Find subsequence Gain_1, ..., Gain_k in Gain_List
            so that G = Σ_{i=1}^{k} Gain_i is maximized
(8)         if G > 0
(9)            do Accept moves 1 · · · k
(10)                Evaluate ΔYield_P and Yield_D
GenMove(DEG, ResLib, LCStrategy)
(1)   MOVE: Choose a move using steepest descent heuristic [33]
(2)   Check whether and where level conversion is needed
(3)   if LC Strategy = Avoidance AND NeedConversion
(4)      do goto MOVE
(5)   if LC Strategy = Synchronous AND NeedConversion
   ▷Check whether conversion is synchronous or not
(6)      do if  Conversion is inside operation chaining
(7)                  do goto MOVE
(8)   Count the overhead of level conversion
(9)   Evaluate the Gain of this move
(10)  Return Gain
```

Algorithm 1: Outline of the variation-aware resource binding algorithm.

of timing borrowing. The effectiveness of this strategy is to be explored by the optimization algorithm;

(iii) asynchronous level conversionL: asynchronous level converters (ALCs) are inserted wherever level conversion is needed, as dark-to-light transition can happen anywhere in Figure 6. This aggressive strategy provides the maximum flexibility for resource binding and timing borrowing. Although it brings in delay and power overhead, it still has great potential for timing yield improvement.

5.3. Moves Used in the Iterative Search. In order to fully explore the design space, three types of moves are used in the iterative search for resource binding;

(i) resource rebinding: in this move, an operation is assigned to a different function unit in the library with different timing and power characteristics. The key benefit of the multi-V_{th}/V_{dd} techniques is that it provides an enlarged design space for exploration, and optimal improvements are more likely to be obtained;

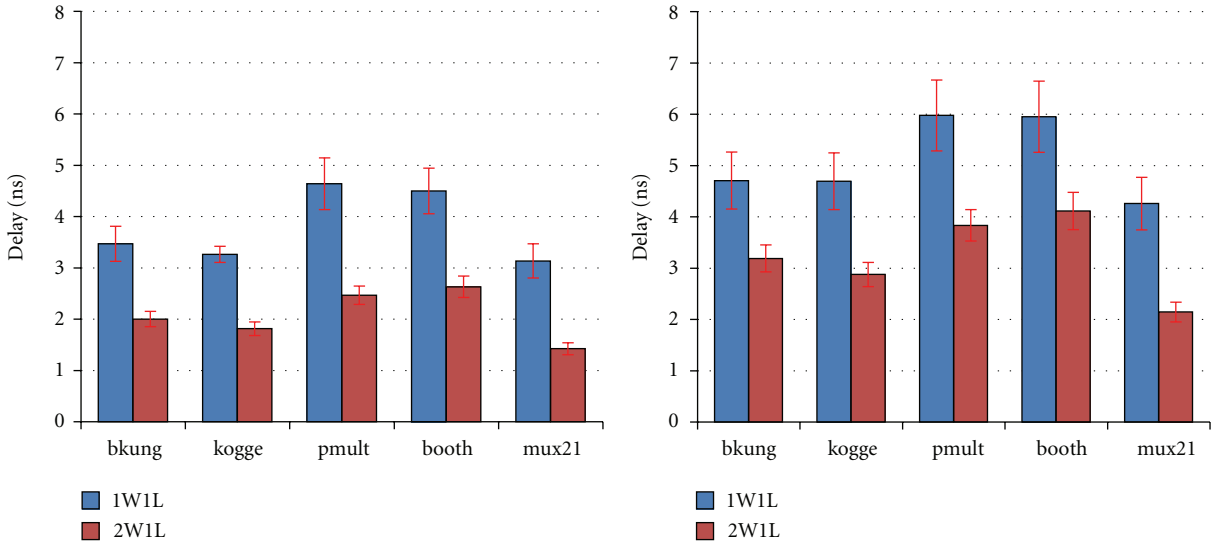
(ii) resource sharing: in this move, two function units that are originally bound to different function units, are now merged to share the same function unit. The type of move reduces the resource usage and consequently improves the power yield;

(iii) resource splitting: in this move, the operation that originally shared function unit with other operations, is split from the shared function unit. This type of move might lead to other moves such as resource rebinding and resource sharing.

After each move, the algorithm checks where the low-supply voltage function units are used and decides whether to insert or remove the level converters, according to the predefined level conversion strategy. If a move is against the strategy, it is revoked, and new moves are generated until a qualifying move is found.

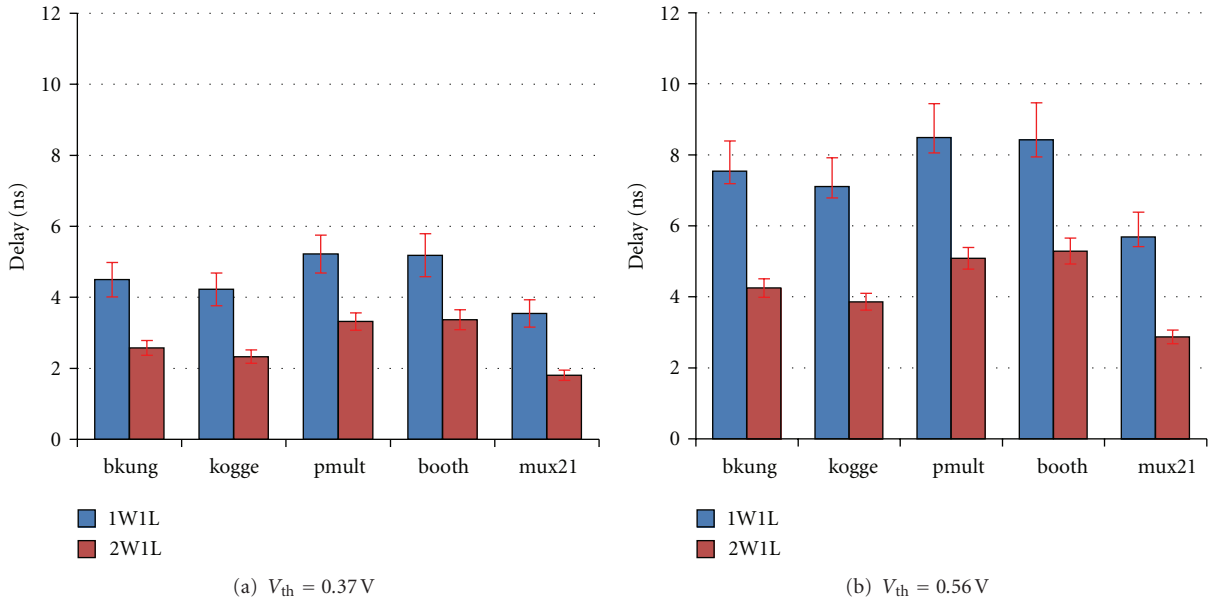
5.4. Algorithm Analysis. It has to be noted that, in the procedure GenMove shown in Algorithm 1, even though the returned Gain might be negative, it still could be accepted. Since the sequence of a cumulative positive gain is considered, the negative gains help the algorithm escape from local minima through hill climbing.

As for the computational complexity, it is generally not possible to give nontrivial upper bounds of run time for local search algorithms [33]. However, for variable depth search in general graph partitioning, Aarts and Lenstra [33] found a near-optimal growth rate of run time to be $O(n \log n)$, where n is the number of nodes in the graph. In our proposed algorithm, the timing and power yield evaluation, as well as the level converter insertion, are performed at each move. Since the yield can be updated using a gradient computation approach [19], the run time for each move is at most $O(n)$. Therefore, the overall run time for the proposed resource binding algorithm is $O(n^2 \log n)$.



(a) $V_{th} = 0.37\,V$ (b) $V_{th} = 0.56\,V$

Figure 7: Delay characterization of function units with multi-V_{th}/V_{dd} and variation awareness.



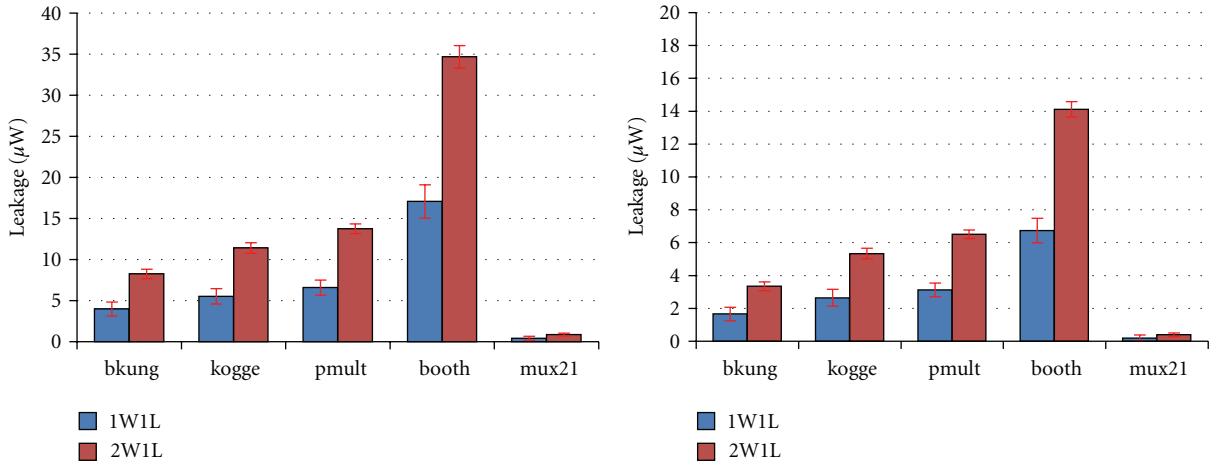
(a) $V_{th} = 0.37\,V$ (b) $V_{th} = 0.56\,V$

Figure 8: Delay characterization of function units with multi-V_{th}/V_{dd} and variation awareness.

6. Experimental Results

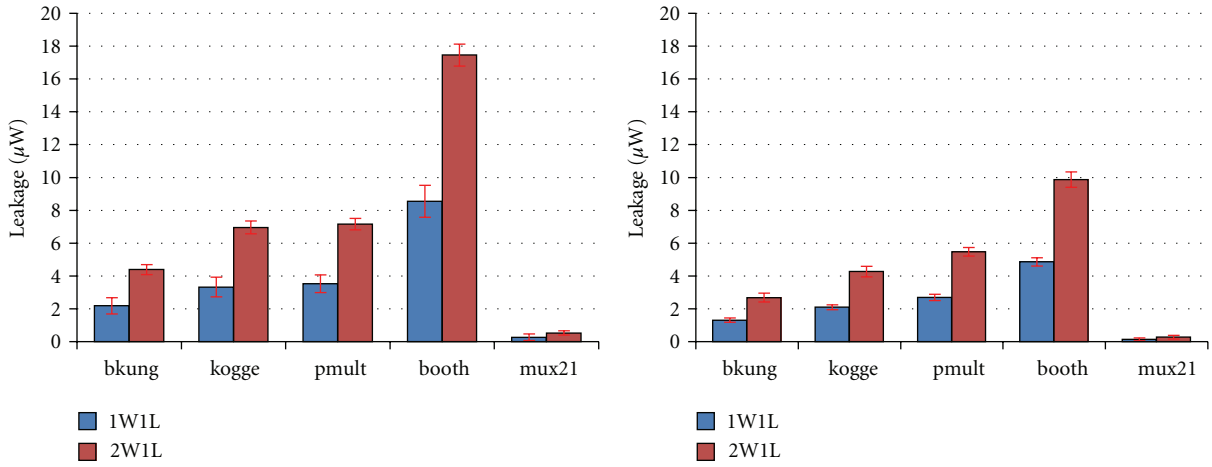
In this section, we present the experimental results of our variation-aware power optimization framework for high-level synthesis. The results show that our method can effectively improve the overall power yield of given designs and reduce the impact of process variations.

We first show the variation-aware delay and power characterization of function units. The characterization is based on NCSU FreePDK 45 nm technology library [23]. The voltage corners for the characterization are set as $V_{th}^{L} = 0.37\,V$, $V_{th}^{H} = 0.56\,V$, $V_{dd}^{L} = 0.9\,V$, and $V_{dd}^{H} = 1.1\,V$. The characterization results for five function units, including two 16-bit adders bkung and kogge, two 8-bit $\times$ 8-bit multipliers pmult and booth, and one 16-bit multiplexer *mux21*, are depicted in Figures 7, 8, 9, 10. In the figures, the color bars show the nominal case values, while the error bars show the deviations. It is clearly shown that with lower V_{dd} and/or higher V_{th}, significant power reductions are achieved at the cost of delay penalty. Meanwhile, up sizing the transistor can improve the circuit performance but also yield to larger power consumption. In terms of variability mitigation, both voltage scaling and device sizing have significant impact on the delay and leakage variations. We can explore this trend



(a) $V_{th} = 0.37$ V (b) $V_{th} = 0.56$ V

Figure 9: Leakage power characterization of function units with multi-V_{th}/V_{dd} and variation awareness.



(a) $V_{th} = 0.37$ V (b) $V_{th} = 0.56$ V

Figure 10: Leakage power characterization of function units with multi-V_{th}/V_{dd} and variation awareness.

further in Figures 11 and 12, where the delay and power distributions of the function unit bkung is sampled at a third V_{th} of 0.45 V. The plotted curves show that the magnitude of delay variation increases for higher V_{th} units, which means larger probabilities of timing violations if these high V_{th} units are placed on near-critical paths. The figures also show that up-sizing the device can effectively reduce the delay and leakage variations, as depicted by the error bars in Figures 11 and 12.

With the variation-aware multi-V_{th}/V_{dd} resource library characterized, our proposed resource binding algorithm is applied on a set of industrial high-level synthesis benchmarks, which are listed in Table 1. A total power limitation P_{LMT} is set for each benchmark to evaluate the power yield improvement. The dynamic power consumption of function units is estimated by Synopsys Design Compiler with multi-V_{th}/V_{dd} technology libraries generated by Liberty NCX. In this work with FreePDK 45 nm technology, the dynamic power is about 2 times the mean leakage power. The power yield before and after the improvement is then computed using (6) in Section 4.2. The proposed resource binding algorithm is implemented in C++, and experiments are conducted on a Linux workstation with Intel Xeon 3.2 GHz processor and 2 GB RAM. All the experiments run in less than 60 s of CPU time.

We compare our variation-aware resource binding algorithm against the traditional deterministic approach, which uses the worst-case ($\mu + 3\sigma$) delay values of function units in the multi-V_{th}/V_{dd} library to guide the resource binding. For deterministic approach, we leverage a commercial HLS tool called Catapult-C to obtain the delay/area/power estimation. The worst-cased based approach will naturally lead to 100% timing yield; however, the power yield is poor as shown in the motivational example in Figure 5. In contrast, our yield-aware statistical optimization algorithm takes the delay and power distributions as inputs, explores the design space

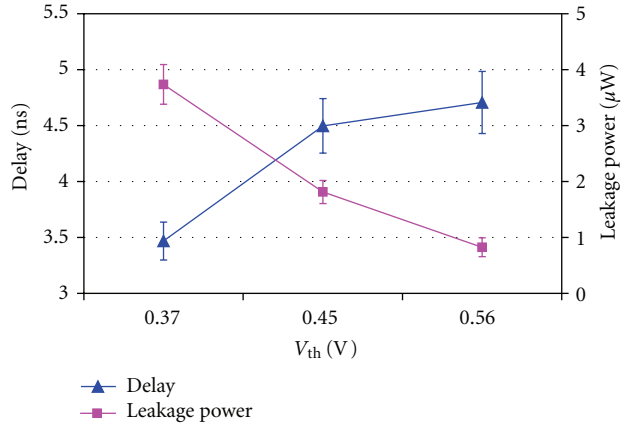


Figure 11: The delay and power tradeoff with increasing V_{th} for bkung with default device sizing (1W1L).

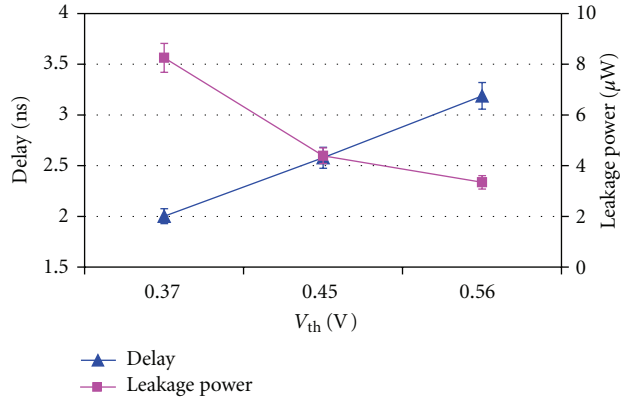


Figure 12: The delay and power tradeoff with increasing V_{th} for bkung with doubled device sizing (2W1L).

Table 1: The profile of test benchmarks.

Name	Nodes number	Edges number	Add number	Mult number
CHEM	33	51	15	18
EWF	34	49	20	12
PR	44	132	26	16
WANG	52	132	26	24
MCM	96	250	64	30
HONDA	99	212	45	52
DIR	150	312	84	64
STEAM	222	470	105	115

with the guidance of YieldGain, and iteratively improves the power yield under a slight timing yield loss. The comparison results are shown in Figures 13, 14, 15 and 16, respectively.

Figure 13 shows the power yield improvement against worst-case delay based approach, with different level conversion strategies. A fixed timing yield constraint of 95% is set for the proposed variation-aware algorithm, using the function units with default device sizes (1W1L). The overheads of the level converters used in this paper are listed in

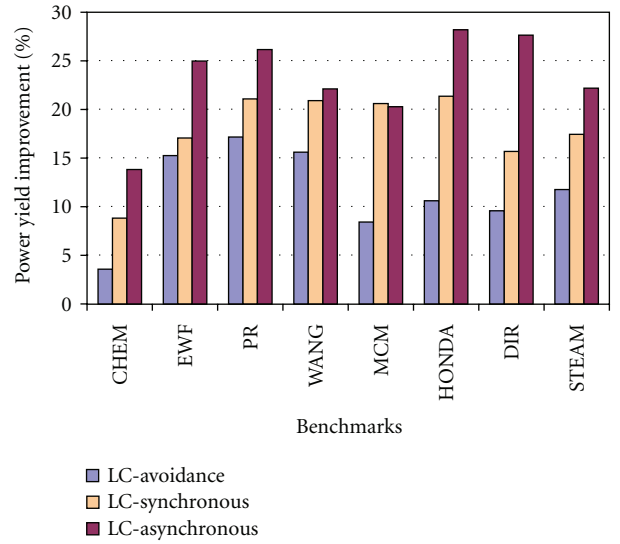


Figure 13: Power yield improvement against deterministic worst-case approach with different level conversion strategies and timing yield constraint of 95%.

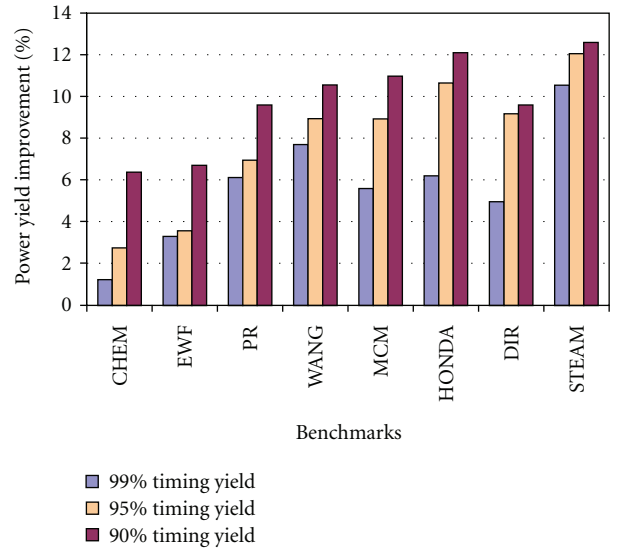


Figure 14: Power yield improvement against deterministic worst-case approach with multi-V_{th} only and different timing yield constraints.

Table 2: The Overheads of voltage level converters.

Type	Delay (ps)	Power (nW)
Synchronous converter	80	Negligible
Asynchronous converter	200	3790

Table 2. The usage of function units and level converters under the three listed conversion strategies (conversion avoidance, synchronous conversion and asynchronous conversion) is listed in Table 3, in which "Vdd-H FUs number" and "Vdd-L FUs number" show the numbers of function units with high/low supply voltages, respectively, and "LCs

Table 3: The usage of function units and level converters with different level conversion strategies.

Bench name	LC-avoidance		LC-synchronous			LC-asynchronous			
	Vdd-H	Vdd-L	Vdd-H	Vdd-L	LCs	Vdd-H	Vdd-L	LCs	Overhead
CHEM	5	1	3	3	1	2	4	2	4.2%
EWF	6	1	4	3	1	3	4	2	3.7%
PR	7	2	6	3	2	4	5	3	4.1%
WANG	9	2	8	3	2	5	6	3	3.4%
MCM	20	4	16	8	4	15	9	5	2.4%
HONDA	22	5	17	10	6	15	12	6	3.9%
DIR	28	4	20	12	8	14	18	12	4.8%
STEAM	34	8	25	19	11	21	23	13	5.0%

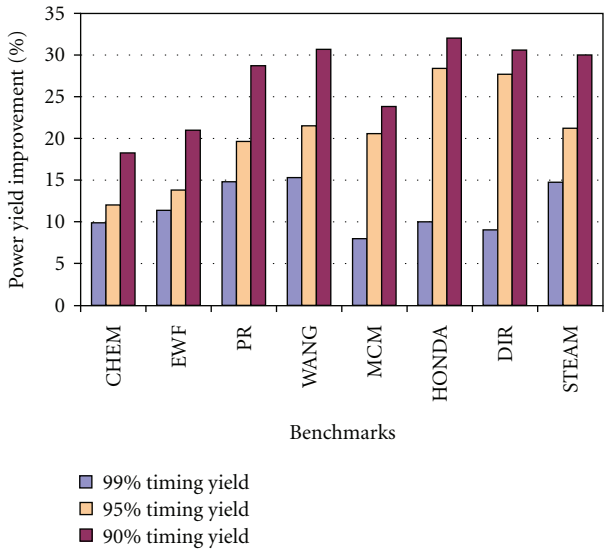


Figure 15: Power yield improvement against deterministic worst-case approach with asynchronous level conversion and different timing yield constraints.

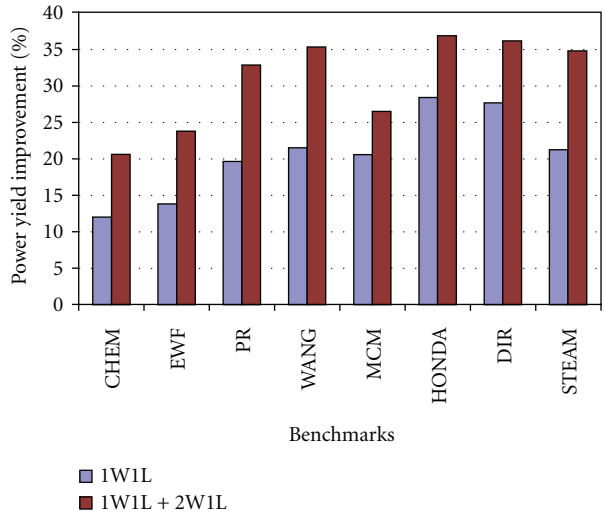


Figure 16: Power yield improvement against deterministic worst-case approach with asynchronous level conversion and different timing yield constraints.

number" counts the number of converters used in the design. The last column counts the total power overhead of the asynchronous level converters. The average power yield improvements for the three strategies are 11.7%, 17.9%, and 22.2%, respectively. From Figure 13 and Table 3 we can see that larger power yield improvements can be achieved when more low-Vdd function units are used in the design. The results also validate our claims in Section 4.3 and Section 5.2 that, asynchronous level conversion is more favorable in statistical optimization because it enables timing borrowing between function units and leads to the timing yield improvement that can compensate the overhead of the converters. Therefore, compared to the synchronous case, more asynchronous converters are used while yielding better results.

Figure 14 shows power yield improvement with multi-V_{th} technique only, which means only the resource units with nominal supply voltage V_{dd}^{H} can be selected. In this case, no level conversion is needed so there is no overhead for level converters. Only function units with default device sizes (1W1L) are used. The average power yield improvements against worst-case delay based approach, under timing yield constraints 99%, 95%, and 90% are 5.7%, 7.9%, and 9.8%, respectively. At timing yield 95%, the average power yield improvement (7.9%) is smaller than the LC-avoidance case (11.5%) in Figure 13, which shows that using multi-V_{dd} resource units can further improve the power yield.

Figure 15 shows the power yield improvement against worst-case delay based approach, under different timing yield constraints. Asynchronous level conversion is chosen in this series of experiments. Only function units with default device sizes (1W1L) are used. The average power yield improvements under timing yield constraints 99%, 95%, and 90% are 11.6%, 20.6%, and 26.9%, respectively. It is clearly shown that, the power yield improvement largely depends on how much timing yield loss is affordable for the design. This will further push forward the design space exploration for a well-balanced timing and power tradeoff.

Figure 16 shows the power yield improvement against worst-case delay-based approach, using function units with different device sizes. Asynchronous level conversion is chosen in this series of experiments, and a fixed timing yield constraint of 95% is set for the proposed variation-aware algorithm. Compared with the average 20.6% yield improvement in the case using default device size (1W1L) only, using

both default-size (1W1L) and double-size (2W1L) resources can lead to an average power yield improvement of 30.9%. Obviously, upsized device with higher performance and smaller variability provide additional flexibility for design space exploration; however, this is achieved at the cost of larger silicon area.

7. Conclusions

In this paper, we investigate the impact of process variations on multi-V_{th}/V_{dd} and device sizing techniques for low-power-high-level synthesis. We characterize delay and power variations of function units under different threshold and supply voltages, and feed the variation-characterized resource library to the HLS design loop. Statistical timing and power analysis for high-level synthesis is then introduced, to help our proposed resource binding algorithm explore the design space and maximize the power yield of designs under given timing yield constraints. Experimental results show that significant power reduction can be achieved with the proposed variation-aware framework, compared with traditional worst-case based deterministic approaches.

Acknowledgment

This work was supported in part by NSF 0643902,0903432, and 1017277, NSFC 60870001/61028006 and a grant from SRC.

References

[1] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, Springer, 2009.

[2] W. T. Shiue, "High level synthesis for peak power minimization using ILP," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 103–112, July 2000.

[3] K. S. Khouri and N. K. Jha, "Leakage power analysis and reduction during behavioral synthesis," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, no. 6, pp. 876–885, 2002.

[4] X. Tang, H. Zhou, and P. Banerje, "Leakage power optimization with dual-Vth library in high-level synthesis," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 202–207, June 2005.

[5] W. L. Hung, X. Wu, and Y. Xie, "Guaranteeing performance yield in high-level synthesis," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '06)*, pp. 303–309, November 2006.

[6] J. Jung and T. Kim, "Timing variation-aware high-level synthesis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '07)*, pp. 424–428, November 2007.

[7] F. Wang, G. Sun, and Y. Xie, "A variation aware high level synthesis framework," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1063–1068, March 2008.

[8] G. Lucas, S. Cromar, and D. Chen, "FastYield: Variation-aware, layout-driven simultaneous binding and module selection for performance yield optimization," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, pp. 61–66, January 2009.

[9] A. Srivastava, T. Kachru, and D. Sylvester, "Low-power-design space exploration considering process variation using robust optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 67–78, 2007.

[10] K. Usami and M. Igarashi, "Low-power design methodology and applications utilizing dual supply voltages," in *Proceedings of the Design Automation Conference (ASPDAC '00)*, pp. 123–128, Yokohama, Japan, 2000.

[11] A. Srivastava and D. Sylvester, "Minimizing total power by simultaneous Vdd/Vth assignment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 5, pp. 665–677, 2004.

[12] C. P. Chen, C. C. N. Chu, and D. F. Wong, "Fast and exact simultaneous gate and wire sizing by lagrangian relaxation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 617–624, ACM, New York, NY, USA, 1998.

[13] S. Sirichotiyakul, T. Edwards, C. Oh et al., "Stand-by power minimization through simultaneous threshold voltage selection and circuit sizing," in *Proceedings of the 1999 36th Annual Design Automation Conference (DAC '99)*, pp. 436–441, June 1999.

[14] L. Wei, K. Roy, and C. K. Koh, "Power minimization by simultaneous dual-Vth assignment and gate-sizing," in *Proceedings of the 22nd Annual Custom Integrated Circuits Conference (CICC '00)*, pp. 413–416, May 2000.

[15] P. Pant, R. K. Roy, and A. Chatterjee, "Dual-threshold voltage assignment with transistor sizing for low power CMOS circuits," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, no. 2, pp. 390–394, 2001.

[16] T. Karnik, Y. Ye, J. Tschanz et al., "Total power optimization by simultaneous dual-Vt allocation and device sizing in high performance microprocessors," in *Proceedings of the 39th Annual Design Automation Conference (DAC '02)*, pp. 486–491, June 2002.

[17] S. Insup, P. Seungwhun, and S. Youngsoo, "Register allocation for high-level synthesis using dual supply voltages," in *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC '09)*, pp. 937–942, July 2009.

[18] S. P. Mohanty and E. Kougianos, "Simultaneous power fluctuation and average power minimization during nano-CMOS behavioral synthesis," in *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID '07)*, pp. 577–582, January 2007.

[19] F. Wang, X. Wu, and Y. Xie, "Variability-driven module selection with joint design time optimization and post-silicon tuning," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '08)*, pp. 2–9, March 2008.

[20] R. W. Keyes, "Physical limits in digital electronics," *Proceedings of the IEEE*, vol. 63, no. 5, pp. 740–767, 1975.

[21] D. S. Boning and S. Nassif, "Models of process variations in device and interconnect," in *Design of High Performance Microprocessor Circuits*, IEEE Press, 2000.

[22] B. Zhai, S. Hanson, D. Blaauw, and D. Sylvester, "Analysis and mitigation of variability in subthreshold design," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 20–25, August 2005.

[23] NCSU, "45 nm FreePDK," http://www.eda.ncsu.edu/wiki/FreePDK.

[24] M. Meterelliyoz, A. Goel, J. P. Kulkarni, and K. Roy, "Accurate characterization of random process variations using a robust

low-voltage high-sensitivity sensor featuring replica-bias circuit," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC '10)*, pp. 186–187, February 2010.

[25] C. Jacoboni and P. Lugli, *The Monte Carlo Method for Semiconductor Device Simulation*, Springer, 1990.

[26] N. S. Kim, T. Austin, D. Blaauw et al., "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–64, 2003.

[27] K. M. Cao, W. C. Lee, W. Liu et al., "BSIM4 gate leakage model including source-drain partition," in *Proceedings of the IEEE International Electron Devices Meeting*, pp. 815–818, December 2000.

[28] N. C. Beaulieu, A. A. Abu-Dayya, and P. J. McLane, "Comparison of methods of computing lognormal sum distributions and outages for digital wireless applications," in *Proceedings of the IEEE International Conference on Communications*, pp. 1270–1275, May 1994.

[29] S. H. Kulkarni, A. N. Srivastava, and D. Sylvester, "A new algorithm for improved VDD assignment in low power dual VDD systems," in *Proceedings of the 2004 International Symposium on Lower Power Electronics and Design (ISLPED '04)*, pp. 200–205, August 2004.

[30] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, "Matching properties of MOS transistors," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1433–1440, 1989.

[31] J. Kwong and A. P. Chandrakasan, "Variation-driven device sizing for minimum energy sub-threshold circuits," in *Proceedings of the 11th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '06)*, pp. 8–13, October 2006.

[32] S. A. Tawfik and V. Kursun, "Multi-Vth level conversion circuits for multi-VDD systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 1397–1400, May 2007.

[33] E. Aarts and J. K. Lenstra, *Local Search in Combinatorial Optimization*, Princeton University Press, 2003.

[34] A. Raghunathan and N. K. Jha, "Iterative improvement algorithm for low power data path synthesis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 597–602, November 1995.

[35] F. Ishihara, F. Sheikh, and B. Nikolić, "Level conversion for dual-supply systems," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 12, no. 2, pp. 185–195, 2004.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 691864, 24 pages
doi:10.1155/2012/691864

Research Article

Task-Level Data Model for Hardware Synthesis Based on Concurrent Collections

Jason Cong, Karthik Gururaj, Peng Zhang, and Yi Zou

Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90095, USA

Correspondence should be addressed to Karthik Gururaj, karthikg@cs.ucla.edu

Received 17 October 2011; Revised 30 December 2011; Accepted 11 January 2012

Academic Editor: Yuan Xie

The ever-increasing design complexity of modern digital systems makes it necessary to develop electronic system-level (ESL) methodologies with automation and optimization in the higher abstraction level. How the concurrency is modeled in the application specification plays a significant role in ESL design frameworks. The state-of-art concurrent specification models are not suitable for modeling task-level concurrent behavior for the hardware synthesis design flow. Based on the concurrent collection (CnC) model, which provides the maximum freedom of task rescheduling, we propose task-level data model (TLDM), targeted at the task-level optimization in hardware synthesis for data processing applications. Polyhedral models are embedded in TLDM for concise expression of task instances, array accesses, and dependencies. Examples are shown to illustrate the advantages of our TLDM specification compared to other widely used concurrency specifications.

1. Introduction

As electronic systems become increasingly complex, the motivation for raising the level of design abstraction to the electronic system level (ESL) increases. One of the biggest challenges in ESL design and optimization is that of efficiently exploiting and managing the concurrency of a large amount of parallel tasks and design components in the system. In most ESL methodologies, task-level concurrency specifications, as the starting point of the optimization flow, play a vital role in the final implementation quality of results (QoR). The concurrency specification encapsulates detailed behavior within the tasks, and explicitly specifies the coarse-grained parallelism and the communications between the tasks. System-level implementation and optimization can be performed directly from the system-level information of the application. Several ESL methodologies have been proposed previously. We refer the reader to [1, 2] for a comprehensive survey of the state-of-art ESL design flows.

High-level synthesis (HLS) is a driving force behind the ESL design automation. Modern HLS systems can generate register transaction-level (RTL) hardware specifications that come quite close to hand-generated designs [3] for synthesizing computation-intensive modules into a form of hardware accelerators with bus interfaces. At this time, it is not easy for the current HLS tools to handle task-level optimizations such as data transfer between accelerators, programmable cores, and memory hierarchies. The sequential C/C++ programming language has inherent limitations in specifying the task-level parallelism, while SystemC requires many implementation details, such as explicitly defined port/module structures. Both languages impose significant constraints on optimization flows and heavy burdens for algorithm/software designers. A tool-friendly and designer-friendly concurrency specification model is vital to the successful application of the automated ESL methodology in practical designs.

The topic of concurrency specification has been researched for several decades—from the very early days of computer science to today's research in parallel programming. From the ESL point of view, some of the results were too general and led to high implementation costs for general hardware structures [4, 5], and some results were too restricted and could only model a small set of applications [6, 7]. In addition, most of the previous models focused only on the description of the behavior or computation, but introduced redundant constraints unintentional for

implementation—such as the restrictive execution order of iterative task instances. CnC [8] proposed the concept of decoupling the algorithm specification with the implementation optimization; this provides the larger freedom of task scheduling optimization and a larger design space for potentially better implementation QoR. But CnC was originally designed for multicore processor-based platforms that contain a number of dynamic syntax elements in the specification—such as dynamic task instance generation, dynamic data allocation, and unbounded array index. A hardware-synthesis-oriented concurrency specification for both behavior correctness and optimization opportunities is needed by ESL methodologies to automatically optimize the design QoR.

In this paper we propose a task-level concurrency specification (TLDM) targeted at ESL hardware synthesis based on CnC. It has the following advantages.

(1) Allowing maximum degree of concurrency for task scheduling.

(2) Support for the integration of different module-level specifications.

(3) Support for mapping to heterogeneous computation platforms, including multicore CPUs, GPUs, and FPGAs (e.g., as described in [9]).

(4) Static and concise syntax for hardware synthesis.

The remainder of our paper is organized as follows. Section 2 briefly reviews previous work on concurrency specifications. Section 3 overviews the basic concepts in CnC. Section 4 presents the details of our TLDM specification. In Section 5 we illustrate the benefits of our TLDM specification with some concrete examples. Finally, we conclude our paper in Section 6 with a discussion of ongoing work.

2. Concurrent Specification Models

A variety of task-level concurrent specification models exist, and each concurrent specification model has its underlying model of computation (MoC) [10]. One class of these models is derived from precisely defined MoCs. Another class is derived from extensions of sequential programming languages (like SystemC [11]) or hardware description languages (like Bluespec System Verilog [12]) in which the underlying MoC has no precise or explicit definitions. These languages always have the ability to specify different MoCs at multiple abstraction levels. In this section we focus on the underlying MoCs in the concurrent specification models and ignore the specific languages that are used to textually express these MoCs.

The analyzability and expressibility of the concurrent specification model is determined by the underlying MoC [10]. Different MoCs define different aspects of task concurrency and implementation constraints for the applications. The intrinsic characteristics of each specific MoC are used to build the efficient synthesizer and optimizer for the MoC. The choice of the MoC greatly influences the applicable optimizations and the final implementation results as well. The key considerations in MoC selection are the following.

(i) *Application scope*: the range of applications that the MoC can model or efficiently model.

(ii) *Ease of use*: the effort required for a designer to specify the application using the MoC.

(iii) *Suitability for automated optimization*: while a highly generic MoC might be able to model a large class of applications with minimal user changes, it might be very difficult to develop efficient automatic synthesis flows for such models.

(iv) *Suitability for the target platform*: for example, a MoC, which implicitly assumes a shared memory architecture (such as CnC [8]), may not be well suited for synthesis onto an FPGA platform where support for an efficient shared memory system may not exist.

While most of these choices appear highly subjective, we list some characteristics that we believe are essential to an MoC under consideration for automated synthesis.

(i) *Deterministic execution*: unless the application domain/system being modeled is nondeterministic, the MoC should guarantee that, for a given input, execution proceeds in a deterministic manner. This makes it more convenient for the designer (and ESL tools) to verify correctness when generating different implementations.

(ii) *Hierarchy*: in general, applications are broken down into subtasks, and different users/teams could be involved in designing/implementing each subtask. The MoC should be powerful enough to model such applications in a hierarchical fashion. An MoC that supports only a flat specification would be difficult to work with because of the large design space available.

(iii) *Support of heterogeneous target platforms and refinement:* modern SoC platforms consist of a variety of components—general-purpose processor cores, custom hardware accelerators (implemented on ASICs or FPGAs), graphics processing units (GPUs), memory blocks, and interconnection fabric. While it may not be possible for a single MoC specification to efficiently map to different platforms, the MoC should provide directives to refine the application specification so that it can be mapped to the target platform. This also emphasizes the need for hierarchy because different subtasks might be suited to different components (e.g., FPGA versus GPUs); hence, the refinements could be specific to a subtask.

2.1. General Models of Computation. We start our review of previous work with the most general models of computation. These models impose minimum limitations in the specification and hence can be broadly applied to describe a large variety of applications.

Communicating sequential process (*CSP*) [4] allows the description of systems in terms of component processes that operate independently and interact with each other solely through message-passing communication. The relationships between different processes, and the way each process

communicates with its environment, are described using various process algebraic operators.

Hierarchy is supported in CSP where each individual process can itself be composed of subprocesses (whose interaction is modeled by the available operators). CSP allows processes to interact in a nondeterministic fashion with the environment; for example, the nondeterministic choice operator in a CSP specification allows a process to read a pair of events from the environment and decide its behavior based on the choice of one of the two events in a nondeterministic fashion.

One of the key applications of the CSP specification is the verification of large-scale parallel applications. It can be applied to detect deadlocks and livelocks between the concurrent processes. Examples of tools that use CSP to perform such verification include FDR2 [13] and ARC [14]. Verification is performed through a combination of CSP model refinement and CSP simulation. CSP is also used for software architecture description in a Wright ADL project [15] to check system consistency; this approach is similar to FDR2 and ARC.

Petri nets [16] consist of places that hold tokens (tokens represent input or output data) and transitions that describe the process of consuming and producing tokens (transitions are similar to processes in CSP). A transition is enabled when the number of tokens on each input arc is greater than or equal to the required number of input tokens. When multiple transitions are enabled at the same time, any one of them can fire in a nondeterministic way; also, a transition need not fire even if it is enabled. Extensions were proposed to the Petri net model to support hierarchy [17]. Petri nets are used for modeling distributed systems—the main aim being to determine whether a given system can reach any one of the user-specified erroneous states (starting from some initial state).

Event-Driven Model (EDM). The execution of concurrent processes in EDM is triggered by a series of events. The events could be generated by the environment (system inputs) or processes within the system. This is an extremely general model for specifying concurrent computation and can, in fact, represent many specific models of computation [18]. This general model can easily support hierarchical specification but cannot provide deterministic execution in general.

Metropolis [18] is a modeling and simulation environment for platform-based designs that uses the event-driven execution model for functionally describing application/computation. Implementation platform modeling is also provided by users as an input to Metropolis, from which the tool can perform synthesis, simulation, and design refinement.

Transaction-Level Models (TLMs). In [19] the authors define six kinds of TLMs; however, the common point of all TLMs is the separation of communication and computation. The kinds of TLMs differ in the level of detail specified for the different computation and communication components. The specification of computation/communication could be cycle-accurate, approximately timed or untimed, purely functional, or implementation specific (e.g., a bus for communication).

The main concern with using general models for hardware synthesis is that the models may not be suitable for analysis and optimization by the synthesis tool. This could lead to conservative implementations that may be inefficient.

2.2. Process Networks. A process network is the abstract model in most graphical programming environments, where the nodes of the graph can be viewed as processes that run concurrently and exchange data over the arcs of the graph. Processes and their interactions in process networks are much more constrained than those of CSP. Determinism is achieved by two restrictions: (1) each process is specified as a deterministic program, and (2) the quantity and the order of data transfers for each process are statically defined. Process networks are widely used to model the data flow characteristics in data-intensive applications, such as signal processing.

We address three representative process network MoCs (KPN, DPN, and SDF). They differ in the way that they specify the execution/firing and data transfer; this brings differences in determining the scheduling and communication channel size.

2.2.1. Kahn Process Network (KPN). KPN [20] defines a set of sequential processes communicating through unbounded first-in-first-out (FIFO) channels. Writes to FIFOs are nonblocking (since the FIFOs are unbounded), and reads from FIFOs are blocking—which means the process will be blocked when it reads an empty FIFO channel. Peeking into FIFOs is not allowed under the classic KPN model. Applications modeled as KPN are deterministic: for a given input sequence, the output sequence is independent of the timing of the individual processes.

Data communication is specified in the process program in terms of channel FIFO reads and writes. The access patterns of data transfers can, in general, be data dependent and dynamically determined in runtime. It is hard to statically analyze the access patterns and optimize the process scheduling based on them. A FIFO-based self-timed dynamic scheduling is always adopted in KPN-based design flows, where the timing of the process execution is determined by the FIFO status.

Daedulus [21] provides a rich framework for exploration and synthesis of MPSoC systems that use KPNs as a model of computation. In addition to downstream synthesis and exploration tools, Daedulus provides a tool called KPNGen [21], which takes as input a sequential C program consisting of affine loop nests and generates the KPN representation of the application.

2.2.2. Data Flow Process Network (DPN). The dataflow process network [5] is a special case of KPN. In dataflow process networks, each process consists of repeated "firings" of a dataflow "actor." An actor defines a (often functional)

quantum of computation. Actors are assumed to fire (execute atomically) when a certain finite number of input tokens are available on each input edge (arc). A firing is defined as consuming a certain number of input tokens and producing a certain number of output tokens. The firing condition for each actor and the tokens consumed/produced during the firing are specified by a set of firing rules that can be tested in a predefined order using only blocking read. The mechanism of actor firings helps to reduce the overhead of context switching in multicore platforms because the synchronization occurs only at the boundary of the firing, not within the firings. But compared to KPN, this benefit does not help much in hardware synthesis.

2.2.3. Synchronous Data Flow Graph (SDF). SDF [7] is a more restricted MoC than DPN, in which the number of tokens that can be consumed/produced by each firing of a node is fixed statically. The fixed data rate feature can be used to efficiently synthesize the cyclic scheduling of the firings at the compile time. Algorithms have been developed to statically schedule SDFs (such as [7]). An additional benefit is that for certain kinds of SDFs (satisfying a mathematical condition based on the number of tokens produced/consumed by each node), the maximum buffer size needed can be determined statically [7].

StreamIt [22] is a programming language and a compilation infrastructure that uses the SDF MoC for modeling real streaming applications. An overall implementation and optimization framework is built to map from SDF-based specifications of large streaming applications to various general-purpose architectures such as uniprocessors, multicore architectures, and clusters of workstations. Enhancements proposed to the classic SDF model include split and join nodes that model data parallelism of the actors in the implementations. In general, StreamIt has the same application expressiveness with SDF.

KPN, DPN, and SDF are suitable for modeling the concurrency in data processing applications, in which execution is determined by the data availability. The possibility of data streaming (pipelining) is intrinsically modeled by the FIFO-based communication. However, these models are not user-friendly enough as a preferred concurrency specification for data processing applications, because users need to change the original shared-memory-based coding style into a FIFO-based one. In addition, these models are not tool-friendly enough in the sense that (1) data reuse and buffer space reuse are relatively harder to perform in this overconstrained FIFO-based communication, (2) no concise expressions for iterative task instances (which perform identical behavior on different data sets) are embedded in one sequential process with a fixed and overconstrained execution order, and (3) it is hard to model access conflicts in shared resources, such as off-chip memory, which is an essential common problem in ESL design.

2.3. Parallel Finite State Machines (FSMs). FSMs are mainly used to describe applications/computations that are control-intensive. Extensions have been proposed to the classic FSM model to support concurrent and communicating FSMs, such as StateChart [6] and Codesign FSM (CFSM) [23]. Figure 1 shows that, in a hierarchical FSM, separate states and their transitions can be grouped into a hierarchical state D, and state D encapsulates the details of state B and C. Similarly, Figure 2 shows a concurrent FSM (Figure 2(a)) and one transition of its combined FSM (Figure 2(b)). A state of combined FSM is a combination of the states of the concurrent FSMs. The Koski synthesis flow [24] for ESL synthesis of MPSoC platforms uses the StateChart model for describing functionality of the system. All the state changes in the StateChart are assumed to be synchronous, which means, in the cases of transition from AX to BY, conditions p and q are validated simultaneously, and there is no intermediate state like AY or BX during the transition. However, in real systems the components being modeled as FSMs can make state changes at different times (asynchronously). CFSM breaks the limitation and introduces asynchronous communication between multiple FSMs. The Polis system [25] uses the CFSM model to represent applications for hardware-software codesign, synthesis, and simulation.

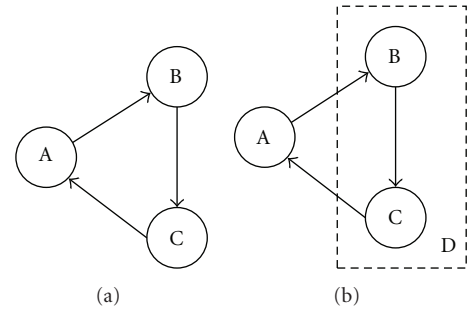


Figure 1: Hierarchical FSMs.

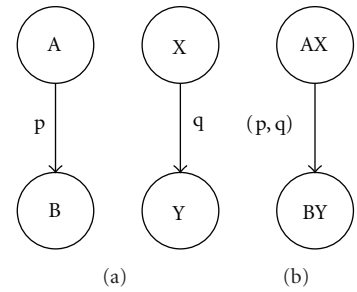


Figure 2: Parallel FSMs.

For a real application with both control and data flow, it is possible to embed the control branches into the module/actor specification of a data streaming model. For example, as mentioned in [26], additional tokens can be inserted into the SDF specification to represent global state and irregular (conditional) actor behavior. However, this would lose the opportunities to optimize the control path at system level, which may lead to a larger buffer size and more write operations to the buffer.

There is a series of hybrid specifications containing both FSM and data flow. *Finite state machine with datapath (FSMD)* models the synchronized data paths and FSM transitions at clock cycle level. *Synchronous piggyback dataflow networks* [26] introduce a global state table into the conventional SDF to support conditional execution. This model of computation was used by the PeaCE ESL synthesis framework [27], which, in turn, is an extension of the Ptolemy framework [19].

The *FunState* model [28] combines dynamic data flow graphs with a hierarchical and parallel FSM similar to StateCharts. Each transition in the FSM may be equivalent to the firing of a process (or function) in the network. The condition for a transition is a Boolean function of the number of tokens in the channels connecting the processes in the network. The System Co-Designer tool for mapping applications to MPSoC platforms uses the FunState model; the mapped implementation result is represented as a TLM (transaction-level model).

Program state machines (PSMs) [29] use the concept of hierarchical and concurrent FSMDs and extend it by replacing the cycle-based timing model with a general discrete timing model where the program can execute for arbitrary amount of time, and state transitions occur when the execution of the program is completed. In the SCE (system-on-chip environment) design flow [30], the SpecC [31] language specifies this particular model of computation. SpecC also provides primitives to specify FSM, parallelism, pipelining, and hierarchy of components. Using a target platform database (bus-based MPSoCs and custom IPs), the SCE flow generates TLMs of the target system, as well as hardware-software implementation for deployment.

2.4. Parallel Programming Languages. OpenMP (Open Multi-Processing) is an extension of C/C++ and Fortran languages to support multithread parallelism on a shared memory platform. A thread is a series of instructions executed consecutively. The OpenMP program starts execution from a master thread. The code segments that are to be run in parallel are marked with preprocessor directives (such as #pragma omp parallel). When the master thread comes to a parallel directive, it forks a specific number of slave threads. After the execution of the parallelized code, the slave threads join back the master thread. Both task parallelism and data parallelism can be specified in OpenMP. A global memory space is shared by all the threads, and synchronization mechanisms between threads are supported to avoid race conditions.

MPI is a set of APIs standardized for programmers to write portable message-passing programs in C and Fortran. Instead of preprocessing directives, MPI uses explicit API calling to start and stop the parallelization of the threads. Data transfers between parallelized threads are through message-passing using API function calls. Blocking access is supported in MPI to perform synchronization between threads.

Cilk [32] is another multithreaded language for parallel programming that proposes extensions to the C language for parallel processing. Cilk introduces the *spawn* construct to launch computations that can run in parallel with the thread that invokes spawn and the *sync* construct which makes the invoking thread wait for all the spawned computations to complete and return. The Cilk implementation also involves a runtime manager that decides how the computations generated by spawn operations are assigned to different threads.

Habanero-Java/C [33, 34] includes a set of task parallel programming constructs, in a form of the extensions to standard Java/C programs, to take advantage of today's multicore and heterogeneous architectures. Habanero-Java/C has two basic primitives: async and finish. The async statement, async <stmt>, causes the parent task to fork a new child task that executes <stmt> (<stmt> can be a signal statement or a basic block). Execution of the async statement returns immediately. The finish statement, finish <stmt>, performs a join operation that causes the parent task to execute <stmt> and then wait until all the tasks created within <stmt> have terminated (including transitively spawned tasks). Compared to previous languages (like Cilk), more flexible structures of task forking and joining are supported in Habanero-Java/C because the fork and join can happen in arbitrary function hierarchies. Habanero-Java/C also defines specific programming structures such as *phasers* for synchronization and *hierarchical place trees* (HPTs) for hardware placement locality.

However, all these parallel programming language extensions were originally designed for high-performance software development on multicore processors or distributed computers. They have some intrinsic obstacles when specifying a synthesized hardware system in ESL design: (1) general runtime routines performing task creation, synchronization, and dynamic scheduling are hard to implement in hardware; (2) the dynamic feature of task creation makes it hard to analyze the hardware resource utilization at synthesis time.

3. Review of Concurrent Collections (CnCs)

The CnC [8] was developed for the purpose of separating the implementation details for implementation tuning experts from the behavior details for application domain experts—which provides both tool-friendly and user-friendly concurrency specifications. The iterations of iterative tasks are defined explicitly and concisely, while the model-level details are encapsulated within the task body. Although most of these concurrent specifications target general-purpose multicore platforms, the concepts can also be used for the task-level behavior specification of hardware systems.

3.1. Basic Concepts and Definitions. A behavior specification of the application for hardware synthesis can be considered as a logical or algorithmic mapping function from the input data to the output data. The purpose of hardware synthesis is to map the computation and storage in the behavior specification into temporal (by scheduling) and spatial (by binding/allocation) implementation design spaces for performance, cost, and power optimization.

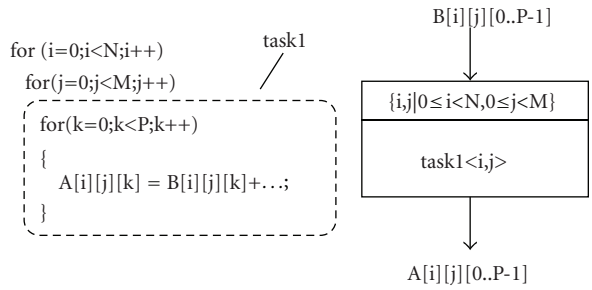


Figure 3: Concurrent task modeling.

In general, higher-level optimization has a larger design space and may lead to better potential results. To raise the abstraction level of the optimization flow, CnC uses steps as the basic units to specify the system-level behavior of an application. A step is defined as a statically determined mapping function from the values of an input data set to those of an output data set, in which the same input data will generate the same output data regardless of the timing of input and output data. A step is supposed to execute multiple times to process the different sets of data. Each execution of the step is defined as a step instance and is invoked by a control tag. Each control tag has an integer vector that is used to identify or index the instance of the step. An iteration domain is the set of all the iterator vectors corresponding to the step, representing all the step instances. The step instance of step t indexed by iterators (i, j) is notated as t<i,j> or t:i,j. As shown in Figure 3, we encapsulate the loop k into the task1, and task1 will execute N×M times according to the loop iterations indexed by variables i and j. Task1 has an iteration domain of {i, j | 0≤i<N, 0≤j<M}. Each iterator vector (i, j) in the iteration domain represents a step instance task1<i,j>. Compared to the explicit order of loops and loop iterations imposed in the sequential programming languages, no overconstrained order is defined between steps and step instances if there is no data dependence. In the concurrent specification, an application includes a collection of steps instead of a sequence of steps, and each step consists of a collection of step instances instead of a sequence of step instances.

A data item is, in general, defined as a multidimensional array used for the communication between steps and step instances. The array elements are indexed by data tags, each data tag represents the subscript vector of an array element. The data domain of an array defines the set of available subscript vectors. For an access reference of a multidimensional array A, its data tag is notated as $(A_0, A_1, A_2, \ldots)$. For example, the subscript vector of A[i][j][k] is (i, j, k), where A_0= i, A_1= j and A_2= k. Each step instance will access one or more of the elements in the arrays. A data access is defined as a statically determined mapping from the step instance iterator vector to the subscript vectors of the array elements that the task instance reads from or writes to. The input data access of task1, B[i][j][0...(P − 1)] in Figure 3, is the mapping {<i,j> → B[B_0][B_1][B_2] | B_0=i, B_1=j}. In this access, no constraints are placed on the subscript B_2, which means any data elements with the same B_0 and B_1 but different B_2 will be accessed in one step instance, task1<B_0, B_1 >. From the iteration domain and I/O accesses of a task, we can derive the set of data elements accessed by all the instances of the step.

Dynamic single assignment (DSA) is a constraint of the specification on data accesses. DSA requires that each data element only be written once during the execution of the application. Under the DSA constraint, an array element can hold only one data value, and memory reuse for multiple liveness-nonoverlap data is forbidden. CnC adopts the DSA constraint in its specification to avoid the conflicts of the concurrent accesses into one array element and thus provide the intrinsic determinism of the execution model.

Dependence represents the execution precedence constraints between step instances. If one step generates data that are used in another step, the dependence is implicitly specified by the I/O access functions of the same data object in the two steps. Dependence can also be used to describe any kind of step instance precedence constraints that may guide the synthesizer to generate correct and efficient implementations. For example, to explicitly constrain that the outmost loop i in Figure 3 is to be scheduled sequentially, we can specify the dependence like {task1<i, *> → task1<i+1, *>}, a concise form for $\forall j_1, j_2$, task1<i,j_1 > → task1<i+1,j_2 >. From the iteration domains and dependence mapping of two steps (the two steps are the same for self-dependence), we can derive the set of precedence constraints related to the instances of the two steps.

The execution of the concurrent steps is enabled when the two conditions are satisfied: (1) the control tag is generated (by primary input or by the execution of steps); (2) all the input data corresponding to the step instance is generated (by primary input or by the execution of steps). In Intel CnC, the iteration domains (control tag collections) of the steps are not statically defined and control tags are generated dynamically during the execution. An enabled step instance is not necessary in order to execute immediately. A runtime scheduler can dynamically schedule an enabled task instance at any time to optimize different implementation metrics.

3.2. Benefits and Limitations of CnC. There are some properties that the other MoCs have in common with CnC—support for hierarchy, deterministic execution, and specification of both control and data flow. However, there are a few points where CnC is quite distinct from the other concurrent specification models.

While most specification models expect the user to explicitly specify parallelism, CnC allows users to specify dependences, and the synthesizer or the runtime decides when and which step instances to schedule in parallel. Thus, parallelism is implicit and dynamic in the description. The dynamic nature of parallelism has the benefit of platform independency. If an application specified in CnC has to run on two different systems—an 8-core system (multicore CPU) and a 256-core system (GPU-like)—then the specification need not change to take into account the difference in the number of cores. The runtime would decide how many step

instances should be executed in parallel on the two systems. However, for the other MoCs, the specification may need to be changed in order to efficiently use the two systems.

Another benefit of using CnC is that since the dependence between the step and item collections is explicit in the model, it allows the compiler/synthesis tool/runtime manager to decide the best schedule. An example could be rescheduling the order of the step instances to optimize for data locality. Other MoCs do not contain such dependence information in an explicit fashion; the ordering of different node executions/firings is decided by the user and could be hidden from any downstream compiler/synthesis tool/runtime manager.

However, CnC was originally developed for general-purpose multicore platforms. Some issues need to be solved in order for CnC to become a good specification model for task-level hardware synthesis. First, dynamic semantics, such as step instance generation, make it hard to manage the task scheduling without a complex runtime kernel. This leads to a large implementation overhead in hardware platforms such as FPGA. Second, memory space for data items is not specified, which may imply unbounded memory size because the data item tags are associated with the dynamically generated control tags. Third, design experience shows that DSA constraints cause many inconveniences in algorithm specifications, and this makes CnC user-unfriendly. Fourth, currently there is not a stable version of CnC that formally supports hierarchy in the specification. As for those limitations of CnC as a task-level concurrency specification model for hardware synthesis, we have proposed our TLDM based on CnC and adapted it to hardware platforms.

4. Task-Level Data Model

In this section a task-level concurrency specification model is proposed based on the Intel CnC. We introduce a series of adaptations of Intel CnC targeted at task-level hardware system synthesis. We first overview the correspondence and differences between TLDM and CnC. Then our TLDM specification is defined in detail in a C++ form, and classes and fields are defined to model the high-level information used for task scheduling. While the C++ classes are used as the in-memory representation in the automation flow, we also define a text-based format for users to specify TLDM directly. These two forms of specifications have the equivalent semantics. An example application is specified using our TLDM specification. We also make a detailed comparison of the proposed TLDM with the CnC specification to demonstrate that the proposed TLDM is more suitable for hardware synthesis.

4.1. Overview of TLDM. Our TLDM specification model inherits the benefits of CnC and is customized to hardware synthesis. Table 1 summarizes the similarities and differences between TLDM and CnC. TLDM inherits most of the syntax and semantics elements from CnC. The counterparts for step, data, control tag, and data tag in CnC are, in TLDM, task, data, iterator vector and subscript vector, respectively.

```
class tldm_app {
  set<tldm_task *>        task_set;
  set<tldm_data *>        data_set;
  set<tldm_access *>      access_set;
  set<tldm_dependence *>dependence_set;
};
```

Listing 1

TLDM removes the syntax of the control item in CnC to avoid the dynamic behavior in task instantiation. In CnC a step instance is enabled when its control item is generated and all its input data is ready; in TLDM a task instance is enabled when its input data is ready. Control items may be transformed into data items to model the precedence relation between corresponding task instances. In TLDM, iteration domain, data domain, and data (input/output) accesses are specified explicitly and statically. According to the execution model of CnC, the dependence between steps is implied by the tag generation and data access statements in the step functions. TLDM provides syntax to explicitly specify the dependence between task instances in the task-level specification. DSA restriction is not formally enforced in the TLDM specification. Programmers need to ensure data coherence and program determinism by imposing dependence constraints of task instances in the case that DSA restriction is broken.

4.2. A C++ Specification for TLDM. A TLDM application consists of a task set, a data set, an access set, and a dependence set (See Listing 1).

The task set specifies all the tasks and their instances in a compact form. One task describes the iterative executions of the same functionality with different input/output data. The instances of one task are indexed by the iterator vectors. Each component of the iterator vector is one dimension of the iterator space, which can be considered as one loop level surrounding the task body in C/C++ language. The iteration domain defines the range of the iteration vector, and each element in the iteration domain corresponds to an execution instance of the task. The input and output accesses in each task instance are specified by affine functions of the iterator vector of the task instance. The access functions are defined in the *tldm_access* class.

Class members *parent* and *children* are used to specify the hierarchy of the tasks. The task hierarchy supported by our TLDM can provide the flexibility to select the task granularity in our optimization flow. A coarse-grained low-complexity optimization flow can help to determine which parts are critical for some specific design target, and fine-grained optimization can further optimize the subtasks locally with a more precise local design target.

A pointer to the details of a task body (task functionality) is kept in our model to obtain certain information (such as module implementation candidates) during the task-level synthesis. We do not define the concrete form to specify the

Table 1: Syntax and semantics summary of CnC and TLDM.

	CnC	TLDM
Computation	Step	Task
Communications	Data item	Data
	Control item	Not Supported
Index of set	Control item tag	Iterator vector
	Data item tag	Subscript vector
Execution model	(1) Control item is generated	Input data are ready
	(2) Input data are ready	
Iteration and data domain	Unbounded and dynamically determined	Bounded and statically specified
Data accesses	Embedded in step body	Specified in task definition
Dependence	Implicitly specified	Can be explicitly specified
DSA	Enforced	Not enforced

```
class tldm_task {
    string                    task_id;
    tldm_iteration_domain*    domain;
    vector<tldm_access *>     io_access; // input and output data accesses
    tldm_task *               parent;
    vector<tldm_task *>       children;
    tldm_ task_body*          body;
};
```

Listing 2

task body in our TLDM specification. A task body can be explicitly specified as a C/C++ task function, or a pointer to the in-memory object in an intermediate representation such as a basic block or a statement, or even the RTL implementations (See Listing 2).

An iteration domain specifies the range of the iterator vectors for a task. We consider the boundaries of iterators in four cases. In the first simple case, the boundaries of all the iterators are determined by constants or precalculated parameters that are independent of the execution of the current task. In the second case, boundaries of the inner iterator are in the form of a linear combination of the outer iterators and parameters (such as a triangular loop structure). The iteration domain of the first two cases can be modeled directly by a polyhedral model in a linear matrix form as *affine_iterator_range*. In the third case, the boundaries of the inner iterators are in a nonlinear form of the outer iterators and parameters. By considering the nonlinear term of outer iterators as pseudoparameters for the inner iterators, we can also handle the third case in a linear matrix form by introducing separate pseudoparameters in the linear matrix form. In the fourth and most complex case, the iterator boundary is determined by some local data-dependent variables varying in each instance of the task. For example, in an iterative algorithm, a data-dependent convergence condition needs to be checked in each iteration. In TLDM, we separate the iterators into data independent (first three cases) and data-dependent (the fourth case). We model data-independent iterators in a polyhedral matrix form (class *polyhedral_set* is described below in this subsection); we model data dependent iterators in a general form as a TLDM expression, which specifies the execution condition for the task instance. The execution conditions of multiple data-dependent iterators are merged into one TLDM expression in a binary-tree form (See Listing 3).

The data set specifies all the data storage elements in the dataflow between different tasks or different instances of the same tasks. Each *tldm_data* object is a multidimensional array variable or just a scalar variable in the application. Each data object has its member *scope* to specify in which task hierarchy the data object is available. In other words, the data object is accessible only by the tasks within its scope hierarchy. If the data object is global, its scope is set to be NULL. The boundaries (or sizes) of a multidimensional array are predefined constants and modeled by a polyhedral matrix *subscript_range*. The detailed form of *subscript_range* is similar to that of *affine_iterator_range* in the iteration domain. Accesses out of the array bound are forbidden in

```
class tldm_iteration_domain{
    vector<tldm_data *>     iterators;
    polyhedral_set          affine_iterator_range;
    tldm_expression*        execute_condition;
};
class tldm_expression {
    tldm_data*              iterator; // the iterator to check the expression
    int                     n_operator;
    tldm_expression*        left_operand;
    tldm_expression*        right_operand;
    tldm_data*              leaf;
};
```

Listing 3

```
class tldm_data {
    string                  data_id;
    tldm_task*              scope;
    int                     n_dimension; // number of dimensions
    polyhedral_set          subscript_range; // ranges in each dimensions
    tldm_data_body*         body;
};
```

Listing 4

our TLDM execution model. A body pointer for data objects is also kept to refer to the array definition in the detailed specification (See Listing 4).

The access set specifies all the data access mappings from task instances to data elements. Array data accesses are modeled as a mapping from the iteration domain to the data domain. If the mapping is in an affine form, it can be modeled by a polyhedral matrix. Otherwise, we assume that possible ranges of the nonaffine accesses (such as indirect access) are bounded. We model the possible range of each nonaffine access by its affine (or rectangular) hull in the multidimensional array space, which can also be expressed as a polyhedral matrix (class *polyhedral_set*). A body pointer (*tldm_access_body* *) for an access object is kept to refer to the access reference in the detailed specification (See Listing 5).

The dependence set specifies timing precedence relations between task instances of the same or different tasks. A dependence relation from (task0, instance0) to (task1, instance1) imposes a constraint in task scheduling that (task0, instance0) must be executed before (task1, instance1). The TLDM specification supports explicit dependence imposed by specified *tldm_dependence* objects and implicit dependence embedded in the data access objects. Implicit dependence can be analyzed statically by a compiler or optimizer to generate derived *tldm_dependence* objects. The explicit dependence specification provides the designer with the flexibility to add user-specified dependence to help the compiler deal with complex array indices. User-specified dependence is also a key factor in relieving designers from the limitations of dynamic single assignment, while maintaining the program semantics. The *access0* and *access*1 fields point to the corresponding *tldm_access* objects and can be optional for the user-specified dependence. In most practical cases, the dependence between task instances can be modeled by affine constraints of corresponding iterators of the two dependent tasks as *dependent_relation*. If the dependence relation between the two iterators is not affine, either a segmented affine form or an affine hull can be used to specify the dependence in an approximate way (See Listing 6).

A polyhedral_set object specifies a series of linear constraints of scalar variables in the data set of the application. The scalar variables can be the iterators (including loop iterators and data subscripts) and the loop-independent parameters. Each constraint is a linear inequality or equality of these scalar variables, and is modeled as an integer vector consisting of the linear coefficients for the scalar variables and a constant term and an inequality/equality flag. Multiple constraints form an integer matrix (See Listing 7).

Figure 4 shows examples of polyhedral matrices for iteration domain, data domain, data access, and task dependence, respectively. The first row of the matrix represents the list of iterators and parameters (*variables*), where $A0$ and $A1$ are the dimensions of array A and the # and $ columns are the constant terms and inequality flags, respectively. The following rows of the matrices represent linear constraints (*polyhedral_matrix*), where the elements of the matrices are the linear coefficients of the variables. For example, the second constraint of the iteration domain case is -i+0j+N-1≥0 (i≤N-1).

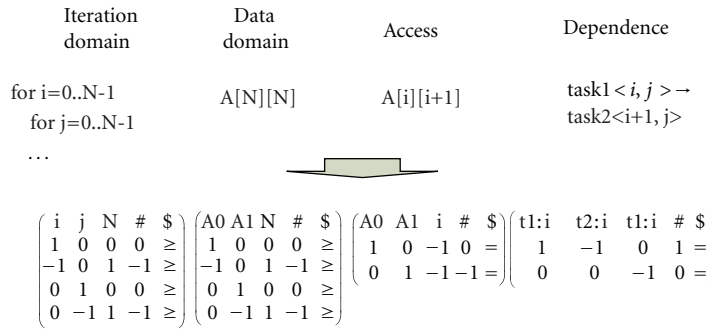


Figure 4: Polyhedral matrix representations.

```
class tldm_access {
    tldm_data*          data_ref; // the data accessed
    bool                is_write; // write or read access
    polyhedral_set      iterators_to_subscripts; // access range
    tldm_access_body*   body;
};
```

Listing 5

```
class tldm_dependence {
    tldm_task*          task0;
    tldm_task*          task1;
    tldm_access*        access0; //optional
    tldm_access*        access1; //optional
    polyhedral_set      iterator_relation;
};
```

Listing 6

```
class polyhedral_set {
    int                         n_iterator;
    int                         n_parameter;
    vector<tldm_data *>         variables;
    vector<vector<int≫          polyhedral_matrix;
};
```

Listing 7

4.3. A Textual Specification for TLDM. Similar to CnC, we use round, square, and angle brackets to specify tasks, data, and domains, respectively. Domain items can be iterator domains for tasks, or subscript domains for data items. Colons are used to specify an instance of these collections (See Listing 8).

Domains define the range constraints of iterator vectors and subscript vectors. For task instances that have variable iteration boundaries, some data items can be used as parameters in specifying the iteration range by arrow notations. Conditional iteration such as convergence testing can be specified by the keyword *cond*(…). Domains are associated with the corresponding data objects or tasks by double-colons (See Listing 9).

```
(task)
[data]
<domain>
(task_instance : iterator_vector)
[data_instance : subscript_vector]
<domain_instance: iterator_vector or subscript_vector>

e.g.
(task1)                 // A task named "task1"
<data0 : i, j, k>       // data element data0[i][j][k]
```

Listing 8

Input and output data accesses of task instance are defined by arrow operators. A range of data elements can be specified in a concise form by double-dot marks (See Listing 10).

Explicit dependence can also be specified by arrow operators. Dependence defines the relation between task instances (See Listing 11).

The body of a task can be defined in the statement where the iterator domain is associated with the task by using a brace bracket. A keyword *body_id*(…) is used to link to the detailed module-level information for the task. Task hierarchy can be defined by embedding the subtask

```
[parameter_list] -> <data_domain : subscript_vector>
{subscript_vector range};
[parameter_list] -> <iterator_domain : iterator_vector> {iterator_vector range};

<data_domain> :: [type data_name];
<iterator_domain> :: (task_name);

e.g.
// A[100][50]
<A_dom : A0, A1> {0<=A0; A0<100; 0<=A1; A1<50;};
<A_dom> : [double A];

// for (i=0; i<p; i++) task1(i);
[p] -><task1_dom : i> {0<=i; i<p;};
<task1_dom> : (task1);

// while (res > 0.1) {res = task2();}
[res] -> <task2_dom : t>  {cond(res > 0.1)};
<task2_dom> : (task2);
```

Listing 9

```
input_data_list -> (task_name : iterator_vector) -> output_data_list;

e.g.
// A[i][j], B[i][j] -> task1<i,j> -> C[j][i]
[A : i, j], [B : i, j] -> (task1 : i, j) -> [C : i, j]

//A[i][0],. . .,A[i][N]-> task2<i> -> B[2* i + 1]
[A : i, 0..N] -> (task1 : i) -> [B : 2* i + 1]
```

Listing 10

definitions into the body of the parent task. Data objects can also be defined in the task bodies to become local data objects (See Listing 12).

4.4. Examples of TLDM Modeling. Tiled Cholesky is an example that is provided by Intel CnC distribution [35]. Listings 13, 14 and 15 show the sequential C-code specification and TLDM modeling of the tiled Cholesky example. In this program we assume that tiling factor p is a constant.

The TLDM built from the tiled Cholesky example is shown in Listing 14. The data set, task set, iteration domain, and access set are modeled in both C++ and textual TLDM specifications.

Listings 16 and 17 show how our TLDM models the nonaffine iteration boundaries. The nonaffine form of outer loop iterators and loop-independent parameters are modeled as a new pseudo-parameter. The pseudo-parameter nonAffine(i) in Listing 17 is embedded in the polyhedral model of the iteration domain. The outer loop iterators (i) are associated with the pseudo-parameter as an input variable. In this way we retain the possibility of parallelizing the loops with nonaffined bounded iterators and keep the overall specification as a linear form.

For the convergence-based algorithms shown in Listings 18 and 19, loop iteration instance is not originally described as a range of iterators. An additional iterator (t) is introduced to the "while" loop to distinguish the iterative instances of the task. If we do not want to impose an upper bound for the iterator t, an unbounded iterator t is supported as in Listing 19. A *tldm_expression* exe_condition is built to model the execution condition of the "while" loop, which is a logic expression of *tldm_data* convergence_data. Conceptually, the instances of the "while" loop are possible to execute simultaneously in the concurrent specification. CnC imposes dynamic single-assignment (DSA) restrictions to avoid nondeterministic results. But DSA restriction requires multiple duplications of the iteratively updated data in the convergence iterations, which leads to a large or even unbounded array capacity. Instead of the DSA restriction in the specification, we support the user-specified dependence on reasonably constrained scheduling of the "while" loop instances to avoid the data access conflicts. Because the convergence condition will be updated and checked in each "while" loop iteration, the execution of the "while" loop must be done in a sequential way. So we add dependence from task1<t> to task1<t+1> for each t as in Listing 17. By removing the DSA restriction in the specification, we do not need multiple duplications of the iterative data, such as convergence or any other internal updating arrays.

```
(task_name0 : iterator_vector) -> (task_name1 : iterator_vector

e.g.
// task1<i,j> -> task2<j,i>
(task1 : i, j) -> (task2 : j, i)

// task1<i-1, 0>,...,task1<i-1, N> -> task1<i, 0>,...,task1<i, N>
(task1 : i-1, 0..N) -> (task1 : i, 0..N)
```

Listing 11

```
<iterator_domain0> :: (task_name)
{
  <data_domain> :: [type local_data_item];

  <iterator_domain1> :: (sub_task_name1)
  {
    // leaf node
    body_id(task_id1);
  };
  <iterator_domain2> :: (sub_task_name2)
  {
    // leaf node
    body_id(task_id2);
  };
  // access and dependence specification
};
```

Listing 12

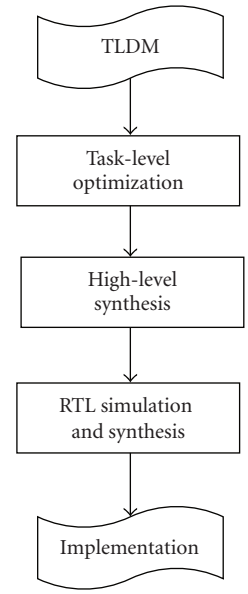


Figure 5: TLDM-based design flow.

Listings 20 and 21 show an indirect access example. Many nonaffine array accesses have their ranges of possible data units. For example, in the case of Listing 20, the indirect access has a preknown range (x<= idx[x] <= x+M). We can conservatively model the affine or rectangular hull of the possible access range in *tldm_access* objects. In Listing 21, the first subscript of the array_A is not a specific value related to the iterators, but a specific affine range of the iterators (j <= A_0 < j+M).

4.5. Demonstrative Design Flow Using TLDM. The detailed design optimizations from TLDM specification are beyond the scope of this paper. We show a simple TLDM-based design flow in Figure 5 and demonstrate how buffer size is reduced by task instance rescheduling as an example of task-level optimizations using TLDM. A task-level optimization flow determines the task-level parallelism and pipelining between the task instances, order of the instances of a task, and the interconnection buffers between the tasks. After task-level optimization, parallel hardware modules are generated automatically by high-level synthesis tools. RTL codes for both the hardware modules and their interconnections are integrated into a system RTL specification and then synthesized into the implementation netlist.

Different task instance scheduling will greatly affect the buffer size between the tasks. Instead of a fixed execution order of task instances in sequential program language, our TLDM can efficiently support selection of the proper execution order for task instance to reduce the buffer requirement between tasks. Listing 22 shows the example program to be optimized.

The TLDM specification for this example is shown in Figure 6. Two tasks are generated with their iteration domains. Four affine accesses to array A are notated in the TLDM graph. Three sets of data dependence are notated in the TLDM graph: two intratask dependences (dep0 and dep1), and one intertask dependence (dep2).

To evaluate the buffer size needed to transfer temporary data from task t0 to t1, we need to analyze the data element access pattern of the two tasks. As the execution order defined in the sequential C program, Figures 7(a) and 7(b) show the access order of the data t0 writes and t1 reads, respectively. A_0 and A_1 are array subscripts in the reference (like $A[A_0][A_1]$). The direction of the intratask data dependence is also shown in the two subgraphs. As we can see in the two illustrations, the access orders are not matched: one is row by row and the other is column by column. To maintain all the active data between the two tasks, an N×N buffer is needed.

```
1 int i, j, k;
2 data_type A[p][p][p+1];
3 for (k=0; k<p; k++) {
4     seqCholesky (A[k][k][k+1] ←A[k][k][k]);
5     for (j=k+1; j<p; j++) {
6         TriSolve(A[j][k][k+1] ←A[j][k][k], A[k][k][k+1]);
7         for (i=k+1; i<=j; i++) {
8             Update (A[j][i][k+1] ←A[j][k][k+1], A[i][k][k+1]);
9         }
10    }
11}
```

Listing 13: Example C-code of tiled Cholesky.

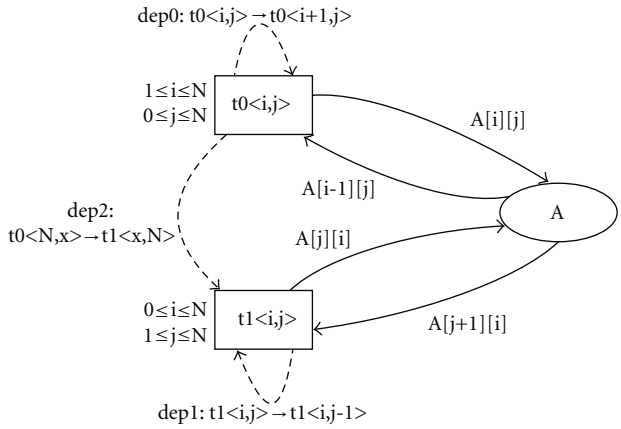


Figure 6: TLDM of the example in Listing 22.

In our TLDM specification, no redundant access order constraints are imposed, and optimization processes can select a proper execution order satisfying the dependence constraints to minimize the buffer requirement between the two tasks. The approach to reducing the buffer size is to match the access order of the array data between two tasks. A comprehensive task instance scheduling methodology is beyond the scope of this paper. Accordingly, we just evaluate two simple scheduling candidates: row-by-row and column by column for both tasks. For the row by row candidate, the intratask dependence for task t0 requires that task t0 start from row 1 to row N, while task t1 needs to start from row N to row 1, so the buffer size is not reduced. But for the column-by-column candidate, we can reorder the task instances of task t0 so that the array A is written column-by-column as shown in Figure 7(c). In this case, the required buffer size is reduced from N×N to N. The final scheduling of task instances is shown in Figure 7(d): each task processes one array column in turn.

Listing 23 shows the generated hardware module by TLDM-based task-level optimization, and it can be further synthesized by HLS tools. The order of the task instances and task-level pipeline are optimized to reduce the buffer size between the two tasks.

5. Advantages of TLDM

Task-level hardware synthesis requires a desired concurrency specification model that is (i) powerful enough to model all the applications in the domain, (ii) simple enough to be used by domain experts, independent of the implementation details, (iii) flexible enough for integrating diverse computations (potentially in different languages or stages of refinement), and (iv) efficient enough for generating a high quality of results in a hardware synthesis flow. This section presents concrete examples to illustrate that the proposed TLDM specification model is designed to satisfy these requirements.

5.1. Task-Level Optimizations. Since our TLDM model is derived from CnC, the benefit of implicit parallelism is common to both models. However, other models, like dataflow networks, do not model dynamic parallelism. The sequential code in Figure 8 is a simple example of parallelizable loops. For process networks, the processes are defined in a sequential way, so the concurrency can only be specified by initiating distinct processes. The number of parallel tasks (parameter p in Figure 8) is specified statically by the user—represented as multiple processes in the network. However, a single statically specified number may not be suitable for different target platforms. For example, on a multicore CPU, the program can be broken down into 8 to 16 segments that can be executed in parallel; however, on a GPU with hundreds of processing units, the program needs to be broken down into segments of finer granularity. In our TLDM, users only need to specify a task collection with its iterator domain and data accesses, and then the maximal parallelism between task instances is implicitly defined. The runtime/synthesis tool will determine the number and granularity of segments to run in parallel, depending on the target platform.

In addition, the collection-based specification does not introduce any redundant execution order constraints on the task scheduling. For example, in the data streaming application in Figure 9, the KPN specification needs to explicitly define the loop order in each task process. In the synthesis and optimization flow, it is possible to reduce

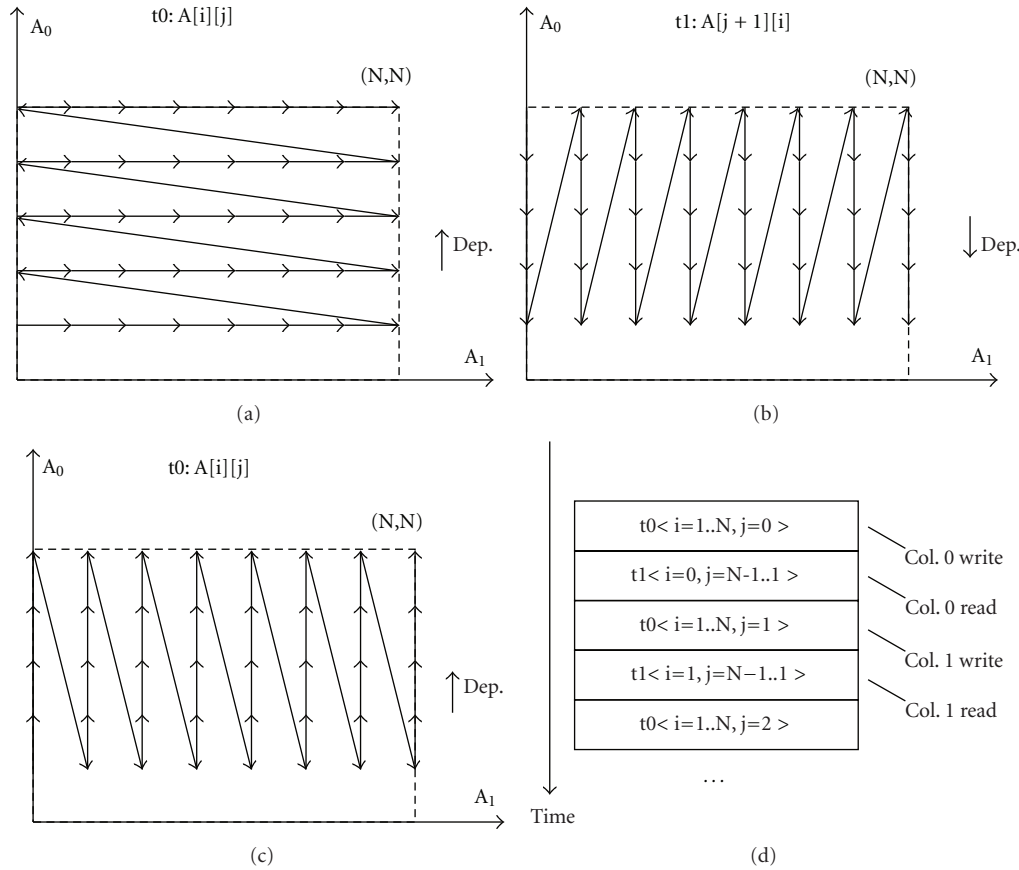


Figure 7: (a) Initial access order of t0 writes. (b) Initial access order of t1 reads. (c) Optimized access order of t0 writes. (d) Optimized task instance scheduling for buffer reduction.

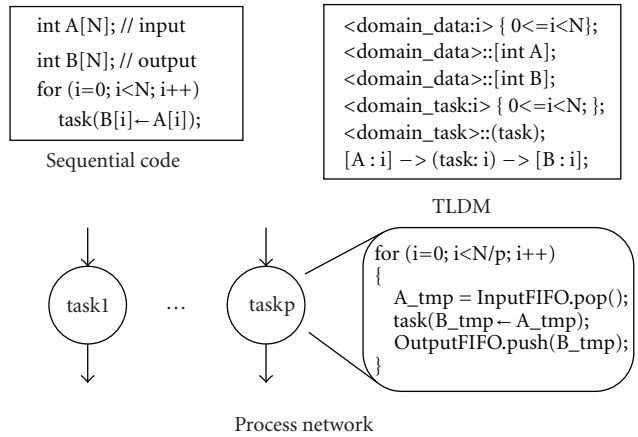


Figure 8: Implicit parallelism (TLDM) versus explicit parallelism (process network).

the buffer size between tasks by reordering the execution order of the loop iterations. But these optimizations are relatively hard for a KPN-based flow because the optimizer needs to analyze deep into the process specification and interact with module-level implementation. However, in our TLDM specification, necessary task-instance-level dependence information is explicitly defined without redundant order constraints in sequential languages. This offers the possibility and convenience of performing task-level optimizations without touching module-level implementation.

5.2. Heterogeneous Platform Supports. Heterogeneous platforms, such as the customized heterogeneous platform being developed by CDSC [9, 36], have gained increased attention

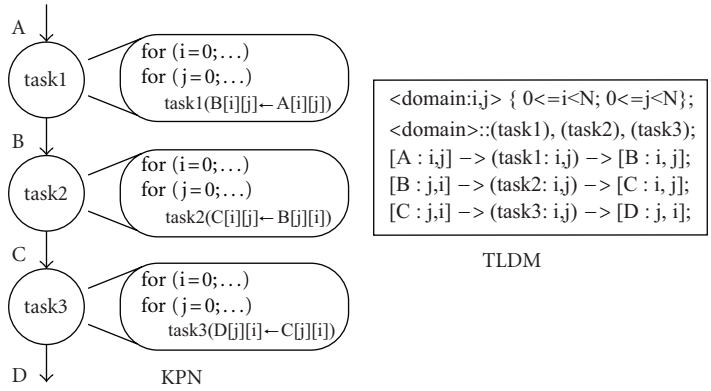


Figure 9: Support for task rescheduling.

in high-performance computation system design because of the benefits brought about by efficient parallelism and customization in a specific application domain. Orders-of-magnitude efficiency improvement for applications can be achieved in the domain using domain-specific computing platforms consisting of customizable processor cores, GPUs, and reconfigurable fabric.

CnC is essentially a coordination language. This means that the CnC specification is largely independent of the internal implementation of a step collection, and individual step collections could be implemented in different languages/models. This makes integrating diverse computational steps simpler for users, compared to rewriting each step into a common concurrent specification. The module-level steps could be implemented in C/C++/Java for GPPs, CUDA/OpenCL for GPUs, or even HDL such as VHDL and Verilog for hardware accelerator designs. In addition, because CnC is a coordination description supporting different implementations, it naturally supports an integrated flow to map domain-specific applications onto heterogeneous platforms by selecting and refining the module-level implementations. Our TLDM model inherits this feature directly from CnC.

We take medical image processing applications as an example. There are three major filters (denoise, registration, and segmentation) in the image processing pipeline. An image sequence with multiple images will be processed by the three filters in a pipelined way. At the task level, we model each filter as a task, and each task instance processes one image in our TLDM model. The task-level dataflow is specified in the TLDM specification, which provides enough information for system-level scheduling. Module-level implementation details are embedded in module-level specifications, which can support different languages and platforms. In the example shown in Figure 10, we are trying to map three filters onto a FPGA+GPU heterogeneous platform. One typical case is that denoise is implemented in an FPGA by a high-level synthesis flow from C++ specification, registration is mapped onto a GPU, and segmentation is specified as a hardware IP in the FPGA. Module specifications using C++ (for HLS-FPGA synthesis), CUDA (for GPU compiling), and RTL (for hardware design IPs) are coordinated with a common TLDM model. For each execution of task instance in TLDM, the module-level function is called once for C++ and CUDA cases, or a synchronization cycle by enabled and done signals is performed for the RTL case, which means the RTL IP has processed one image. The related input data and space for output data are allocated in a unified and shared memory space, so that the task-level data transfer is transparent to the module designers; this largely simplifies the module design in a heterogeneous platform.

To map the application in Figure 10 onto a heterogeneous platform, each task in the TLDM specification can have multiple implementation candidates—such as multi-core CPU and many-core GPU and FPGA versions. These module-level implementation candidates share the same TLDM specification. A synthesizer and simulator can be used to estimate or profile the physical metrics for each task and each implementation candidate. According to this module-level information and the task-level TLDM specification, efficient task-level allocation and scheduling can be performed to determine the architecture mapping and optimize the communication.

5.3. Suitability for Hardware Synthesis. Our TLDM is based on the Intel CnC model. Task, data, and iterator in TLDM are the counterparts of step, data item, and control tag in CnC, respectively. Regular tasks and data are specified in a compact and iterative form and indexed by iterators and array subscripts. Both task instances in TLDM and prescribed steps in CnC need to wait for all the input data to be available in order to start the execution and do not ensure the availability of the output data until the execution of the current instance is finished. The relative execution order of task and step instances is only constrained by true data dependence, not by textual positions in sequential programs. However, there are also great differences between TLDM and CnC. By restricting the general dynamic behavior allowed by CnC, TLDM is more suitable to hardware synthesis for most practical applications, and it differs from the general CnC in the following five aspects.

(1) CnC allows dynamically generating step instances; this is hard to implement in hardware synthesis. In addition,

```
tldm_app tiled_cholesky;

// iterator variables
tldm_data iterator_i (“i”); // scalar variable
tldm_data iterator_j (“j”);
tldm_data iterator_k (“k”);

// environment parameters
tldm_data param_p (“p”);

// array A[p][p][i + 1]
tldm_data array_A(“A”, 3); // n_dimension = 3
array_A.insert_index_constraint(0, “>=”, 0);
array_A.insert_index_constraint(0, “<”, p);
array_A.insert_index_constraint(1, “>=”, 0);
array_A.insert_index_constraint(1, “<”, p);
array_A.insert_index_constraint(2, “>=”, 0);
array_A.insert_index_constraint(2, “<”, p+1);

// attach data into tldm application
tiled_cholesky.attach_data(&iterator_i);
tiled_cholesky.attach_data(&iterator_j);
titled_cholesky.attach_data(&iterator_k);
tiled_cholesky.attach_data(&param_p);
tiled_cholesky.attach_data(&array_A);

// iteration domain of task seq_cholesky
tldm_iteration_domain id_k;
id_k.insert_iterator(iterator_k); // “k”
id_k.insert_affine_constraint(“k”, 1, “>=”, 0); // k*1 >= 0
id_k.insert_affine_constraint(“k”, -1, “p”, 1, “>”, 0); // -k+p > 0

// accesses A[k][k][k+1]
tldm_access acc_A0 (&array_A, WRITE);
acc_A0.insert_affine_constraint(“A(0)”, 1, “k”, -1, “=”, 0); // A0 = k
acc_A0.insert_affine_constraint(“A(1)”, 1, “k”, -1, “=”, 0); // A1 = k
acc_A0.insert_affine_constraint(“A(2)”, 1, “k”, -1, “=”, 1); // A2 = k+1
// accesses A[k][k][k]
tldm_access acc_A1 (&array_A, READ);
acc_A1.insert_affine_constraint(“A(0)”, 1, “k”, -1, “=”, 0);
acc_A1.insert_affine_constraint(“A(1)”, 1, “k”, -1, “=”, 0);
acc_A1.insert_affine_constraint(“A(2)”, 1, “k”, -1, “=”, 0);

// task seqCholesky
tldm_task seq_cholesky(“seqCholesky”);
seq_ cholesky.attach_id(&id_k);
seq_cholesky.attach_access(&acc_A0);
seq_cholesky.attach_access(&acc_A1);
seq_cholesky.attach_parent(NULL);
tiled_cholesky.attach_task(&seg_cholesky);


// iteration domain of task tri_solve
tldm_iteration_domain id_kj = id_k.copy(); // 0 <= k < p
id_kj.insert_iterator(iterator_j); // “j”
id_kj.insert_affine_constraint(“j”, 1, “k”, -1, “>=”, 1); // j-k >= 1
id_kj.insert_affine_constraint(“j”, -1, “p”, 1, “>”, 0); // -j+p > 0
// accesses A[j][k][k + 1]
```

Listing 14: Continued.

```
tldm_access acc_A2 = acc_A0.copy();
acc_A2.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[j][k][k]
tldm_access acc_A3 = acc_A1.copy();
acc_A3.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[k][k][k + 1]
tldm_access acc_A4 = acc_A1.copy();
acc_A4.replace_affine_constraint("A(2)", 1, "k", -1, "=", 1); // A2 = k+1

// task TriSolve
tldm_task tri_solve("TriSolve");
tri_solve.attach_id(&id_kj);
tri_solve.attach_access(&acc_A2);
tri_solve.attach_access(&acc_A3);
tri_solve.attach_access(&acc_A4);
tri_solve.attach_parent(NULL);
tiled_cholesky.attach_task(&tri_solve);

// iteration domain of task update
tldm_iteration_domain id_kji = id_kj.copy();
id_kji.insert_iterator(iterator_i); // "i"
id_kji.insert_affine_constraint("i", 1, "k", -1, ">=", 1); // i-k >= 1
id_ kji.insert_affine_constraint("i", -1, "j", 1, ">=", 0); // -i+j >= 0

// accesses A[j][i][k + 1]
tldm_access acc_A5 = acc_A2.copy();
acc_A5.replace_affine_constraint("A(1)", 1, "i", -1, "=", 0); // A1 = i
// accesses A[j][k][k + 1]
tldm_access acc_A6 = acc_A4.copy();
acc_A6.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[i][k][k + 1]
tldm_access acc_A7 = acc_A4.copy();
acc_A7.replace_affine_constraint("A(0)", 1, "i", -1, "=", 1); // A0 = i

// task Update
tldm_task update ("Update");
update.attach_id(&id_kji);
update.attach_access(&acc_A5);
update.attach_access(&acc_A6);
update.attach_access(&acc_A7);
update.attach_parent(NULL);
tiled_cholesky.attach_task(&update);

// dependence: TriSolve<k,j> -> Update<k,j, *>
tldm_dependence dept(&tri_solve, &update);
dept.insert_affine_constraint("k", 1, 0, "->", "k", 1, 0); // k0=k1
dept.insert_affine_constraint("j", 1, 0, "->", "j", 1, 0); // j0=j1
```

Listing 14: C++ TLDM specification of tiled Cholesky.

data collection in CnC is defined as unbounded: if a new data tag (like array subscripts) is used to access the data collection, a new data unit is generated in the data collection. In TLDM, iteration domain and data domain are statically defined, and the explicit information of these domains helps to estimate the corresponding resource costs in hardware synthesis. Data domain in TLDM is bounded: out-of-bounds access will be forbidden in the specification.

Consider the CnC specification for the vector addition example in Figure 11. The environment is supposed to generate control tags that prescribe the individual step instances that perform the computation. Synthesis tools would need to synthesize hardware to (i) generate the control tags in some sequence and (ii) store the control tags till the prescribed step instances are ready to execute. Such an implementation would be inefficient for such a simple program. TLDM works around this issue by introducing the concept of iteration domains. Iteration domains specify that all control tags are generated as a numeric sequence with a constant stride. This removes the need to synthesize hardware for explicitly generating and storing control tags.

(2) In CnC, input/output accesses to the data collections are embedded implicitly in the step specification. TLDM adopts explicit specification for inter-task data accesses. In

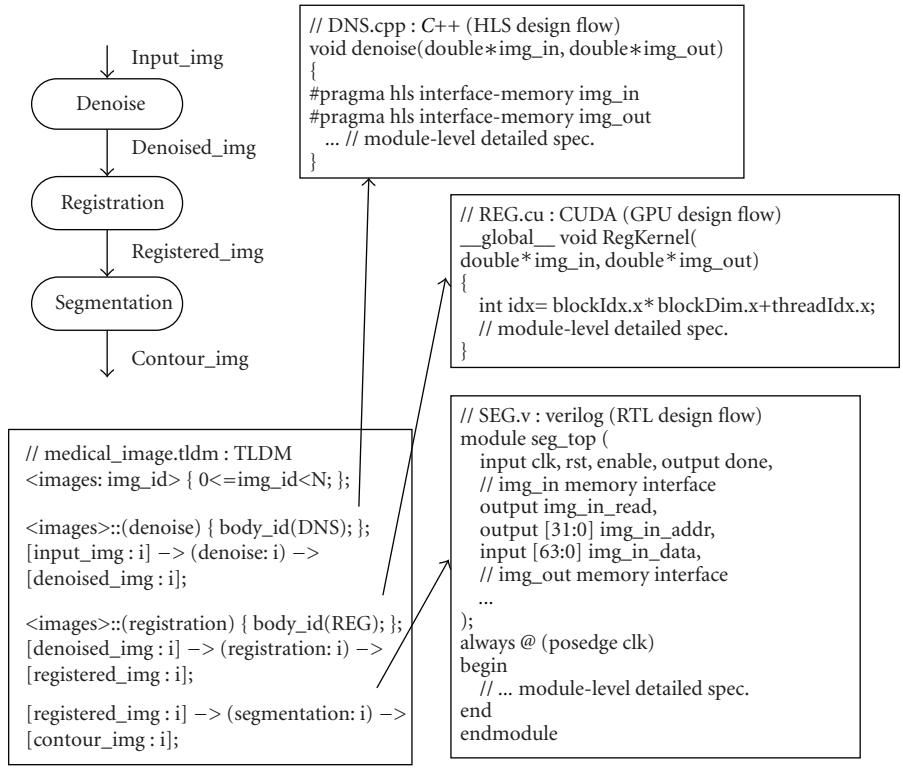


Figure 10: Heterogeneous computation integration using TLDM.

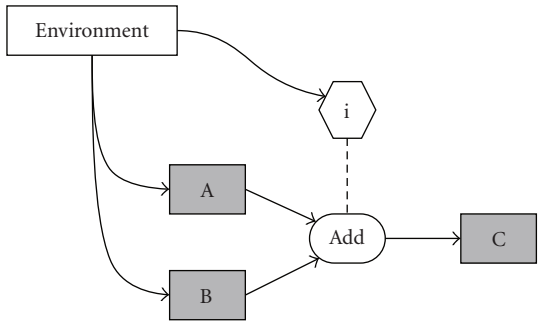


Figure 11: CnC representation for vector add.

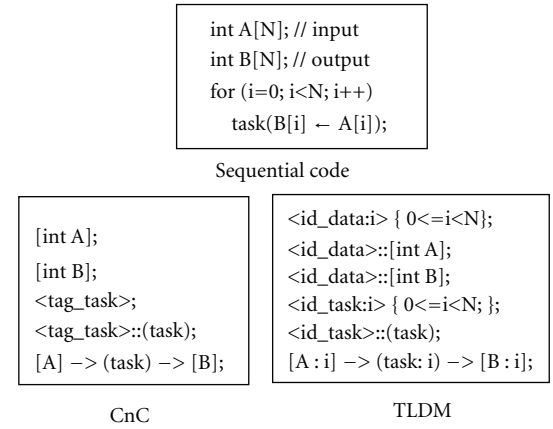


Figure 12: TLDM versus CnC: iterator domain and access pattern are explicitly specified in TLDM.

our simple example in Figure 12, the task *task* reads data from data collections A, but for a given step instance i, task-level CnC does not specify the exact data element to be read. This makes it hard for hardware synthesis to get an efficient implementation without the access information. TLDM specifies the access patterns directly, which can be used for dependence analysis, reuse analysis, memory partitioning, and various memory-related optimizations.

(3) In the static specification for the domain and mapping relations, a polyhedral model is adopted in TLDM to model the integer sets and mappings with linear inequality and equations. As shown in Figure 13, the range of the iterators can be defined as linear inequalities, which can be finally expressed concisely with integer matrices. Each row of the matrix is a linear constraint, and each column of the matrices is the coefficient associated with each variable. Standard polyhedral libraries can be used to perform transformation on these integer sets and to analyze specific metrics such as counting the size. The polyhedral model framework helps us to utilize linear/convex programming and linear algebra properties in our synthesis and optimization flow.

Note that TLDM introduces a hybrid specification to model the affine parts in the polyhedral model and non-affine parts in the polyhedral approximation or nonaffine execution conditions. Compared to the traditional linear

```
<> :: (top_level) // top level task, single instance
{
<> :: [data_type p]; // no domain for scalar variable

// data A[p][p][p+1]
[p] -> <A_dom : a0,a1,a2> {0<=a0<p; 0<=a1<p; 0<=a2<p+1;};
<A_dom> :: [array_A];

// task seqCholesky
[p] -> <task1_dom:k> {0<=k<p;};
<task1_dom> :: (task1) {body_id("seqCholesky");};
[A : k, k, k] -> (task1:k) -> [A : k, k, k + 1];
//task TriSolve
[p] -> <task2_dom : k,j> {0<=k<p; k+1<=j<p;};
<task2_dom> :: (task2) {body_id("TriSolve");};

// task Update
[p] -> <task3_dom:k,j,i> {0<=k<p; k+1<=j<p; k+1<=i<=j;};
<task3_dom> :: (task3) {body_id("Update");};
[A : j, k, k + 1],[A : i, k, k + 1] -> (task3 : k,j,i) -> [A : j, i, k + 1];

//dependence
(task2 : k,j) -> (task3 : k,j,(k+1)..j)
};
```

Listing 15: Textual TLDM specification of tiled Cholesky.

```
for (i=0; i<N; i++)
    for (j=0; j<i*i; j++) // nonaffine boundary i*i
        task0(...);
```

Listing 16: Example C-code of nonaffine iterator boundary.

```
tldm_iteration_domain id_ij;
id_ij.insert_iterator(iterator_i); // "i"
id_ij.insert_iterator(iterator_j); // "j"
id_ij.insert_affine_constraint("i", 1, ">=", 0); // i*1 >= 0
id_ij.insert_affine_constraint("i", -1, "N", 1, ">", 0); // -i+N > 0
id_ij.insert_affine_constraint("j", 1, ">=", 0); // i*1 >= 0
id_ij.insert_affine_constraint("j", -1, "nonAffine(i)", 1, ">", 0); // -j+(i*i)>0

[N] -> <id_ij: i, j> {0<=i<N; 0<=j<i*i;};
```

Listing 17: TLDM specification of nonaffine iterator boundary.

```
while (!convergence)
    for (i=0; i<N; i++)
        task1(i, convergence,...); // convergence is updated in task0
```

Listing 18: Example C-code of convergence algorithm.

```
tldm_data convergence_data("convergence");

tldm _iteration_domain id_ti;
id_ti.insert_iterator(iterator_t); // "t" for the while loop
id_ti.insert_iterator(iterator_i); // "i"
id_ti.insert_affine_constraint("i", 1, ">=", 0); // i*1 >= 0
id_ti.insert_affine_constraint("i", -1, "N", 1, ">", 0); // -i+N > 0
id_ti.insert_affine_constraint("t", 1, ">=", 0); // t >= 0
tldm_expression exe_condition(&iterator_t, "!", &convergence_data);
id_ti.insert_exe_condition(&exe_condition);

// non-DSA access: different iterations access the same scalar data unit
tldm_access convergence_acc (&convergence_data, WRITE);

tldm_task task1("task1");
seq_cholesky.attach_id(&id_ti);
seq_cholesky.attach_access(&convergence_acc);

// dependence is needed to specify to avoid access conflicts
tldm_dependence dept(&task1, &task1);
// dependence : task1<t> → task1<t+1>
// dept.insert_affine_constraint ("t", 1, 0, "->", "t", 1, 1); // (t+0) → (t+1)
// dependence are added to make the while loop iterations to execute in sequence, 0, 1, 2, ...
<> :: [convergence];
[N] -> < id_ti : t, j> {cond(!convergence); 0<=i<N;};
<id_ti> ::(task1) {body_id("task1")};
(task1: t, i) -> [convergence]
(task1 : t-1, 0..N) -> (task1 : t, 0..N);
```

Listing 19: TLDM specification of convergence algorithm.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        task2(... ← A [idx[j]][i],...); // implication : x<= idx[x] <= x+M
```

Listing 20: Example C-code of indirect access.

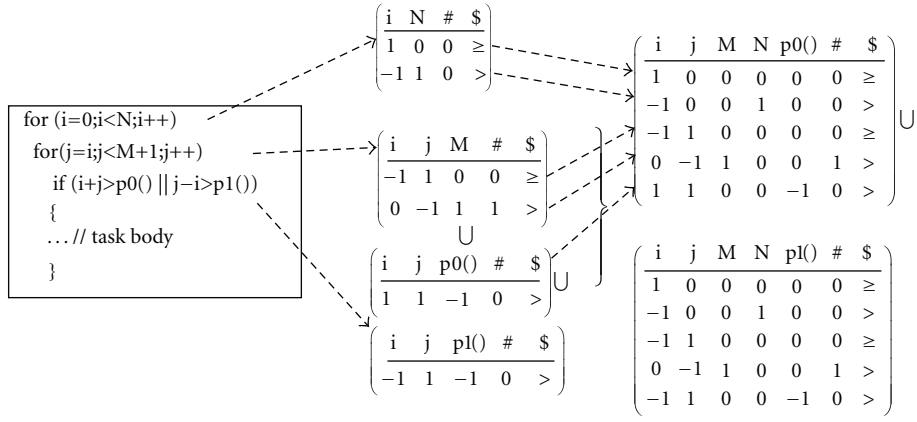


Figure 13: Polyhedral representation for iterator domain.

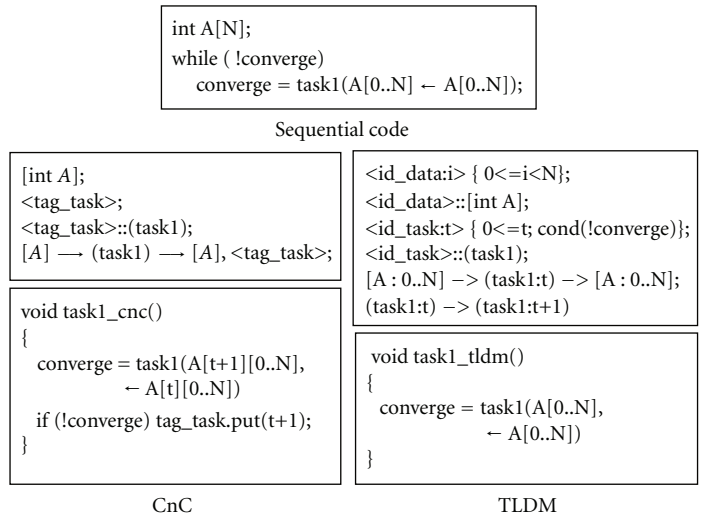


Figure 14: DSA (CnC) versus user-specified dependence (TLDM).

```
tldm_data array_A("A");

// accesses A[idx[j]][i] // implication: x<= idx[x] < x+M
tldm_access indirect_acc(&array_A, READ);
indirect_acc.insert_affine_constraint("A(0)", 1, "j", -1, ">=", 0); // A0 >= j
indirect_acc.insert_affine_constraint("A(0)", 1, "j", -1, "M", -1, "<", 0); // A0 < j+M
indirect_acc.insert_affine_constraint ("A(1)", 1, " i", -1, "=", 0); // A1 = i

[A : j..(j + M), i] -> (task2 : i, j)
```

Listing 21: TLDM specification of indirect access.

```
1 for (i=1; i<=N; i++)
2     for (j=0; j<=N; j++)
3         t0: A[i][j] = A[i-1][j] + . . .;
4 for (i=0; i<=N; i++)
5     for (j=N-1; j>=0; j- -)
6         t1: A[j][i] = A[j + 1][i] +. . .;
```

Listing 22: Example application for buffer reduction.

polyhedral model, we support the extensions in the following three aspects. First, if the boundaries of the iterator are in the nonaffine form of outer iterators and task-independent parameters, parallelism and execution order transformation for this non-affine-bound iterator is still possible. We introduce pseudoparameters to handle these inner-loop-independent nonaffine boundaries. Second, if the boundaries of the iterator are dependent on data items generated by the task itself—for example, in the case of a convergent algorithm—nonaffine data-dependent execution conditions are specified explicitly by *tldm_expression* objects, which are easy to analyze in the optimization and hardware generation flows. Third, for nonaffine array accesses or block data access in an array, we support a polyhedral-based form to specify the affine (rectangle) hull of the possible accessed array elements. With this approach, we can conservatively and efficiently handle the possible optimizations for many of the nonaffine array accesses—such as indirect accesses (shown in Listing 21).

(4) CnC restricts dynamic single assignment (DSA) in the data access specification to achieve the intrinsic deterministic characteristics in a concurrent specification. But practical experience shows that the DSA restriction is not convenient for designers when specifying their application for efficient hardware implementation. Memory space reuse in CnC is forbidden, which makes the designer lose the capability of specifying memory space reuse schemes. In addition, loop-carried variables need to maintain one duplication for each iteration; this leads to an unbounded data domain when the iteration domain is unbounded. For example, in Figure 14 array A needs to have different copies for each iteration in the CnC specification; this leads to an additional dimension for A to index the specific copy. In TLDM we do not enforce the DSA restriction in the specification, but DSA is still recommended. For those cases in which designers intend to break DSA restriction, write access conflicts need to be handled by the designer using our user-specified dependence specifications to constrain the possible

```
1  col_buf[N];
2  for (i=0; i<=N; i++)
3  { // pragma hls loop pipeline
4       t0(1..N, i, col_buf); // write one column of array A[N][N]
5       t1(i, N-1..1, col_buf); // read one column of array A[N][N]
6  }
```

Listing 23: Generated HW module after task-level scheduling.

conflict accesses into different time spans. In Figure 14 dependence (task1 : t)->task(1 : t+1) ensures the sequential execution of the convergence iterations, which can guarantee the correctness without DSA constraints.

(5) The traditional CnC specification is not scalable because it does not support the hierarchy of steps. In our TLDM specification, task graphs are built in a hierarchical way. Designers can specify a set of highly coupled subtasks into a compound task. Hardware synthesis flow can select various methodologies to perform coarse-grained or fine-grained optimizations based on the task hierarchy. In addition, specific synthesis constraints or optimization targets can be applied to different compound tasks according to the computation characteristics of the compound tasks. Data elements can also be defined in a smaller scope for better locality in memory optimizations.

5.4. Discussion and Comparison. Table 2 compares some widely used concurrency specifications in different aspects. CSP is the general underlying MoC of the event-driven Metropolis framework. The PSM model is specified in a structural language, SpecC, and used in the SCE framework. SDF and KPN are the most popular models for data-intensive applications and have been adopted by StreamIt and Daedalus projects. CnC and TLDM were recently proposed to separate implementation optimizations from concurrency specifications.

In Table 2 the first column lists the features for the concurrency specifications. In the table, solid circles, hollow circles, and blanks mean full support, partial support, and no support, respectively. All of these models are designed to support data streaming applications because they are the most popular application types in electronic system design. For control-intensive applications, CSP and CnC can support general irregular (conditional) behavior like dynamic instance generation and conditional execution; PSM has an intrinsic support for concurrent FSMs; KPN and TLDM support data-dependent execution, but their process instance is statically defined; SDF has regular data accesses and actor firing rates, which make it inefficient to model conditional execution. CSP is nondeterministic because of its general trigger rules; PSM has parallel processes that can write the same data; other models are all deterministic. Hierarchy is fully supported by CSP, PSM, and TLDM, and hierarchical CnC is under development now. In CSP, PSM, and KPN, data parallelism has to be specified explicitly because they do not have the concept of task instance as CnC and TLDM. SDF has partially scalable data parallelism because the data parallelism can be relatively easily exploited by automated transformations on the specification like StreamIt. Reordering the task instance can help to improve the task parallelism and memory usage, which can only be supported by CnC and TLDM. CSP and CnC support dynamic generation of task instances, and the dynamic behavior is hard to analyze statically by a hardware synthesizer. On the contrary, SDF, KPN, and TLDM are limited in multicore processor-based platforms because the user needs to transform the dynamic specification into equivalent static specification. CSP (Metropolis), PSM and TLDM have explicit specification for the interfaces and accesses between tasks, which can fully encapsulate the implementation details of the task body for different platforms. In SDF, KPN, and CnC, the access information is embedded in the task body specification, and additional processes are needed to extract the information from different specifications of heterogeneous platforms. From the comparison, we can see that the proposed TLDM is the most suitable specification for hardware synthesis, especially for data-intensive applications.

6. Conclusion and Ongoing Work

This paper proposes a practical high-level concurrent specification for hardware synthesis and optimization for data processing applications. In our TLDM specification, parallelism of the task instances is intrinsically modeled, and the dependence constraints for scheduling are explicit. Compared to the previous concurrent specification, TLDM aims to specify the applications in a static and bounded way with minimal overconstraints for concurrency. A polyhedral model is embedded in the specification of TLDM as a standard and unified representation for iteration domains, data domains, access patterns, and dependence. Extensions for nonaffine terms in the program are comprehensively considered in the specification as well to support the analysis and synthesis of irregular behavior. Concrete examples show the benefits of our TLDM specification in modeling a task-level concurrency for hardware synthesis in heterogeneous platforms. We are currently in the process of developing the TLDM-based hardware synthesis flow.

Acknowledgments

This work is partially funded by the Center for Domain-Specific Computing (NSF Expeditions in Computing Award

Table 2: Summary of the concurrency specifications.

	CSP	PSM	SDF	KPN	CnC	TLDM
Data streaming application	•	•	•	•	•	•
Control-intensive application	•	•		◦	•	◦
Nondeterministic behavior	•	◦				
Hierarchical structure	•	•			◦	•
Scalable data parallelism			◦		•	•
Task instance reordering					•	•
HW Synthesis		•	•	•		•
Multicore platform	•	•	◦	◦	•	◦
Heterogeneous platform	•	•	◦	◦	◦	•

CCF-0926127) and the Semiconductor Research Corporation under Contract 2009-TJ-1984. The authors would like to thank Vivek Sarkar for helpful discussions and Janice Wheeler for editing this paper.

References

[1] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[2] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, no. 3, Article ID 4167779, pp. 467–506, 2007.

[3] "An independent evaluation of the AutoESL autopilot high-level synthesis tool," Tech. Rep., Berkeley Design Technology, 2010.

[4] C. A. R. Hoare, "Communicating sequential processes. Commun," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[5] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[6] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 19, pp. 1235–1245, 1987.

[8] Intel—Concurrent Collections for C/C++: User's Guide, 2010, http://software.intel.com/file/30235.

[9] J. Cong, G. Reinman, A. Bui, and V. Sarkar, "Customizable domain-specific computing," *IEEE Design and Test of Computers*, vol. 28, no. 2, pp. 6–14, 2011.

[10] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[11] SystemC, http://www.accellera.org/.

[12] BlueSpec, http://bluespec.com/.

[13] FDR2 User Manual, 2010, http://fsel.com/documentation/fdr2/html/fdr2manual_5.html.

[14] ARC CSP model checking environment, 2010, http://cs.adelaide.edu.au/~esser/arc.html.

[15] R. Allen, *A formal approach to software architecture*, Ph.D. thesis, Carnegie Mellon, School of Computer Science, 1997, Issued as CMU Technical Report CMU-CS-97-144.

[16] T. Murata, "Petri nets: properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[17] Petri net, 2010, http://en.wikipedia.org/wiki/Petri_net.

[18] A. Davare, D. Densmore, T. Meyerowitz et al., "A next-generation design framework for platform-based design," *DVCon*, 2007.

[19] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, *Readings in Hardware/Software Co-Design. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, Kluwer Academic, Norwell, Mass, USA, 2002.

[20] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed., North-Holland, Stockholm, Sweden, August 1974.

[21] H. Nikolov, M. Thompson, T. Stefanov et al., "Daedalus: toward composable multimedia MP-SoC design," in *Proceedings of the 45th Design Automation Conference (DAC '08)*, pp. 574–579, ACM, New York, NY, USA, June 2008.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: a language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*, pp. 49–84, Springer, Berlin, Germany, 2002.

[23] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proceedings of the International Workshop on Hardware/Software Co-Design*, 1993.

[24] 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.

[25] F. Balarin, M. Chiodo, and P. Giusto, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic, Norwell, Mass, USA, 1997.

[26] C. Park, J. Jung, and S. Ha, "Extended synchronous dataflow for efficient DSP system prototyping," *Design Automation for Embedded Systems*, vol. 6, no. 3, pp. 295–322, 2002.

[27] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. P. Joo, "PeaCE: a hardware-software codesign environment for multimedia embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, 2007.

[28] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—an internal design representation for codesign," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524–544, 2001.

[29] K. Grüttner and W. Nebel, "Modelling program-state machines in SystemC^{TM}," in *Proceedings of the Forum on Specification, Verification and Design Languages (FDL '08)*, pp. 7–12, Stuttgart, Germany, September 2008.

[30] R. Dömer, A. Gerstlauer, J. Peng et al., "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 647953, 13 pages, 2008.

[31] M. Fujita and H. Nakamura, "The standard SpecC language," in *Proceedings of the 14th International Symposium on System Synthesis (ISSS '01)*, pp. 81–86, ACM, New York, NY, USA, 2001.

[32] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 207–216, August 1995.

[33] Habanero-C project, 2011, https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[34] V. Cave, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old x10," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ '11)*, 2011.

[35] Intel CnC distribution, 2011, http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[36] Center for Customizable Domain-Specific Computing (CDSC), http://cdsc.ucla.edu/.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 593532, 12 pages
doi:10.1155/2012/593532

Research Article

Selectively Fortifying Reconfigurable Computing Device to Achieve Higher Error Resilience

Mingjie Lin,[1] Yu Bai,[1] and John Wawrzynek[2]

[1] *Department of Electrical Engineering and Comouter Science, University of Central Florida, Orlando, 32816 FL, USA*
[2] *Department of Electrical Engineering and Comouter Science, University of California at Berkeley, Berkeley, 94720 CA, USA*

Correspondence should be addressed to Mingjie Lin, mingjie@eecs.ucf.edu

Received 11 September 2011; Revised 9 January 2012; Accepted 11 January 2012

Academic Editor: Deming Chen

With the advent of 10 nm CMOS devices and "exotic" nanodevices, the location and occurrence time of hardware defects and design faults become increasingly unpredictable, therefore posing severe challenges to existing techniques for error-resilient computing because most of them statically assign hardware redundancy and do not account for the error tolerance inherently existing in many mission-critical applications. This work proposes a novel approach to selectively fortifying a target reconfigurable computing device in order to achieve hardware-efficient error resilience for a specific target application. We intend to demonstrate that such error resilience can be significantly improved with effective hardware support. The major contributions of this work include (1) the development of a complete methodology to perform sensitivity and criticality analysis of hardware redundancy, (2) a novel problem formulation and an efficient heuristic methodology to selectively allocate hardware redundancy among a target design's key components in order to maximize its overall error resilience, and (3) an academic prototype of SFC computing device that illustrates a 4 times improvement of error resilience for a H.264 encoder implemented with an FPGA device.

1. Introduction

With the advent of 10 nm CMOS devices and "exotic" nanodevices [1–3], error resilience is becoming a major concern for many mission- and life-critical computing systems. Unfortunately, as these systems grow in both design size and implementation complexity, device reliability severely diminishes due to escalating thermal profiles, process-level variability, and harsh environments (such as outer space, high-altitude flight, nuclear reactors, or particle accelerators). Unlike design faults, device failures often are *spatially probabilistic* (the locations of hardware defects and design faults within a chip are unknown) and *temporally unpredictable* [4, 5] (the occurrence time of hardware defects are hard to foretell).

1.1. Related Work. Traditionally, either special circuit technique or conservative design is employed to ensure correct operation and to achieve high error resilience. Existing circuit techniques include guard banding, conservative voltage scaling, and even radiation hardening [6, 7]. For example, qualified versions of SRAM-based FPGAs, such as Xilinx's QPro [8], are commercially available for mitigating SEUs at the circuit level. More recently, hardware designers started to employ information redundancy, hardware redundancy, time redundancy, or a combination of these techniques in order to circumvent hardware defects at the device, circuit, and architectural levels. Specifically, the use of redundant hardware, data integrity checking, and data redundancy across multiple devices is gaining considerable popularity [9–11]. In fact, several engineers and researchers [12, 13] have demonstrated that logic-level triple-modular redundancy (TMR) with scrubbing of the FPGAs programming (or configuration) data effectively mitigates the results of radiation-induced single-bit upsets (SBUs). However, most of these redundancy techniques detect and recover from errors through expensive redundancy statically, that is, the allocation of hardware redundancy based on previously known or estimated error information, usually treating all

components within a system indiscriminately, and therefore incur huge area and performance penalties.

Recognizing TMR is costly; a recent study from BYU [14] proposed applying partial TMR method on the most critical sections of the design while sacrificing some reliability of the overall system. Specially, they introduced an automated software tool that uses the partial TMR method to apply TMR incrementally at a very fine level until the available resources are utilized, thus maximizing reliability gain for the specified area cost. Although with apparent conceptual similarity to this work, there are several important distinction between studies [14–16] and this one. First, while the study [14] focuses primarily on TMR or partial one, we consider more general case of selecting among n-nary modular redundancy (nMR) choices, where $n = 3, 5, 7, \ldots$. This generalization turned out to be important. In Section 7, we will show that in our target H.264 application, if more than TMR, such as 7-MR, is applied to certain components, the overall error resilience can be significantly improved. In addition, we provide a stochastic formulation of maximizing a system's error resilience when hardware failures are *spatially probabilistic* and *temporally unpredictable*. As a result, we can in principle obtain a mathematically provable optimal solution to maximizing the system's overall error resilience while being constrained by a total available hardware redundancy. Finally, while the techniques studied in [14–16] are mostly specific to FPGA fabric, our proposed methodology can be readily introduced into logic synthesis for an application-specific integrated circuit (ASIC) design.

Identifying the most critical circuit structures, thus trading hardware cost for SEU immunity, is not a new idea. In fact, Samudrala et al. proposed the selective triple modular redundancy (STMR) method which uses signal probabilities to find the SEU-sensitive subcircuits of a design [15]. Morgan et al. have proposed a partial mitigation method based on the concept of persistence [16]. We approach the criticality analysis differently in this study. Instead of being logic circuit or FPGA fabric-specific, we analyze individual system criticality by computing output sensitivity subjected to various amount of input perturbation probabilistically. Therefore, we are able to not only distinguish critical and noncritical components, but also quantitatively assigning different criticality values, which leads to a more cost-effective allocation of hardware redundancy.

This paper proposes a selectively fortified computing (SFC) approach to providing error resilience that preserves the delivery of expected performance and accurate results, despite of the presence of faulty components, in a robust and efficient way. The key idea of SFC is *to judiciously allocate hardware redundancy according to each key component's criticality towards target device's overall error resilience*. It leverages two emerging trends in modern computing. First, while ASIC design and manufacturing costs are soaring today with each new technology node, the computing power and logic capacity of modern FPGAs steadily advance. As such, we anticipate that FPGA-like reconfigurable fabric, due to its inherent regularity and built-in hardware redundancy, will become increasingly attractive for robust computing. Moreover, we believe that modern FPGA devices, bigger and more powerful, will become increasingly more suitable hardware platforms to apply the SFC concept and methodology for the improvements of fault-tolerance and computing robustness. Second, many newly emerging applications, such as data mining, market analysis, cognitive systems, and computational biology, are expected to dominate modern computing demands [17]. Unlike conventional computing applications, they typically process massive amounts of data and build mathematical models in order to answer real-world questions and to facilitate analyzing complex system, therefore tolerant to imprecision and approximation. In other words, for these applications, computation results need not always be perfect as long as the accuracy of the computation is "acceptable" to human users [18].

The rest of the paper is organized as follows. Section 2 overviews the proposed SFC framework and Section 3 discusses its target applications. We then delve into more detailed descriptions of criticality analysis and optimally allocating hardware redundancy with constrained hardware allowance. Subsequently, Section 6 describes in detail our H.264 decoder prototype and numerous design decisions. Finally, we present and analyze the error resilience results from a SFC design against its baseline without hardware redundancy in order to demonstrate its effectiveness, with Section 7 concluding the paper.

2. SFC Framework Overview

There are two conceptual steps depicted in Figure 1 to achieve hardware-efficient reliable computing by the approach of selectively fortified computing (SFC): criticality probing and hardware redundancy allocation. Because there is a fundamental tradeoff between hardware cost and computing reliability, it is infeasible to construct a computing system that can tolerate all the faults of each of its elements. Therefore, the most critical hardware components in a target device need to be identified and their associated computation should be protected as a priority. In SFC, we develop methods to identify sensitive elements of the system whose failures might cause the most critical system failures, prioritize them based on their criticality to user perception and their associated hardware cost, and allocate hardware redundancy efficiently and accordingly.

3. Target Applications

Many mission-critical applications, such as multimedia processing, wireless communications, networking, and recognition, mining, and synthesis, possess a certain degree of inherent resilience, that is, when facing device or component failure, their overall performance degrades gracefully [19–22]. Such resilience is due to several factors. First, the input data to these algorithms are often quite noisy and nevertheless these applications are designed to handle them. Moreover, the input data are typically large in quantity and frequently possess significant redundancy. Second, the algorithms underlying these applications are often statistical or probabilistic in nature. Finally, the output data of these

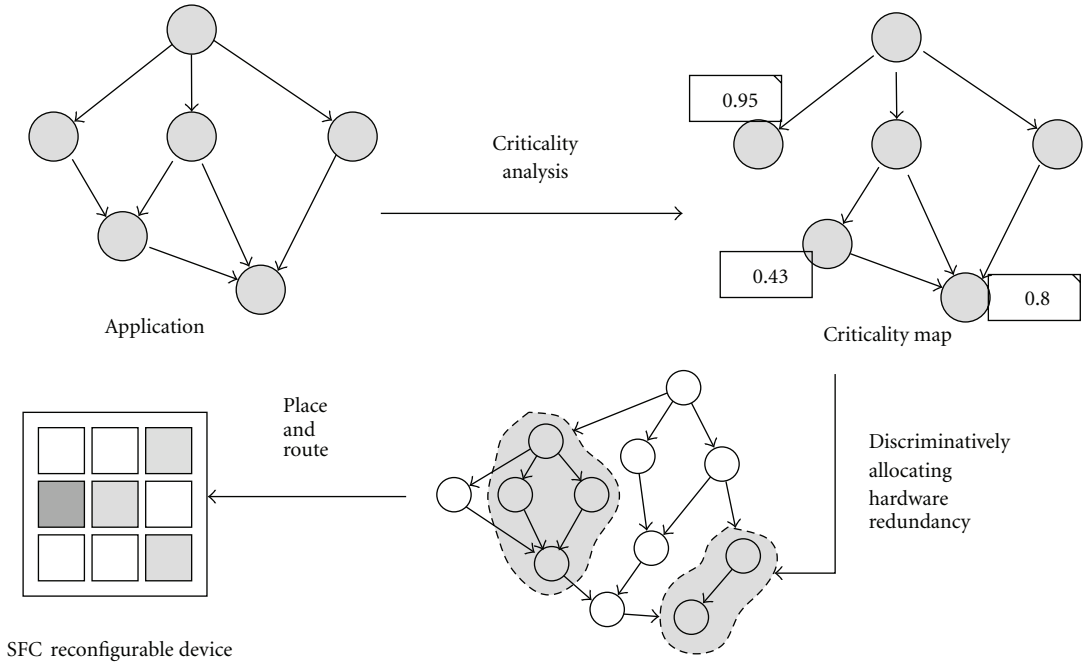


Figure 1: Conceptual flow of SFC (selectively fortified computing) methodology.

applications are mostly for human consumption, therefore some imperfections can be tolerated due to limited human perception. We believe that these applications are ideal SFC candidates because their overall system error resilience can potentially be significantly improved by selectively fortifying their key components. In this study, we illustrate our proposed SFC strategy by focusing on a highly efficient 720 p HD H.264 encoder [23]. We choose H.264 encoder because it is widely used in many mission-critical instruments. Moreover, H.264 contains a variety of computational motifs, from highly data parallel algorithms (motion estimation) to control intensive ones (CABAC) as depicted in Figure 2.

Our baseline FPGA implementation consists of five major functional components that account for more than 99% of the total execution time. Among these functional components, IME (integer motion estimation) finds the closest match for an image block from a previous reference image and computes a vector to represent the observed motion. While it is one of the most compute intensive parts of the encoder, the basic algorithm lends itself well to data parallel architectures. The next step, FME (fractional motion estimation), refines the initial match from integer motion estimation and finds a match at quarter-pixel resolution. FME is also data parallel, but it has some sequential dependencies and a more complex computation kernel that makes it more challenging to parallelize. IP (intraprediction) then uses previously encoded neighboring image blocks within the current image to form a prediction for the current image block. Next, in DCT/Quant (transform and quantization), the difference between a current and predicted image block is transformed and quantized to generate quantized coefficients, which then go through the inverse quantization and inverse transform to generate the reconstructed pixels. Finally, CABAC (context adaptive binary arithmetic coding) is used to entropy-encode the coefficients and other elements of the bit-stream. Unlike the previous algorithms, CABAC is sequential and control dominated. While it takes only a small fraction of the execution (1%), CABAC often becomes the bottleneck in parallel systems due to its sequential nature.

To appreciate the limited resilience characteristics of the HD H.264 encoder, we first implement all of five components using a standard HDL synthesis flow with Xilinx ISE Design Suite 13.0. Subsequently, errors are injected to four key components—IME (integer motion estimation), FME (fractional motion estimation), DCT/Quant (transform and quantization), and CABAC (context adaptive binary arithmetic coding) separately through VPI interfaces (see Section 5.2). These four components are quite diverse and representative in their computing patterns. More importantly, they together consume more than 90% of total execution load and more than 75% of total energy consumption [23]. For each of our target components X (I, II, III, and IV in Figure 2), we conduct 10000 fault injection experiments for each different error rate r according to the procedure outlined in Section 5.2. At a different redundancy ratio w_i, we repeat this procedure and measure the output deviation between the resulting image and its reference and quantify the result error as in Section 5.3. More details of our measurement procedure can be found in Section 5.

Before discussing the results presented in Figures 3 and 4, we formally define several key parameters. Some of them will be used in subsequent sections. Let h_i denote the hardware cost of implementing the component i in our target design. We define redundancy ratio $w_i = (h_i - h_{i,0})/(h_{i,0})$, where $h_{i,0}$ denotes the hardware cost of the component i without any redundancy.

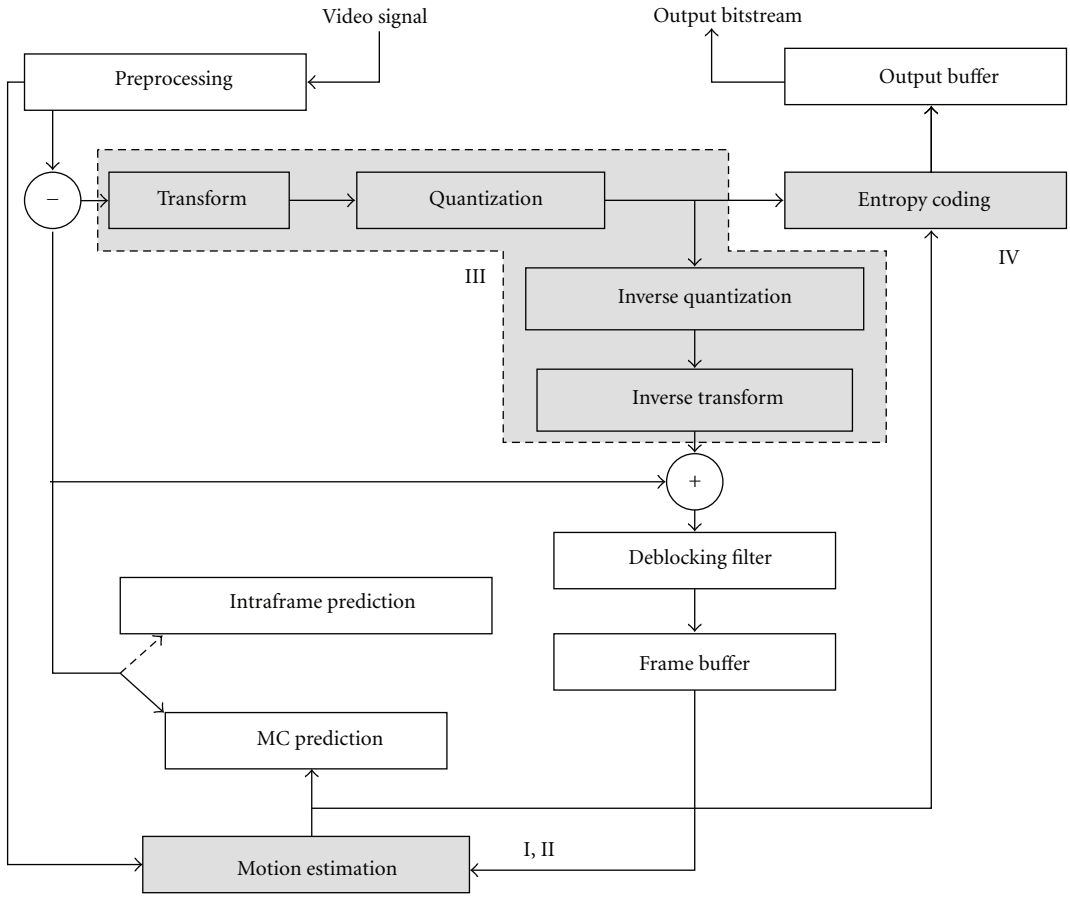


Figure 2: Block diagram of a standard H.264 encoder.

Figure 3 plots our measurement results for IME (integer motion estimation). Different curves represent output responses for different hardware redundancy ratios. Intuitively, the higher the value of w_i is, the more computing fortification the target device will possess. Roughly speaking, $w_i = 2$ is equivalent of a TMR (triple modular redundancy) unit. The values of w_i can be fractional because of partial redundancy as discussed in Section 6.1. In our experiments, error injection rate is measured as in error/gate/clock-cycle. We defer all discussions of error modeling and injection procedure to Section 5.2. Note in Figure 3 that errors injected at rate below 1×10^{-4} error/gate/clock-cycle do not impact the output quality by more than 1%, revealing the inherent resilience of such applications to rare data errors. However, error rates beyond 2.0×10^{-4} error/gate/clock-cycle impact image outputs significantly. We call this error rate as critical point. In our set of experiments, we injected errors by randomly selecting one of considered subcomponents and flipping one bit at a randomly chosen location. Our IME unit becomes totally dysfunctional after an average of 67 error injections when the error injection rate was as low as 1.13×10^{-4} error/gate/clock-cycle. These results confirm that, without proper system design, we cannot expect IME to produce useful results on unreliable hardware with relatively high error rate. Interestingly, as we increase hardware redundancy ratio, the critical point noticeably shifts towards high error rate. For our particular IME unit instance, diminish return of hardware redundancy quickly appears as w_i goes beyond 4, which is manifested by the flattening in its curve. Finally, at each w_i, the middle portion of each curve in Figure 3 exhibits a strong linearity, thus can be accurately approximated as a straight line.

In Figure 4, we fix error inject rate at 3.0×10^{-4} error/gate/clock-cycle and plot our measurement results for all four key components. The values of w_i can be fractional because of partial redundancy as discussed in Section 6.1. In general, different component responds quite differently to the same error rate. As w_i increases, the output error decreases consistently except for the component IV: CABAC. The relative small slopes of all curves in the middle portion again display early occurrence of diminished return of hardware redundancy. The most important finding of this experiment is to discover that DCT is far more sensitive to hardware redundancy than other components, while CABAC show exactly the opposite. Intuitively, this means that if facing hardware cost constrains, DCT should be assigned with a higher priority of hardware fortification and will have a larger benefit in improving the overall error resilience of the whole system. Finally, these results show that without

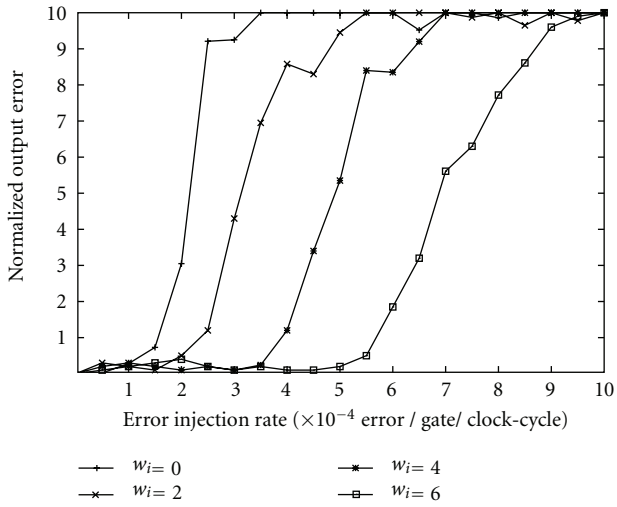


Figure 3: Output error versus error injection rate at different w_i for IME unit.

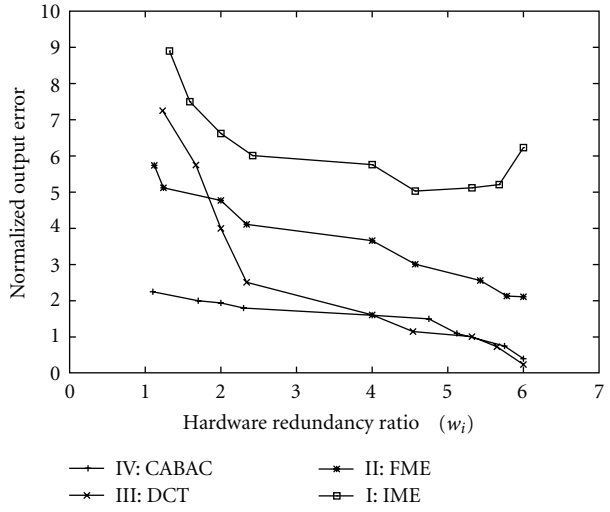


Figure 4: Output error versus hardware redundancy ratio w_i for different units.

computing fortification, the inherent error resilience for these key components, although exists, is quite limited.

4. Stochastic Formulation of Error Resilience

For a target design $\mathfrak{G}$ consisting of K functional components, each component C_i will be assigned a criticality value φ_i as defined in Section 5. Intuitively, a higher criticality value for a particular component C_i means more importance in its effect towards the overall correctness of final results. Moreover, we should allocate relatively more hardware resources to components with higher criticality in order to maximize the overall system's error resilience when constrained by total hardware usage.

Assume each component C_i consumes A_i units of hardware in the target application's final FPGA implementation. The unit used here can be either gate count or physical chip area. Furthermore, assuming all hardware defects are distributed in an i.i.d. manner, that is, all hardware defects are independent and identically distributed, all following a uniform distribution. Therefore, for any randomly occurring device error, the probability for one specific component C_i to be affected is $A_i/\sum_{j=1}^{K} A_j = g_i$. Note that in our proposed DFHS methodology, the random distribution of hardware effects can be in other form and does not limit the applicability of our proposed approach. In other words, if the randomly distributed hardware failures are more concentrated (i.e., occurring with higher probability) in certain component i, we can accordingly increase C_i to accommodate such changes.

Finally, we define the error probability of each component C_i to be $e_i = F(w_i, n_i)$, a function of two variables: w_i and n_i. w_i denotes the hardware redundancy ratio defined in Section 3, which quantifies how much hardware redundancy is built in the component C_i. Associated with each of the n functional components, there exist several choices of design alternatives having different hardware redundancy ratios w (as defined in Section 3). Typically, higher w_i values intuitively indicate more error resilience for a particular component at the price of high hardware usage. n_i denotes the number of hardware defects occurring in the component under consideration.

Given all the above definitions, further assuming error propagation within this target system is multiplicative, the objective of maximizing the overall error resilience of the target system $\mathfrak{G}$, under the attack of N randomly located hardware defects, can be formulated as minimizing the overall product of error probability and criticality among all its components:

$$E(\vec{w}, N) = \sum_{i=1}^{K} \varphi_i g_i e_i = \sum_{i=1}^{K} \varphi_i \left(\sum_{j=1}^{N} \left(\frac{A_i}{\sum_{k=1}^{K} A_k} \right)^j F(w_i, j) \right), \quad (1)$$

where $\vec{w}$ denotes $w_1, w_2, \ldots, w_K$. Clearly, constrained by a total hardware resource H, the problem is to determine which hardware redundancy ratio to select for each key component in order to achieve the greatest error resilience while keeping the total hardware cost within the allowable amount, taking into consideration each component's distinct criticality. This formulation of the problem leads to minimizing the total system error probability E:

$$\arg \min_{w_i, i \in [1,K]} E(\vec{w}, N), \qquad \text{s.t.} \sum_{i=1}^{n} A_i \leq H. \quad (2)$$

Interestingly, because both g_i and e_i depend on w_i, (1) is fundamentally a nonlinear multivariate optimization problem. More discussion on solving this optimization problem can be found in Section 6.

5. Criticality Analysis

Criticality analysis (CA) provides relative measures of significance of the effects of individual components on the

overall correctness of system operation. In essence, it is a tool that ranks the significance of each potential failure for each component in the system's design based on a failure rate and a severity ranking. Intuitively, because some parts within an application may be more critical than others, more computing resources should be allocated to these more critical components, that is, stronger computing "fortification," in order to achieve higher overall error resilience. In the framework of SFC, because the target computing device to be fortified is synthesized with a standard HDL flow and then realized (placed and routed) with a reconfigurable device such as an FPGA, criticality analysis is the key step to optimally allocate hardware redundancy within the target implementation.

Criticality analysis has been extensively studied within software domain [24–26], but quite rare in error-resilient computing device research. Our proposed approach to quantifying criticality directly benefits from mature reliability engineering techniques and scenario-based software architecture analysis. The key insight underlying our approach is that criticality analysis can be recast as a combined problem of uncertainty analysis and sensitivity analysis. Sensitivity analysis involves determining the contribution of individual input factors to uncertainty in model predictions. The most commonly used approach when doing a sensitivity analysis on spatial models is using Monte Carlo simulation. There are a number of techniques for calculating sensitivity indices from the Monte Carlo simulations, some more effective or efficient than others (see Figure 5).

Figure 6 depicts our procedure of criticality analysis. For each target system implemented with HDL, we first dissect it into multiple components. Often based on domain-specific knowledge of designer, a set of key components is selected and undergoes our criticality analysis. Fault modeling and injection mechanism are discussed in Section 5.2, and how to quantify the output errors is described in Section 5.3. To facilitate later discussion, we now define two metrics: sensitivity ψ_i and criticality φ_i for a specific component i at error rate e, the sensitivity value $\psi_i = \Delta y_i(e)/\Delta w_i$. Moreover, we define the criticality value φ_i as

$$\varphi_i = \mathbf{E}\left[\frac{(1/\psi_i)}{\sum_j \left(1/\psi_j\right)(e)}\right], \tag{3}$$

where $\mathbf{E}[\cdot]$ denotes the expected averaging over a wide range of error injection rates. In our case, we picked an i.i.d. uniform distribution for different error rates. ψ_i can be further approximated as a linear equation $\psi_i(e) = a_i(e) + b_i(e)w_i$. Both a_i and b_i values can be obtained through performing least square data fitting on our empirical data exemplified in Figure 4.

5.1. System Component Dissection. For a given target hardware implementation, before performing sensitivity and criticality analysis, the first step is to decompose the overall system into components. In this study, such system dissection typically follows naturally with individual module boundary. In most instances, block diagrams of logic implementation clearly shows the boundary between different system modules.

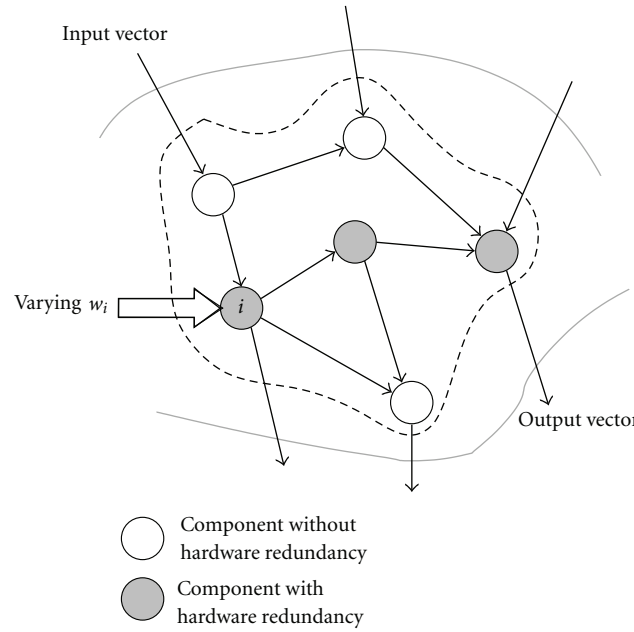


Figure 5: Conceptual diagram of sensitivity and criticality analysis.

We now use IME as an example to illustrate our FPGA implementation in detail. More importantly, we demonstrate how we perform component dissection in this SFC study. IME computes the motion vector predictor (MVP) of current macroblock (MB). There are many kinds of algorithms for block-based IME. The most accurate strategy is the full search (FS) algorithm. By exhaustively comparing all reference blocks in the search window, FS gives the most accurate motion vector which causes minimum sum of absolute differences (SAD) or sum of square difference (SSD). Because motion estimation in H.264/AVC supports variable block sizes and multiple reference frames, high computational complexity and huge data traffic become main difficulties in VLSI implementation [23]. Therefore, current VLSI designs usually adopt parallel architecture to increase the total throughput and solve high computational complexity. We implemented our IME unit in Verilog HDL language by closely following the block diagram in Figure 7. Among all units, PE array and four-input comparators perform computation on input pixel values and are most critical to the accuracy of final outputs. We totally have 41 four-input comparators and 138 PE units. Our approach allows either fortifying some or all of these subunits.

5.2. Fault Modeling and Injection. Much of our fault modeling and error injection mechanism is based on prior studies [27–29]. Our fault injection and evaluation infrastructure exploit Verilog programming interface (VPI) in order to quickly and flexibly inject emulated hardware defects into Verilog-based digital design in real time. Unlike many traditional fault injection methods, which either need to modify the HDL code or utilize simulator commands for

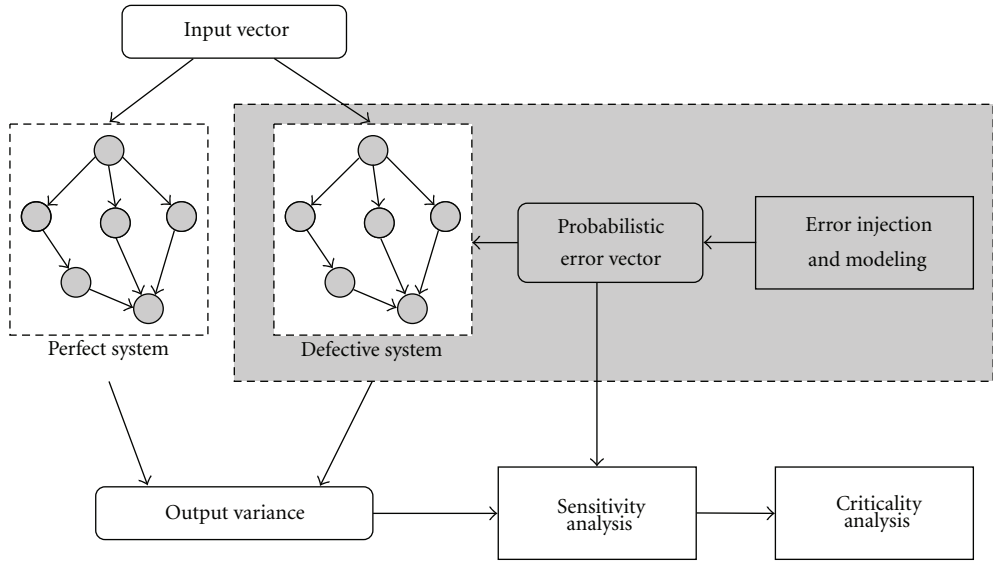


Figure 6: Flow chart of criticality analysis.

fault injection, our VPI-based approach is minimally invasive to the target design. Due to standardization of the VPI, the framework is independent from the used simulator. Additionally, it does not require recompilation for different fault injection experiments like techniques modifying the Verilog code for fault injection.

As shown in Figure 8, at the beginning of the simulation, the simulator calls the system with the command `$HDL_InjectError`, which generates one fault injection callback that is subsequently registered according to the specified arguments fault type and place. Each of such callbacks is specified by its time instant and duration and will be scheduled to execute at the injection instant and the end of injection. Then, when the inject instant reaches, the according callback will be executed by the simulator. This callback sets the conditions when the value modification on the specified place will occur. During the injection, the value of the injection place is modified by using VPI to control the Verilog simulator. At the end of fault, the injection callback is executed again to restore the normal value. One advantage of using VPI-based fault injection is that it can be applied to all VPI compliant Verilog simulators.

In order to make fault injection and evaluation feasible, we limit the fault models used in this work to single bit faults. A variety of fault models are defined in order to represent real physical faults that occur in integrated circuits (ICs). We totally consider four kinds of hardware fault. In this work, the bit flip representing the inverted value at the instant ts is differentiated from the toggling bit flip that toggles with original value. In principle, fault types can be readily extended with additional fault types.

5.3. Evaluating Fault Impact. The criticality analysis in SFC requires accurately quantifying the impact of injected hardware faults by comparing the result images with the reference ones. Unfortunately, computing the overall differences of pixel values will not detect and record the visual differences critical to human perception [30]. In this study, we feel other instances of comparing. In particular, the methodology presented here is based on a stochastic Petri-net (SPN) graph We instead calculate the Hausdorff distance to extract and record local and global features from both images, compare them, and define the percentage of similarity.

The Hausdorff distance measures the extent to which each point of a "model" set lies near some point of an "image" set and vice versa. Thus, this distance can be used to determine the degree of resemblance between two objects that are superimposed on one another. In this paper, we provide efficient algorithms for between all possible relative positions of a binary image and a model.

Given two finite point sets $A = a_1, a_2, \ldots, a_p$ and $B = b_1, b_2, \ldots, b_q$, the Hausdorff distance is defined as

$$H(A,B) = \max(h(A,B), h(B,A)), \tag{4}$$

where

$$H(A,B) = \max_{a \in A} \min_{b \in B} \|a - b\| \tag{5}$$

and $\|\cdot\|$ is some underlying norm on the points of A and B (e.g., the $L2$ or Euclidean norm). The Hausdorff distance $H(A,B)$ is the maximum of $h(A,B)$ and $h(B,A)$. Thus, it measures the degree of mismatch between two sets by measuring the distance of the point of A that is farthest from any point of B and vice versa. Intuitively, if the Hausdorff distance is d, then every point of A must be within a distance d of some point of B and vice versa. Thus, the notion of resemblance encoded by this distance is that each member of A be near some member of B and vice versa. Unlike most methods of comparing shapes, there is no explicit pairing of points of A with points of B (e.g., many points of A may be close to the same point of B). The function $H(A,B)$ can be trivially computed in time $\mathcal{O}(pq)$ for two-point sets

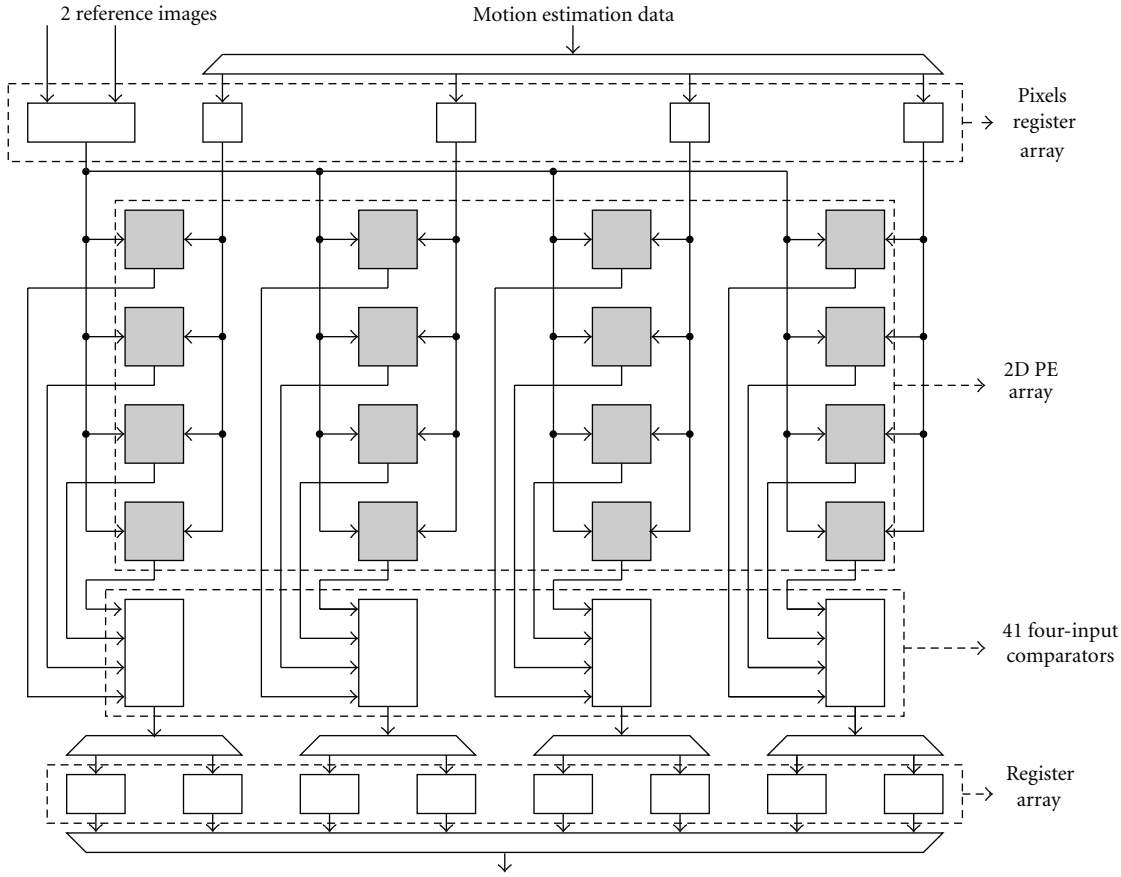


Figure 7: Block diagram of an integer motion estimator design with a macroblock parallel architecture and sixteen 2D PE arrays [23].

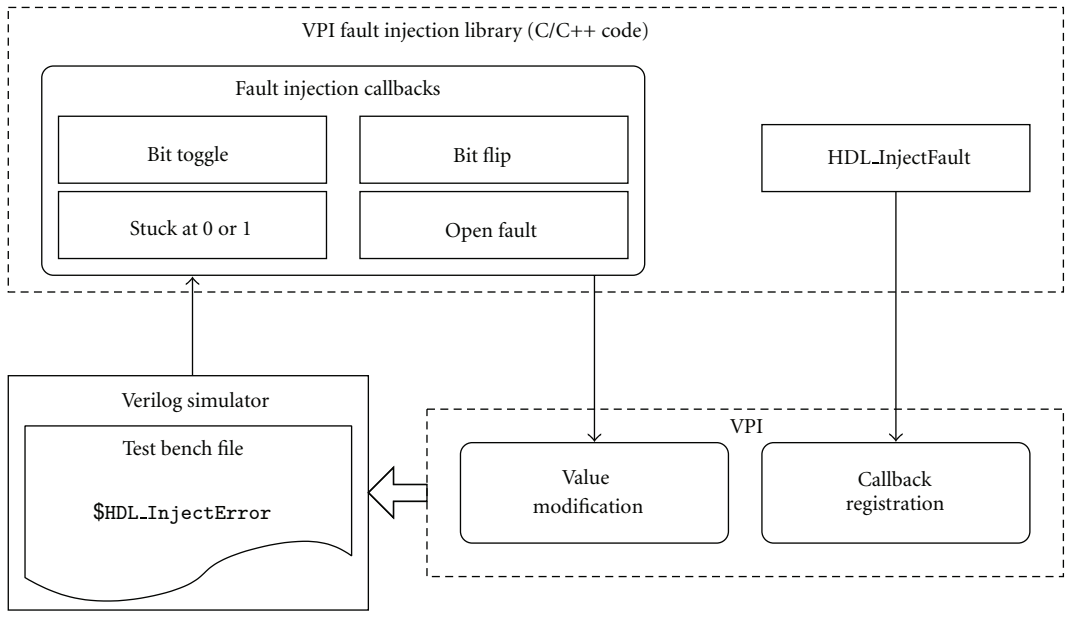


Figure 8: Block diagram of VPI simulation and fault injection flow [29].

of size p and q, respectively, and this can be improved to $\mathcal{O}((p+q)\log(p+q))$ [30].

6. Optimally Allocating Hardware Redundancy

Assuming that our target system consists of n functional components and error propagation within the target system is multiplicative. Associated with each of the n functional components, there exist several choices of design alternatives having different hardware redundancy ratio w (as defined in Section 3). Let e_i represent the ith component with a specified inherent error tolerance, and let $R_i(w)$ denote the derived error tolerance function of the ith component when the hardware redundancy ratio of e_i is w. Given the overall hardware cost limit of H, the problem is to determine which hardware redundancy ratio to select for each key component in order to achieve the greatest error resilience while keeping the total hardware cost within the allowable amount. This formulation of the problem leads to the maximization of system reliability R given by the product of unit error resilience

$$\arg \max_{w_i, i\in[1,n]} R = \prod_{i=1}^{n} R_i(w_i), \qquad \text{s.t.} \sum_{i=1}^{n} h_i \le H. \tag{6}$$

We now discuss the procedure to obtain both $R_i(w)$ and h_i and how to solve the above multivariate optimization problem.

6.1. Modeling Reliability and Its Hardware Cost. We consider two kinds of hardware redundancy: fractional redundancy and n-nary modular redundancy. First, for multiple subcomponents within one unit, we allow a fraction of these subcomponents to have hardware redundancy. For example, in Figure 7, a typical IME unit contains 138 PE units, each of which independently performs sum of absolute differences (SAD) and sum of square difference (SSD) operations. Due to stringent hardware costs, we normally cannot afford to build hardware redundancy into each of these PE units. Instead, our SFC framework allows part of these units to be fortified. In this study, the selection of these units is done randomly for a given fixed percentage. Second, our SFC system allows classic n-nary modular redundancy for chosen components. An n-tuple modular system is a natural generalization of the TMR (triple-modular redundancy) concepts [31], which utilizes $2n+1$ units together with a voter circuitry and its reliability is $R(NMR) = \sum_{i=0}^{n} \binom{N}{i} (1-R)^i R^{N-i}$ where the combinatorial notation $\binom{N}{i} = N!/((N-i)!i!)$.

Despite being conceptually simple, analytically computing the cost of hardware redundancy and its impact on the overall system's error resilience proves to be extremely difficult and inaccurate [31]. In this study, we opted to obtain both values empirically. As shown in Figure 3, $\varphi_i(w)$ can be approximated as $a_i + b_i w_i$, where a_i and b_i are the regression coefficients derived from experimental data. Furthermore, the hardware cost of implementing fortified component i is $(1+w_i)h_{i,0}$ according to w_i's definition.

6.2. Solving Allocation Problem. In this section, we determine the optimal allocation for a generic hardware design subject to a set of hardware redundancy with various costs, for example, solving the optimization problem defined in (6). Note $\arg\max_{w_i, i\in[1,n]} R$ equals $\arg\max_{w_i, i\in[1,n]} \ln R = \arg\max_{w_i, i\in[1,n]} \sum_{i=1}^{n} \ln R_i(w_i)$, where $\ln(a_i + b_i w_i)$ can be extended using Taylor series as

$$\begin{aligned}(a_{i,0} + b_{i,0}w_{i,0}) + \frac{d\ln(a_i + b_i w_i)}{dw_i}\Big|_{w_i = w_{i,0}} \cdot (w_i - w_{i,0}) + \cdots \\ \approx \alpha_i w_i + \beta_i,\end{aligned} \tag{7}$$

where $\alpha_i = (b_{i,0})/(a_{i,0} + b_{i,0}w_{i,0})$ and $\beta_i = ((a_{i,0} + b_{i,0}w_{i,0})^2 - b_{i,0}w_{i,0})/(a_{i,0} + b_{i,0}w_{i,0})$. In practice, $w_{i,0}$ should be chosen to be close to the optimal solution w_i^* of (6) in order to approximate more accurately. Therefore, (6) becomes

$$\arg \max_{w_i, i\in[1,n]} \sum_{i=1}^{n} \alpha_i w_i + \beta_i, \qquad \text{s.t.} \sum_{i=1}^{n} h_{i,0}(w_i + 1) \le H. \tag{8}$$

Equation (8) is a classic linear programming problem with bounded constraints and therefore can be readily solved by the simplex algorithm [32] by constructing a feasible solution at a vertex of the polytope and then walking along a path on the edges of the polytope to vertices with nondecreasing values of the objective function until an optimum is reached. Although in theory, the simplex algorithm can perform poorly for the worse case, in practice, the simplex algorithm is quite efficient and can be guaranteed to find the global optimum rather quickly [32]. In this study, we implemented a revised simplex method according to the algorithm outlined on page 101 of [33]. For all the simplex runs we have performed, they all complete successfully within 20,000 iterations.

7. Hardware Prototyping and Performance Comparison

Our prototype of H.264 encoder is based on the parallel architecture suggested by [23] and implemented with a Virtex-5 FPGA (XCV5LX155T-2) device. The functionality of this prototype is verified against a software reference design provided with MediaBench II Benchmark [34]. Table 1 lists the hardware usage of various components, in which IME and FME consume more than half of the total hardware usage and exhibits a clear data parallel computing pattern while CABAC is much smaller in gate usage but is totally control dominated.

For input image data, we used the base video input data set that has a 4 CIF video resolution (704×576) with a moderate degree of motion and is compressed to a bitrate providing medium/average video quality using a search window of ± 16 pixels for motion estimation [34]. As shown in Figure 9, we compare error resilience results between our baseline design with zero hardware redundancy, equal criticality solution, and the SFC design with optimally allocated redundancy. All three designs are subjected to

Table 1: Hardware usage of a H.264 encoder with a Virtex-5 FPGA (XCV5LX155T-2) device.

Unit	Logic elements	Memory bits	%	φ_i	w_i
IME	58960	4500	21.2	0.13	2.13
FME	79842	7600	29.0	0.10	1.89
IP	47823	3600	17.1	0.23	3.42
DCT	8966	750	3.2	0.43	5.78
CABAC	62451	2100	21.5	0.11	2.11
Other	10210	410	8	—	—

a wide range of error injection rates. To make a fair comparison, both the equal criticality solution and the SFC design are only allowed 30% additional hardware for design redundancy. For our baseline design, the error resilience is quite limited. As soon as the error injection rate exceeds 0.5×10^{-4} error/gate/clock-cycle, the normalized output error immediately jumps to more than 1, which roughly means more than 10% of output image data is corrupted and unrecognizable. We obtained the equal criticality solution by equalizing the ratio between hardware usage and criticality among all key components, that is, for components $i = 1, \ldots, n$, equalizing h_i/φ_i. Intuitively, this allocation scheme means assigning more hardware redundancy to more critical components, which impact more onto the system's overall error resilience. From Figure 9, we can see clearly that this criticality equalizing approach, despite conceptually simple, yields significant improvements. In fact, on average, the equal criticality solution improves the average error resilience by almost 1.5 times. Finally, after applying the method outlined in Section 6.2, the SFC solution produces even further improvements in error resilience. Now, the H.264 encoder can perform almost undisturbed even the error injection rate reaches beyond 4×10^{-4} error/gate/clock-cycle, an almost 4 and 2.5 times improvement over the baseline design, respectively.

As shown in Figure 9, for different number of hardware faults N and a total amount of available hardware redundancy W, the optimal hardware redundancy allocation is different. As a user of DFHS, when we make our FPGA design, we do not know exactly how many hardware errors will be encounter, therefore how do we make our choice of $\vec{w}$ to maximize its error resilience? Figure 10 presents three different allowable total hardware redundancy $W = 50\%$, 75%, and 85%. For each W, we plot two curves showing the total error resilience E as defined in (1) versus different numbers of hardware errors N. The lower curve is the result of optimal $\vec{w}$, whereas the upper curve is the worst E for each different N when choosing any of optimal $\vec{w}$ solutions at different Ns. This figure is useful because, for any specified maximumly allowed error resilience, the user can use Figure 10 to decide how much total hardware redundancy should be used and how many hardware failures the resulted design can tolerate. For example, if the maximumly allowed E is 0.12, using 50% total hardware redundancy and any optimal solution $\vec{w}$ when considering N ranges from 1 to 20,

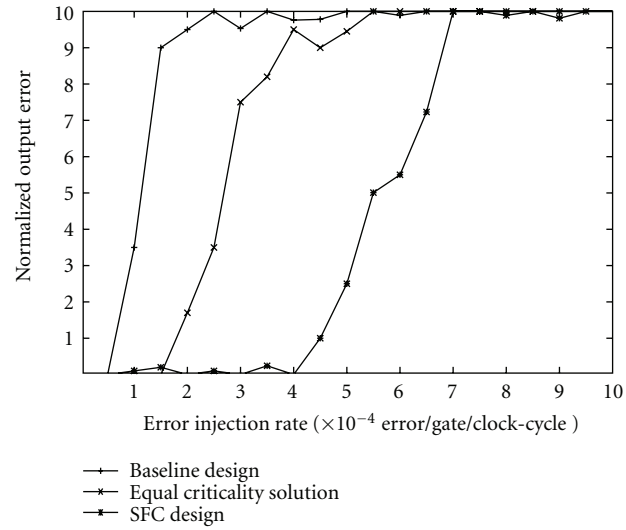


Figure 9: Error resilience comparison between baseline, equal criticality solution, and SFC design.

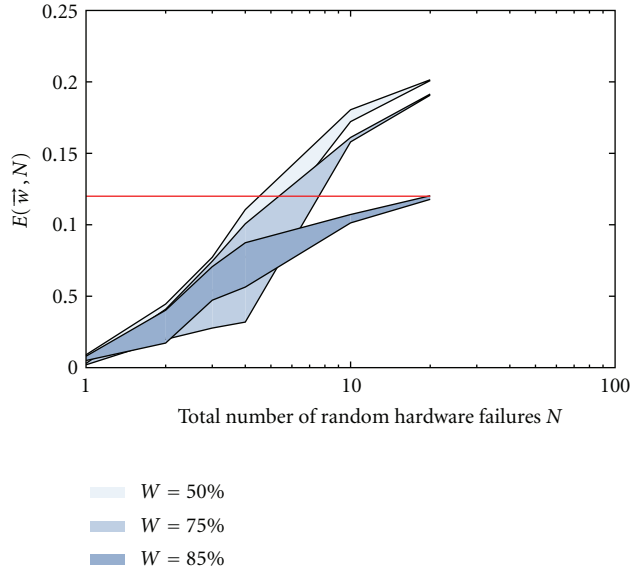


Figure 10: Design space exploration of DFHS for different allowed amount of hardware redundancy.

the resulted design can tolerate no more than 4 hardware failures. If the user desires to tolerate no more than 20 hardware failures ($E \leq 0.12$), the design needs to use at least 85% of total hardware redundancy.

8. Conclusion

This study introduces the concept of selectively fortified computing (SFC) for many mission-critical applications with limited inherent error resilience. The SFC methodology deviates significantly from the conventional approaches that heavily rely on static temporal and/or spatial redundancy and sophisticated error prediction or estimation techniques.

Instead of treating hardware or software redundancy as static objects and assuming prior knowledge of defects, the SFC approach focuses on providing much more hardware-efficient reliable computing through efficiently and strategically distributing hardware redundancy to maximize the overall achievable reliability.

To validate the feasibility to implement an error-resilient reconfigurable computing system with SFC, we have implemented a 720P H.264/AVC encoder prototype with an Virtex 5 FPGA device that operates at very high error rates. The goal was to show that even under such harsh error environment, our SFC system can still maintain high error tolerance without incurring excessive hardware redundancy. To achieve such robustness, various algorithms are developed to perform hardware sensitivity and criticality analysis, followed by an efficient heuristic methodology to optimally allocate hardware redundancy. Our experimental results from a 720P H.264/AVC encoder prototype implemented with an Virtex 5 device has clearly demonstrated the effectiveness of SFC operating under a wide range of error rates. Specifically, this H.264 prototype can tolerate the error rates as high as 18,000 errors/sec/RRC, which are emulated by hardware defects uniformly distributed through the whole computing device. Such high error rates extend far beyond radiation-induced soft error rates and may be caused by highly frequent erratic intermittent errors, process variations, voltage droops, and Vccmin. Compared with the unmodified reference design, the SFC prototype maintains 90% or better accuracy of output results and achieves more than four times of error resilience when compared with a baseline design. In principle, SFC is not restricted to video processing applications and can be applied to other general-purpose applications that are less resilient to errors.

Acknowledgments

This work was partially supported by DARPA System Center (Grant no. N66001-04-1-8916). Partial results of this study were published in the HASE2011 conference.

References

[1] P. Bose, "Designing reliable systems with unreliable components," *IEEE Micro*, vol. 26, no. 5, pp. 5–6, 2006.

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[3] W. Robinett, G. S. Snider, P. J. Kuekes, and R. S. Williams, "Computing with a trillion crummy components," *Communications of the ACM*, vol. 50, no. 9, pp. 35–39, 2007.

[4] S. L. Jeng, J. C. Lu, and K. Wang, "A review of reliability research on nanotechnology," *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 401–410, 2007.

[5] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 10)*, pp. 1011–1016, March 2010.

[6] N Haddad, R. Brown, T. Cronauer, and H. Phan, "Radiation hardened cots-based 32-bit microprocessor," in *Proceedings of the 5th European Conference on Radiation and Its Effects on Components and Systems (RADECS '99)*, pp. 593–597, Fontevraud , France, 1999.

[7] W. Heidergott, "SEU tolerant device, circuit and processor design," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 5–10, ACM, New York, NY, USA, June 2005.

[8] "QPro Virtex-II Pro 1.5V Platform FPGAs," http://www.xilinx.com/support/documentation/defenseqpro.htm/.

[9] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 76–83, 2003.

[10] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC V8 architecture," in *Proceedings of the International Conference on Dependable Systems and Networks (DNS '02)*, pp. 409–415, June 2002.

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

[12] N. Rollins, M. Wirthlin, M. Caffrey, and P. Graham, "Evaluating TMR techniques in the presence of single event upsets," in *Proceedings of the 6th Annual International Conference on Military and Aerospace Programmable Logic Devices*, pp. 63–70, September 2003.

[13] G. M. Swift, S. Rezgui, J. George et al., "Dynamic testing of xilinx virtex-II field programmable gate array (FPGA) input/output blocks (IOBs)," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, pp. 3469–3474, 2004.

[14] B. Pratt, M. Caffrey, J. F. Carroll, P. Graham, K. Morgan, and M. Wirthlin, "Fine-grain SEU mitigation for FPGAs using partial TMR," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, Article ID 4636895, pp. 2274–2280, 2008.

[15] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, 2004.

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

[17] S. Narayanan, G. V. Varatkar, D. L. Jones, and N. R. Shanbhag, "Computation as estimation: estimation-theoretic IC design improves robustness and reduces power consumption," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '08)*, pp. 1421–1424, April 2008.

[18] M. A. Breuer, "Multi-media applications and imprecise computation," in *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD '05)*, pp. 2–7, September 2005.

[19] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *Proceedings of the International Conference on Low Power Electronics and Design (ISLPED '99)*, pp. 30–35, ACM, New York, NY, USA, August 1999.

[20] D. Mohapatra, G. Karakonstantis, and K. Roy, "Significance driven computation: a voltage-scalable, variation-aware, quality-tuning motion estimator," in *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '09)*, pp. 195–200, ACM, New York, NY, USA, August 2009.

[21] R. Nair, "Models for energy-efficient approximate computing," in *Proceedings of the 16th ACM/IEEE International*

Symposium on Low Power Electronics and Design (ISLPED '10), pp. 359–360, ACM, New York, NY, USA, 2010.

[22] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "Ersa: error resilient system architecture for probabilistic applications," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pp. 1560–1565, European Design and Automation Association, Leuven, Belgium, 2010.

[23] Y.-L. S. Lin, C.-Y. Kao, H.-C. Kuo, and J.-W. Chen, *VLSI Design for Video Coding: H.264/AVC Encoding from Standard Specification to Chip*, Springer, 1st edition, 2010.

[24] P. G. Bishop, R. E. Bloomfield, T. Clement, and S. Guerra, "Software criticality analysis of cots/soup," in *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security (SAFECOMP '02)*, pp. 198–211, Springer, London, UK, 2002.

[25] C. Ebert, "Fuzzy classification for software criticality analysis," *Expert Systems with Applications*, vol. 11, no. 3, pp. 323–342, 1996.

[26] P. Anderson, T. Reps, and T. Teitelbaum, "Design and implementation of a fine-grained software inspection tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 721–733, 2003.

[27] R. Leveugle, D. Cimonnet, and A. Ammari, "System-level dependability analysis with RT-level fault injection accuracy," in *Proceedings of the 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '04)*, pp. 451–458, October 2004.

[28] J. Arlat, M. Aguera, L. Amat et al., "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[29] D. Kammler, J. Guan, G. Ascheid, R. Leupers, and H. Meyr, "A fast and flexible platform for fault injection and evaluation in Verilog-based simulations," in *Proceedings of the 3rd IEEE International Conference on Secure Software Integration Reliability Improvement (SSIRI '09)*, pp. 309–314, July 2009.

[30] N. G. Bourbakis, "Emulating human visual perception for measuring difference in images using an SPN graph approach," *IEEE Transactions on Systems, Man, and Cybernetics B*, vol. 32, no. 2, pp. 191–201, 2002.

[31] F. P. Mathur and A. Avižienis, "Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair," in *Proceedings of the Spring Joint Computer Conference (AFIPS '70)*, pp. 375–383, ACM, New York, NY, USA, May 1970.

[32] G. B. Dantzig and M. N. Thapa, *Linear Programming 1: Introduction*, Springer, Secaucus, NJ, USA, 1997.

[33] R. Darst, *Introduction to Linear Programming: Applications and Extensions*, Pure and Applied Mathematics, M. Dekker, 1991.

[34] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II video: expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301–318, 2009.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 537327, 16 pages
doi:10.1155/2012/537327

Research Article

A State-Based Modeling Approach for Efficient Performance Evaluation of Embedded System Architectures at Transaction Level

Anthony Barreteau, Sébastien Le Nours, and Olivier Pasquier

IREENA, EA1770, Université de Nantes, Polytech-Nantes, rue C. Pauc, Bât. IRESTE, BP 50609, 44000 Nantes, France

Correspondence should be addressed to Anthony Barreteau, anthony.barreteau@univ-nantes.fr

Received 29 June 2011; Revised 7 October 2011; Accepted 20 November 2011

Academic Editor: Philippe Coussy

Abstract models are necessary to assist system architects in the evaluation process of hardware/software architectures and to cope with the still increasing complexity of embedded systems. Efficient methods are required to create reliable models of system architectures and to allow early performance evaluation and fast exploration of the design space. In this paper, we present a specific transaction level modeling approach for performance evaluation of hardware/software architectures. This approach relies on a generic execution model that exhibits light modeling effort. Created models are used to evaluate by simulation expected processing and memory resources according to various architectures. The proposed execution model relies on a specific computation method defined to improve the simulation speed of transaction level models. The benefits of the proposed approach are highlighted through two case studies. The first case study is a didactic example illustrating the modeling approach. In this example, a simulation speed-up by a factor of 7,62 is achieved by using the proposed computation method. The second case study concerns the analysis of a communication receiver supporting part of the physical layer of the LTE protocol. In this case study, architecture exploration is led in order to improve the allocation of processing functions.

1. Introduction

In the consumer domain, current trends in embedded systems design are related to integration of high-performance applications and improvement of communication capabilities and mobility. Such functionalities have big influence on system architectures, significantly rising complexity of software, and hardware resources implemented. Typically, hardware resources are organized as multicore platforms consisting of a set of modules like fully programmable processor cores, standard interface modules, memories, and dedicated hardware blocks. Advances in chip technology will allow more resources to be integrated. Consequently, massively parallel architectures clustered by application category will be adopted [1]. Furthermore, in order to improve scalability of such platforms, network infrastructure represents a convenient solution to replace bus-based communication.

In this context, the process of system architecting consists in optimally defining allocation of system applications on platform resources and fixing characteristics of processing, communication, and memory resources according to functional and nonfunctional requirements. Functional requirements express what the designer wishes to implement whereas nonfunctional requirements are used to correctly tune parameters of related architecture. Typical nonfunctional requirements under consideration for embedded systems are timing constraints, power consumption, and cost. Exploration of the design space is led according to these requirements to identify potential architectures. Performances of candidate architectures are then evaluated and compared. In order to maintain short design time, fast exploration of the design space, and reliable evaluation of nonfunctional properties early in the development process

have then become mandatory to avoid costly design iterations. Due to increasing system complexity, evaluation of architecture performances calls for specific methods and tools to assist system architects in creating reliable models.

As reported in [2], the principles of the Y-chart model are usually followed for creation of models for performance evaluation of architectures. Following this approach, a model of the *application* is *mapped* onto a model of the considered *platform* and the resulting description is then evaluated through simulation or analytical methods. Analytical methods are used to perform formal analysis on architecture models. As stated in [3], these methods fit well if deterministic or worstcase behavior is a reasonable assumption for the architecture under evaluation. Simulation approaches rely on execution of a model of the architecture under evaluation with respect to a given set of stimuli. Compared to analytical approaches, simulation methods are required to investigate dynamic and nondeterministic effects in the system model. Simulation results are obtained in order to compare performances of limited set of candidate architectures, as illustrated in approaches presented in [4, 5]. The definition of efficient simulation-based approaches targets light modeling effort and improved simulation speed.

Simulation speed and accuracy are directly related to the level of abstraction considered to model the system architecture. On both application and platform sides, modeling of computation and modeling of communication can be strongly separated and defined at various abstraction levels. Among simulation-based approaches, the Transaction Level Modeling (TLM) approach has recently received wide interest in industrial and research communities in order to improve system design and its productivity [6]. This modeling approach provides facilities to hide unnecessary details of computation and communication (pins, wires, clock, etc.). The different levels of abstraction considered in transaction level models are classified according to time accuracy and granularity of computation and communication [6, 7]. Most of recent simulation-based approaches for performance evaluation rely on languages such as SpecC [8] or SystemC [9] to provide executable specifications of architectures, notably through the TLM2.0 standard promoted by OSCI [10]. Examples of recent TLM approaches are described in [6, 11, 12]. In such approaches, early evaluation of architecture performances is typically performed with transaction level models incorporating approximated time annotations about computation and communication. Architecture models are then simulated to evaluate usage of resources with respect to a given set of stimuli. However, TLM still lacks reference models used to facilitate creation and manipulation of performance models of system architectures and to provide light modeling effort. Besides, the achievable simulation speed of transaction level models is still limited by the amount of required transactions and integration of nonfunctional properties in performance models can significantly reduce simulation speed due to additional properties included. A quantitative analysis of the speed-accuracy tradeoff is presented in [13] through different case studies and different modeling styles.

This paper presents an approach for creation of efficient transaction level models for performance evaluation of system architectures. Compared to existing works, the main contribution is about a generic execution model used to capture evolution of nonfunctional properties assessed for performance evaluation. This execution model serves as a basic instance to create approximately timed models and it can be parameterized in order to evaluate various configurations of system architectures. Furthermore, it relies on a specific computation method proposed to significantly reduce the amount of required transactions during model execution and, consequently, to improve the simulation speed. This computation method is based on the decoupling between the description of model evolution, which is driven by transactions, and the description of nonfunctional properties. This separation of concerns leads to reducing the number of events in transaction level models. Simulation speedup can then be achieved by reducing the number of context switches between modules during model simulation. The proposed execution model and the related computation method have been implemented in a specific modeling framework based on the SystemC language. The considered modeling approach provides fast evaluation of architecture performances and then allows efficient exploration of architectures. The benefits of this approach are highlighted through two case studies. The modeling approach and the generic execution model are first illustrated through a didactic example. Then, the approach is illustrated through the analysis of two possible architectures of a communication receiver based on the Long Term Evolution (LTE) protocol.

The remainder of this paper is structured as follows. Section 2 analyzes the related modeling and simulation approaches for evaluation of performances of embedded systems. In Section 3, the proposed modeling approach is presented and related notations are defined. In Section 4, we describe the proposed generic execution model. The computation method used to improve the simulation speed of models is detailed. Also, we describe the implementation in a specific simulation framework. Section 5 highlights the benefits of the contributions through two separated case studies. Finally conclusions are drawn in Section 6.

2. Related Work

Performance evaluation of embedded systems has been approached in many ways at different levels of abstraction. A good survey of various methods, tools, and environments for early design space exploration is presented in [3]. Typically, performance models aims at capturing characteristics of architectures and they are used to gain reliable data of resource usage. For this purpose, performance evaluation can be performed without considering a complete description of the application. In simulation-based approaches, this abstraction enables efficient simulation speed and favors early performance evaluation. Workload models are then defined to represent computation and communication loads applications caused on platforms when executed. Workload models are mapped onto platform models and the resulting

architecture models are simulated to obtain performance data. Related works mainly differ according to the way application and platform models are created and combined.

The technique called trace-driven simulation has been proposed for performance analysis of architectures in [14]. Following this technique, the execution of the platform model is driven by traces from the execution of the application model. A trace represents the communication and computation workloads imposed to a specific resource of the platform. In this approach, application is described in the Kahn Process Network (KPN) model to expose parallelism. Platform model is made of processing resources and communication interfaces. Each processing element is described by the number of cycles each instruction takes when executed and communication is characterized by buffer size and transfer delay. The Sesame approach [4] extends this simulation technique by introducing the concept of virtual processors. Virtual processor is used to map a trace to a transaction level model of the platform. In this approach, candidate architectures are first selected using analytical modeling and multiobjective optimization according to parameters such as processing capacities, power consumption, and cost. Potential solutions are then simulated at transaction level using SystemC. A similar approach is considered in [5].

Trace-driven simulation is also addressed in the TAPES approach [15]. In this approach, traces abstract the description of functionalities for each resource and they are defined as a sequence of processing delays interleaved with transactions. Depending on the allocation decision, each processing resource of the architecture contains one or more traces that are related to different processing sequences. Shared resources like memory or bus imply generation of additional delays as consequence of competing accesses. Architecture specification is then translated in SystemC and obtained description is simulated by triggering traces required for processing particular data in the respective resources of the architecture.

Approaches presented in [16, 17] describe the combined use of UML2 and SystemC for performance evaluation. Approach presented in [16] gets a strong emphasis on streaming data embedded systems. A specific metamodel is defined to guide designers in the creation process of application and platform models. UML2 activity diagram and class diagram are used to capture workload and platform models. Stereotypes of the UML2 MARTE profile [18] are used for nonfunctional properties and allocation description. Once allocation defined, SystemC description is generated automatically and simulated to obtain performance data. In [17], system requirements are captured as a layered model defined at the service level. Workload models are mapped onto the platform models and the resulting system model is simulated at transaction level to obtain performance data. A specific attention is paid about the way workload models are obtained and three load extraction techniques are proposed: analytical, measurement based, and source code based.

The proposed design framework in [19] aims at evaluating nonfunctional properties such as power consumption and temperature. In this approach, description of temporal behavior is done through a model called communication dependency graph. This model represents a probabilistic quantification of temporal aspects of computation as well as an abstract representation of the control flow of each component. This description is completed by models of nonfunctional properties characterizing the behavior of dynamic power management. Simulation is then performed in SystemC to obtain an evaluation of time evolution of power consumption.

Our approach mainly differs from the above as to the way system architecture is modeled and models of workload are defined. In our approach, architecture specification is done graphically through a specific activity diagram notation. The behavior related to each elementary activity is captured in a state-action table. So, in our approach, models of workload are expressed as finite-state machines in order to describe the influence of application when executed on platform resources. The resulting architecture model is then automatically generated in SystemC to allow simulation and performance assessment. Compared to the related works, a specific attention is paid in order to reduce the time required to create models. In our works, the architecture model relies on a generic execution model proposed to facilitate the capture of the architecture behavior and the related properties. A specific method is also defined to improve the simulation time of models. The proposed modeling approach and the performance evaluation method are close to one presented in [15]. Similarly, our modeling approach considers the description of architecture properties in the form of traces for each resource. However, compared to [15], the description of the architecture model is significantly improved by the use of the proposed execution model. This reference model differs from one presented in [16] because it is not limited to streaming data application. Compared to approaches presented in [4, 16] the architecture model relies on a state diagram notation which allows modeling time-dependent behavior. A similar notation is also adopted in [17, 19] but architecture model does not rely on any reference model and requires specific development. Then, the aim of the contribution is to provide a reference model to build performance models of architectures with light modeling effort. Furthermore, this reference model makes use of the SystemC language to improve simulation speed of created performance models.

3. Considered Modeling Approach for Performance Evaluation of System Architectures

3.1. Graphical Notation. The modeling approach presented in this section aims at creating models in order to evaluate resources composing system architectures. As previously discussed, model of system architecture does not require complete description of system functionalities. In the considered approach, the architecture model combines the structural description of the system application and the description of nonfunctional properties relevant to considered hardware and software resources. The utilization

of resources is described as sequences of processing delays interleaved with exchanged transactions. This approach is illustrated in Figure 1.

The lower part of Figure 1 depicts a typical platform made of communication nodes, memories, and processing resources. Processing resources are classified as processors and dedicated hardware resources. In Figure 1, F_{11}, F_{12}, and F_2 represent the functions of the system application. They are allocated on the processing resources P_1 and P_2 to form the system architecture. For clarity reason, communications and memory accesses induced by this allocation are not represented. The upper part of the figure depicts the structural and the behavioral modeling of the system architecture. The structural description is based on an activity diagram notation inspired from [20]. This notation is close to the UML2 activity diagram. Each activity A_i represents a function, or a set of functions, allocated on a processing resource of the platform. As for example, activity A_{11} models the execution of function F_{11} on processor P_1. Relations M_i between activities are represented by single arrow links. Transactions are exchanged through relations and they are defined as data transfer or synchronization between activities. Following the adopted notation, transactions are exchanged in conformity with the rendezvous protocol. The graphical notation adopted for the description of activity behavior is close to the Statechart [21]. Behavior related to each elementary activity models the usage of resources by each function of the system application. Behavior exhibits waiting conditions on input transactions and production of output transactions. In the notation adopted one important point is about the meaning of temporal dependencies. Here, transitions between states s_i are expressed as waiting transactions, or logical conditions on internal data. A specific data value may be a time variable which evolves naturally. This data is denoted by t in Figure 1. The amount of processing and memory resources used is expressed according to the allocation of functions. In Figure 1, the use of processing resources due to the execution of function F_2 on P_2 is modeled by the evolution of the parameter denoted by Cc_{A2}. For example, Cc_{A2} can be defined as an analytical expression to give the number of operations related to the execution of function F_2. Value of Cc_{A2} can be influenced by data associated to the transaction received through relation M_3.

3.2. Formal Definition. A notation similar to [16] is adopted to define more formally elements of the performance model. We define a system architecture SA modeled according to the considered approach as a tuple

$$SA = (A, M, \mathrm{TC}), \tag{1}$$

where A is the set of activities that compose the architecture model, M is the set of relations connecting activities, and TC is the set of timing constraints related to activities. Due to the communication protocol considered in our approach, no data is stored through relations. This implies that activities can also be used to model specific communication and memory effects such as latency, throughput, or bus contentions due to competing accesses. Relations are unidirectional and each relation $M_i \in M$ is defined as

$$M_i = (A_{\mathrm{src}}, A_{\mathrm{dst}}), \tag{2}$$

where $A_{\mathrm{src}} \in A$ corresponds to the emitting activity and $A_{\mathrm{dst}} \in A$ is the receiving activity. An activity $A_i \in A$ is defined as

$$A_i = (S, M_!, M_?, F), \tag{3}$$

where S is the set of states used for activity description, $M_! \in M$ is the set of input relations, $M_? \in M$ is the set of output relations, and F is the set of transitions used to describe the evolution of A_i. States $S_i \in S$ decompose the evolution of activity in terms of time intervals. In the following, we consider that these intervals only exhibit the use of processing resources. Additional properties such as memory resources could also be addressed. A transition $F_i \in F$ is then defined as

$$F_i = (E, M_{\mathrm{out}}, \mathrm{Cc}), \tag{4}$$

where E is the set of conditions implying a transition of state, $M_{\mathrm{out}} \in M_!$ is the set of output transactions related to the considered transition, and Cc is the computational complexity inferred by the state S_i following the transition F_i. Additional properties such as the memory cost could also be considered. Cc can be influenced by data associated to transactions. Conditions $E_i \in E$ can be defined as a combination of waiting conditions on a set of input relations, time conditions, and logical conditions.

Based on this set of rules, the behavior of an activity can be captured using a state-action table notation as defined in [22]. Compared to a finite state machine sensitive to the evolution of a clock signal, this table gives the evolution of activities defined at the transaction level. The description related to activity A_2 in Figure 1 is presented in Table 1.

Table 1: State-action table for specification of activities.

Current state	Next state		Actions	
	Condition	State	Condition	Action
s_0	M_3	s_1	—	$\mathrm{Cc}_{A2} = 0$
s_1	$k < N$ AND $t = T_j$	s_0	—	$\mathrm{Cc}_{A2} = \mathrm{Cc}_{s1}$
	$k = N$ AND $t = T_j$	s_2		
s_2	$t = T_k$	s_3	—	$\mathrm{Cc}_{A2} = \mathrm{Cc}_{s2}$
s_3	$t = T_l$	s_0	—	$\mathrm{Cc}_{A2} = \mathrm{Cc}_{s3}$ M_4

The first column specifies the set of current states. The second column specifies the next states and the conditions under which the activity will move to those states. The third column specifies assignment of properties under study and production of output transactions. Other actions are not depicted in Table 1 for clarity reason. Conditions under which these assignments occur are also included. This means that state-action table is able to capture both Moore and Mealy types of finite state machines. It should be noted

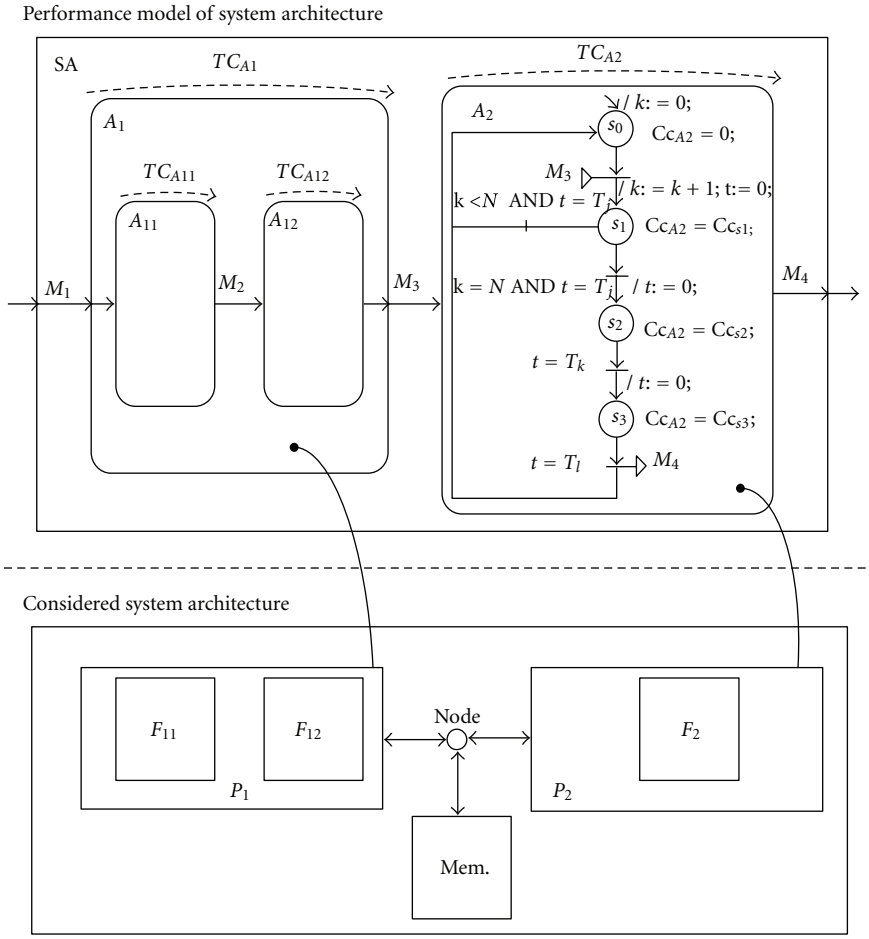


Figure 1: Considered modeling approach for performance evaluation of system architectures.

that assignment of properties is done during current state whereas production of transactions is done only on state transition.

In our approach, state-action tables are used to capture the behavior and the time properties related to each elementary activity. As a result, captured behavior and related time properties depend on the considered allocation of the application.

3.3. Temporal Behavior. Using languages as SystemC, evolution of the model can be analyzed according to the simulated time supported by the simulator. In the following, the simulated time is denoted by t_s. The obtained evolution for activity A_2 is depicted in Figure 2 for internal parameter N set to 3.

As depicted in Figure 2, Cc_{s1} operations are first executed for a duration set to T_j after reception of transaction M_3. Once N transactions received ($N = 3$ in this case), Cc_{s2} operations are executed for a duration set to T_k. The production of transaction M_4 is done once state s_3 finished, after a duration set to T_l. Such an observation gives

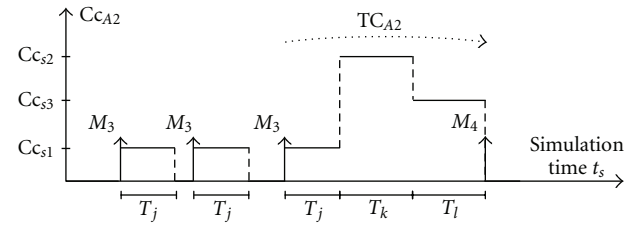


Figure 2: Time evolution of parameter Cc_{A2} according to the simulated time.

indication about the way processing resources are used when architecture executes.

The time properties used are directly influenced by the characteristics of the processing resources and by the characteristics of the communication nodes used for transaction exchange. These properties could be provided by estimations, profiling existing codes, or source code analysis, as illustrated in [17]. In the following the illustration of the modeling

approach will be done considering estimations given by analytical expressions.

Furthermore, the temporal behavior related to each activity is relevant to the function allocation. In case of a single processor architecture, functions are executed sequentially or according to a specific scheduling policy. In case of a multiprocessor architecture, behaviors should express the available parallelism to execute functions. In the following, the different allocations will be obtained by modifying the behavior related to each activity.

Moreover, in the notation adopted in Figure 1, TC_{A1} and TC_{A2} denote the time constraints to be met by the activities A_1 and A_2. TC_{A2} expresses the time constraint to satisfy when function F_2 is executed by P_2. In Figure 2, durations T_j, T_k, and T_l are set in order to meet TC_{A2}. TC_{A1} is the time constraint to satisfy when F_{11} and F_{12} are successively executed by P_1. Values considered for TC_{A11} and TC_{A12} can be modified in order to consider different time repartitions for the execution of F_{11} and F_{12}.

Following this modeling approach, resulting model incorporates evolution of quantitative properties defined analytically and relevant to the use of processing resources, communication nodes, and memories. Using languages as SystemC, created models can then be simulated to evaluate the time evolution of performances obtained for a given set of stimuli. Various platform configurations and function allocations can be compared considering different descriptions of activities. In the following, a generic execution model is proposed to efficiently capture the behavior of activities and then the evolution of nonfunctional properties assessed. This reference model facilitates creation of transaction level models for performance evaluation. State-action tables are then used to parameterize instances of the generic execution model.

4. Proposed Generic Execution Model for Performance Evaluation

4.1. Behavioral Description of Proposed Generic Execution Model. A generic execution model is proposed to describe the behavior of activities and then to easily build architecture models. This execution model expresses the reception and production of transactions and the evolution of resources utilization. Its description can be adapted according to the number of input and output relations and according to the specification captured in the associated state-action table. The proposed execution model is illustrated in Figure 3 for the case of i input relations and j output relations.

As depicted in Figure 3, behavior exhibits two states. State *Waiting* is for reception of transactions. Selection of input relation is expressed through parameter Select_$M_!$. Once a transaction has been received through a relation $M_{!i}$ activity goes to state *Performance analysis*. When *Select_*$M_!$ is set to 0 no transaction is waited. The time conditions related to activity evolution are represented by parameter T_s. T_s is updated during model execution and its value can be influenced by data associated to transactions. This is represented by action *ComputeAfter* $M_!$ and it is depicted

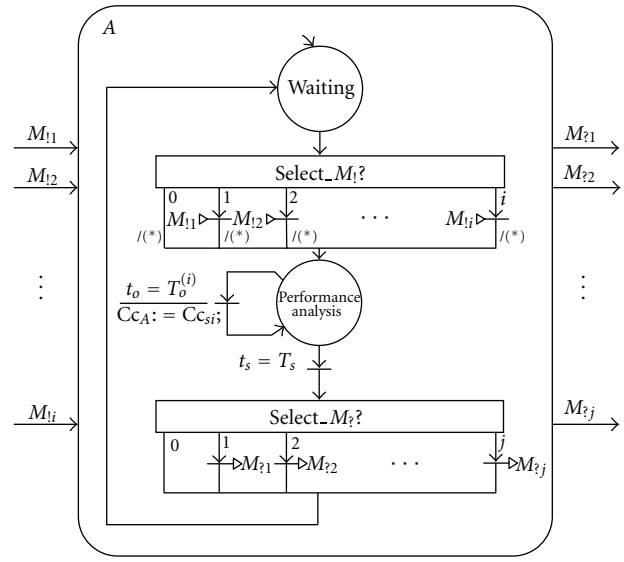


Figure 3: Proposed execution model for activity description.

in Figure 3 by symbol (*). Time conditions are evaluated according to the simulated time t_s. The parameter *Select_*$M_?$ is used to select the output relation when a transaction is produced. When *Select_*$M_?$ is set to 0 no transaction is produced. *Select_*$M_!$ and *Select_*$M_?$ are successively updated in state *Waiting* during model execution to meet the specified behavior.

Evolution of assessed property Cc_A is done in state *Performance analysis*. Successive values are denoted in Figure 3 by Cc_{si}. These values can be evaluated in zero time according to the simulated time t_s. This means that no SystemC wait primitives are used, leading to no thread context switches. Resulting observations correspond to values Cc_{si} and associated timestamps, denoted in Figure 3 by $T_o{}^{(i)}$. Timestamps are also local variables of the activity and their values are considered relatively to what we call the observed time, denoted in Figure 3 by t_o. The observed time t_o is a local time used for property evolution, whereas t_s is the simulated time used as a reference by the simulator. Using this technique, the evolution of considered property Cc_A can be computed locally between successive transactions. Details are given in the next section about how this computation technique can significantly improve simulation time of model execution.

The application of this modeling style to the activity specified in Table 1 is illustrated in Figure 4.

Figure 4 depicts a specific instance of the generic execution model for one input relation and one output relation. As indicated in Table 1, the activity evolution depends on relation M_3, logical conditions, and specific time conditions. The evolution obtained for activity A_2 using the execution model is illustrated in Figure 5. The reception and production of transactions are depicted according to the simulated time t_s. Evolution of property assessed is represented according to the observed time t_o.

The Figure 5(a) depicts the evolution of activity A_2 when transactions occur and according to the simulated time

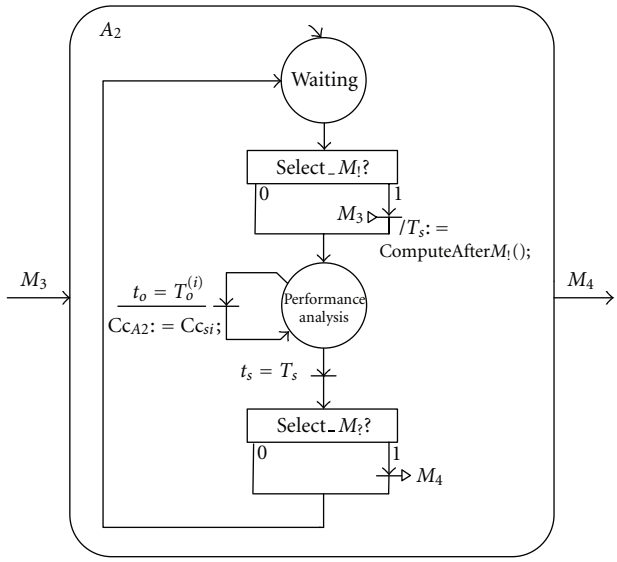


Figure 4: Application of proposed execution model to a specific activity description.

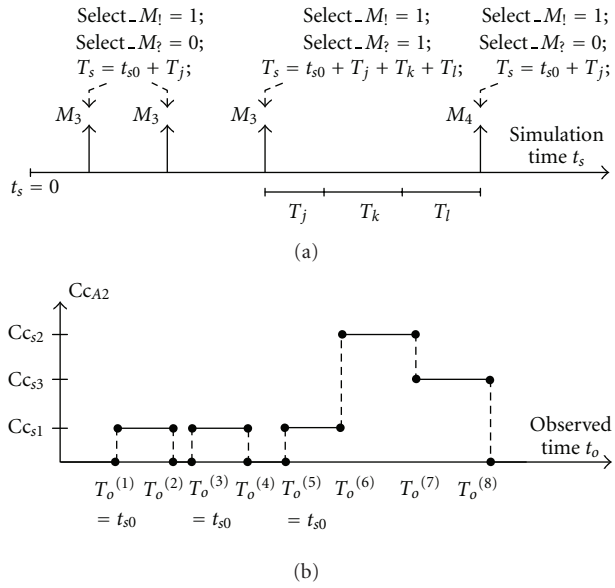


Figure 5: Evolution of activity A_2 according to (a) the simulation time t_s and (b) the observed time t_o.

t_s. Values of *Select_M*$_!$, *Select_M*$_?$, and T_s are successively updated to meet specified behavior. t_{s0} denotes the current simulated time. The evolution of Cc$_{A2}$ according to the observed time t_o is represented in the Figure 5(b). Once transactions received, successive values of Cc$_{A2}$ and timestamps are computed relatively to the arrival time of transaction. For example, in Figure 5, when the third transaction is received successive values of Cc$_{A2}$ and timestamps values $T_o{}^{(6)}$, $T_o{}^{(7)}$, and $T_o{}^{(8)}$ are defined locally. The evolution of Cc$_{A2}$ between the reception of the third transaction through relation M_3 and the production through M_4 does not imply use of SystemC wait primitive and evolution is then obtained in zero time according to the simulated time t_s.

Next section details this computation method and how it can be applied to improve the simulation speed of performance models.

4.2. Proposed Computation Method of Nonfunctional Properties of System Architectures. As previously discussed, the simulation speed of transaction level models can be significantly improved by avoiding context switches between threads. The computation method described in this section relies on the same principle as temporal decoupling supported by the loosely timed coding style defined by OSCI. Using this coding style, parts of the model are permitted to run ahead in a local time until they reach the point when they need to synchronize with the rest of the model. The proposed method can be seen as an application of this principle to create efficient performance models. This method makes it possible to minimize the number of transactions required for the description of properties assessed for evaluation of performances. Figure 6 illustrates the application of proposed computation method to the example considered in Figure 1.

Figure 6 depicts two possible modeling approaches. The upper part of the figure corresponds to a description with 3 successive transactions. Delays between successive transactions are denoted by Δt_1 and Δt_2. In this so-called transaction-based modeling approach, the property Cc$_{A2}$ evolves each time a transaction is received and a similar observation to one depicted in Figure 2 can be obtained. The lower part of the figure considers the description of the activity A_2 considering application of the computation method. Here, we focus only on explanation of the computation method and the application of the generic execution model is not fully represented in Figure 6 for clarity reason. Compared to the situation depicted in the upper part of the figure, only one transaction occurs and the content of the transaction is defined at higher granularity. However, the evolution of property Cc$_{A2}$ can be preserved by considering a decoupling with the evolution of activity A_2. In that case, duration T_s corresponds to the time elapsed between the first transaction and the production of transaction through relation M_4. This value is locally computed relatively to the arrival time of input transaction M_3 and it defines the next output event. In Figure 6, this is denoted by action *ComputeAfter* $M_!$. The time condition is evaluated during state s_0 according to the simulated time t_s. This computation supposes estimates about values of Δt_1 and Δt_2.

The evolution of property Cc$_{A2}$ between two external events is done during state s_0. Successive values, denoted in Figure 6 by Cc$_{si}$, are evaluated in zero time according to the simulated time. This means that no SystemC wait primitives are used, leading to no thread context switches. Resulting observations correspond to values Cc$_{si}$ and associated timestamps $T_o{}^{(i)}$. Timestamps values are considered relatively to what we call the observed time, denoted in Figure 6 by t_o. Using this technique, evolution of the considered property can then be computed locally between successive

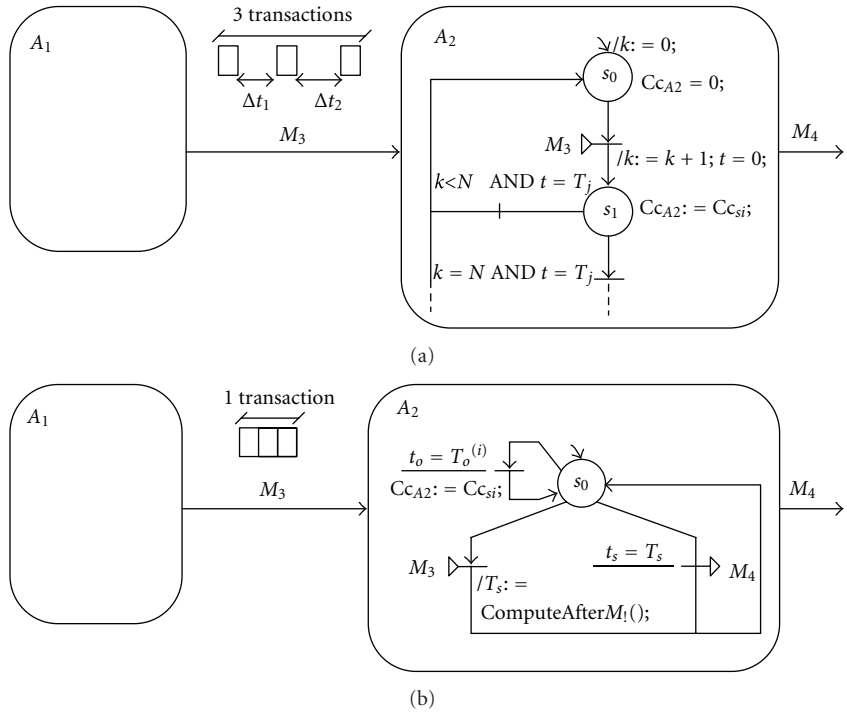


Figure 6: Comparison of two modeling approaches (a) a transaction-based modeling approach and (b) a state-based modeling approach.

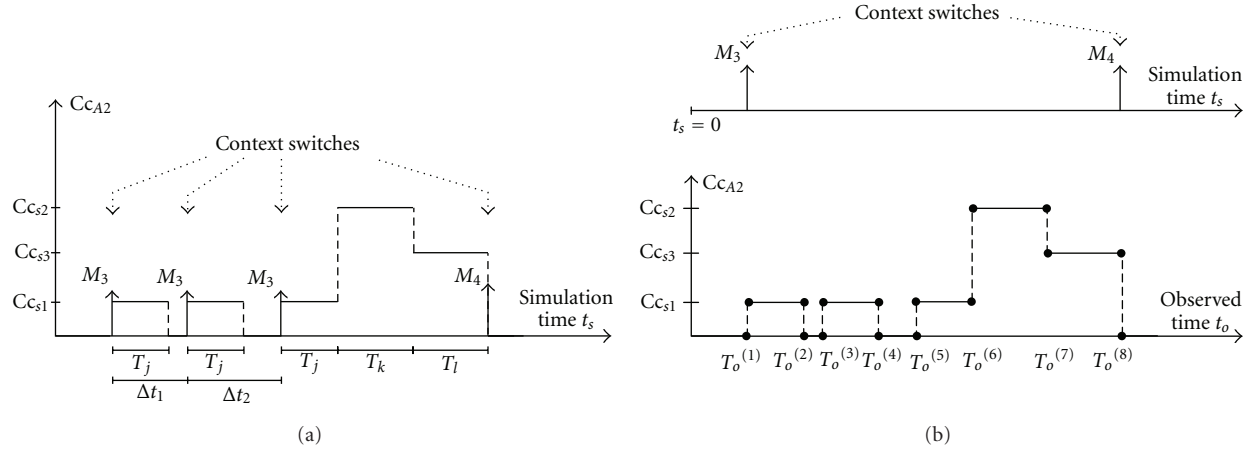


Figure 7: Evolution of property Cc_{A2} considering (a) a transaction-based modeling approach and (b) the proposed state-based modeling approach.

transactions. Compared to the previous transaction-based approach, the second modeling approach can be considered as a state-based approach. Assessed properties are then locally computed in the same state which reduces the number of required transactions. Figure 7 represents time evolution of property Cc_{A2} considering the two modeling approaches illustrated in Figure 6.

The Figure 7(a) illustrates the time evolution of property Cc_{A2} with 3 successive input transactions. During simulation of the model each transaction implies a thread context switch between activities and Cc_{A2} evolves according to the simulated time t_s. On the Figure 7(b), successive values of property Cc_{A2} and associated timestamps are computed at the reception of transaction M_3. Evolution is depicted according to the observed time t_o. Improved simulation time is achieved due to the amount of context switches avoided. More generally, we can consider that when the number of transaction is reduced by a factor of N, a simulation speedup by the same factor can be achieved. This assumption has been verified through various experimentations presented in [23]. Here the achievable simulation speedup is illustrated in the first case study considered in this paper.

The proposed generic execution model makes use of this computation method to provide improved simulation time of performance models. In order to validate this modeling style, we have considered the implementation

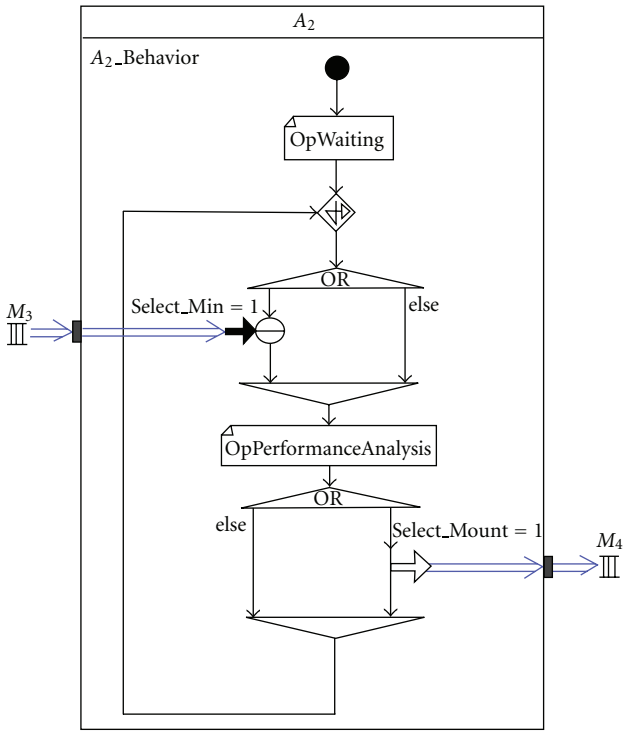


FIGURE 8: Graphical modeling in the CoFluent Studio framework to implement the proposed execution model.

of the proposed execution model in a specific modeling framework.

4.3. Implementation of the Generic Execution Model in a Specific Framework. The proposed execution model has been implemented in the framework CoFluent Studio [24]. This environment supports creation of transaction level models of system applications and architectures. Graphical models captured and associated codes are automatically translated in a SystemC description. This description is then executed to analyze models and to assess performances. We used the so-called *Timed-Behavioral Modeling* part of this framework to create models following the considered approach. Figure 8 illustrates a possible graphical modeling to implement the proposed execution model. It corresponds to the specific case illustrated in Figure 4 with one input relation and one output relation.

In Figure 8, the selection of input relation is represented by an alternative statement. Selection is denoted by parameter *Select_Min*. Transactions are received through relation M_3. Parameter T_s is expressed through the duration of the operation *OpPerformanceAnalysis*. This operation is described in a sequential C/C++ code to define computation of local variables and display. The Algorithm 1 example corresponds to part of required instructions to obtain observations depicted on the Figure 7(b).

The procedure `CurrentUserTime` is used in CoFluent Studio to get the current simulated time. In our case, it is used to get the reception time of transactions and then to compute values of timestamps. The procedure `CofDisplay` is used to display variables in a $Y = f(X)$ chart. In our case, it is used to display studied properties according to the observed time. The keyword `OpDuration` defines the duration of operation *OpPerformanceAnalysis*. It is evaluated according to the simulated time supported by the SystemC simulator. Successive values of Cc_{A2} and timestamps can be provided by estimations and they can be computed according to data associated to transactions. This implementation can be extended to the case of multiple input and output relations. It should be noted that this modeling approach is not limited to a specific environment and it could be applied to other SystemC-based frameworks. In the following, we consider application of this approach to the analysis of two specific case studies.

5. Experiments

5.1. Pipelined FFT Case Study. We aim at illustrating application of the proposed generic execution model and the computation method previously presented in Section 4 through a didactic case study. Application considered here is about a Fast Fourier Transform (FFT) algorithm which is widely used in digital signal processing. A pipeline architecture based on dedicated hardware resources is analyzed. A possible implementation of this architecture is described in [25]. An 8-point FFT is described in order to easily illustrate the proposed modeling approach. The created performance model enables to estimate resource utilization and the computation method is used to reduce the simulation time required to obtain reliable information. Figure 9 represents the pipeline architecture and the related performance model.

The lower part of Figure 9 shows the 3-stage pipeline architecture analyzed. This architecture enables to simultaneously perform transform calculations on a current frame of 8 complex symbols, load input data for the next frame of data, and unload the 8 output complex symbols of the previous frame. Each pipeline stage implements its own memory banks to store input and intermediate data. Each one is made of a processing unit that enables to perform arithmetic operations (addition, subtraction, and multiplication) on two complex numbers. Each processing unit performs an equivalent of 10 arithmetic operations on 2 real numbers at each iteration and four iterations of each processing unit are required to perform one transform calculation on 8 points. The number of clock cycles to perform each iteration is related to the way logic resources are implemented. The clock frequency of each processing unit is defined according to the expected data throughput and calculation latency. The upper part of Figure 9 gives the structural description of the proposed performance model. The behavior of each activity is specified to describe the resource utilization for each stage of the architecture. Each activity is described using the proposed generic execution model with one input and one output as presented in Figure 4. Time constraints are denoted by TC_{Stage1}, TC_{Stage2}, and TC_{Stage3}.

```
{
   switch (s)
   {
   {...}
      case s1:
         ts0 = CurrentUserTime(ns);
         To = ts0;
         CofDisplay(''to=%f ns, CcA2=%f op/s'', To, 0);
         CofDisplay(''to=%f ns, CcA2=%f op/s'', To, Ccs1);
         To = ts0 + Tj;
         CofDisplay(''to=%f ns, CcA2=%f op/s'', To, Ccs1);
         CofDisplay(''to=%f ns, CcA2=%f op/s'', To, 0);
         Select_Min = 1;
         Select_Mout = 0;
         OpDuration = Tj;
         if(k < N)s = s0;
         else    {s = s2; k++;}
         break;
      case s2:
   {...}
   }
}
```

Algorithm 1

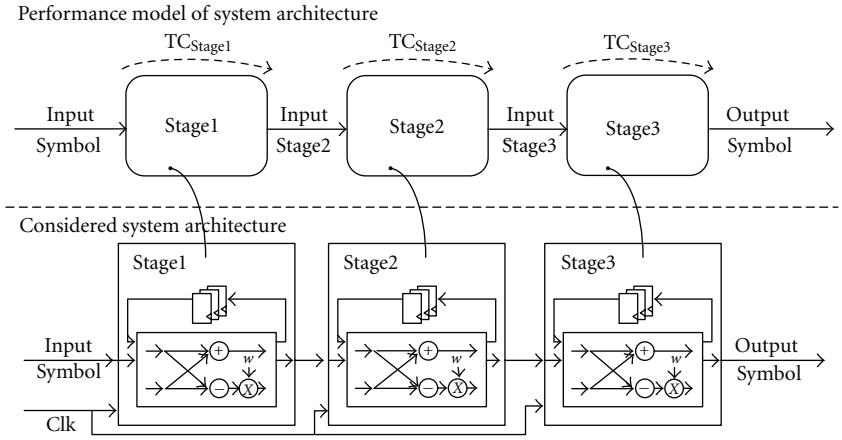


Figure 9: Performance model of a 3-stage pipeline architecture.

A first performance model has been defined by using the state-based modeling approach presented in Section 4. At the abstraction level considered, a transaction is made of 8 complex symbols. Table 2 gives the specification of the activity *Stage3*.

s_0 is the waiting state for reception of transactions through relation *InputStage3*. During processing state s_1, Cc_{s1} operations are executed. T_{Proc} represents the computation duration. T_{Idle} represents the delay between two iterations of the processing unit. For one 8-point FFT execution, four iterations of s_1 and s_2 are required to analyze the resource utilization for this pipeline stage. Considering the previously presented computation method, the evolution instants of Cc_{Stage3} are computed locally according to the arrival time of transactions. A transaction made of 8 complex symbols is finally produced through relation *OutputSymbol*. The specifications of *Stage1* and *Stage2* activities are described in the same way. Three states are needed to describe reception of transactions through the relations *InputSymbol* and *InputStage2*, evolution of assessed properties Cc_{Stage1} and Cc_{Stage2} and production of output transactions through the relations *InputStage2* and *InputStage3*. The specifications of each activity described with state-action tables are then used to parameterize the generic execution model given in Figure 4.

Figures 10 and 11 depict possible observations obtained from the simulation of the performance model described with the CoFluent Studio tool.

Figure 10 shows the resource utilization and the time evolution of the computational complexity per time unit for the third stage of pipeline to perform one 8-point FFT. In this

Table 2: Behavior of *Stage3* activity described by using the state-based modeling approach.

Current state	Next state		Actions	
	Condition	State	Condition	Action
s_0	InputStage3	s_1	—	$Cc_{Stage3} = 0$
s_1	$t = T_{Proc}$	s_2	—	$Cc_{Stage3} = Cc_{s1}$
s_2	$t = T_{Idle}$ AND $k < 4$	s_1	—	$Cc_{Stage3} = 0$
	$t = T_{Idle}$ AND $k = 4$	s_0		OutputSymbol

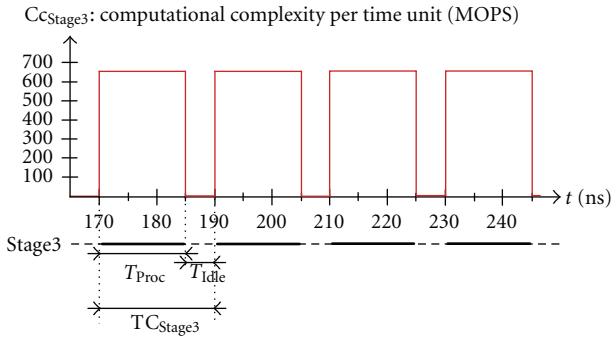


Figure 10: Time evolution of the computational complexity (in MOPS) of the third pipeline stage.

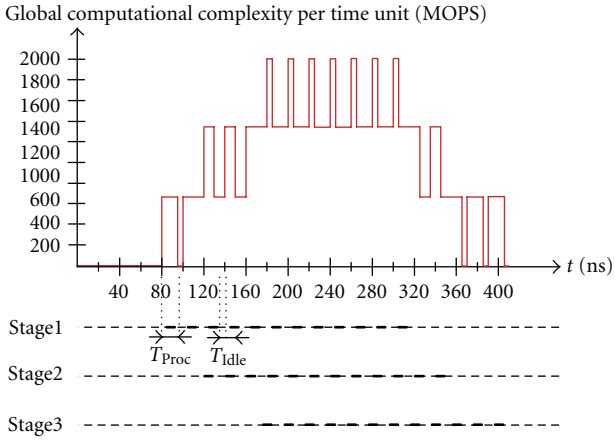


Figure 11: Time evolution of the global computational complexity (in MOPS) of the pipeline architecture with configuration considered.

example a data throughput of 100 Mega complex symbols per second is considered. Then, an 8-point FFT is to be performed every 80 ns. During this period, four iterations of each processing unit are executed. To meet the expected data throughput, time constraints TC_{Stage1}, TC_{Stage2}, and TC_{Stage3} are set to 20 ns. Observations depicted in Figure 10 are obtained with a computation duration T_{Proc} set to 15 ns. A computational complexity per time unit of 666 MOPS is also observed.

Figure 11 presents simulation results obtained to observe evolution of the global computational complexity related to the three stages of the pipeline. Three successive 8-point FFT are executed in order to observe the influence of simultaneous processing.

The lower part of Figure 11 gives information about the occupation of processing units on each pipeline stage. The upper part of Figure 11 enables to analyze the resource utilization according to the computation duration applied on each pipeline stage. For each stage T_{Proc} is fixed to 15 ns and T_{Idle} to 5 ns. A maximal computational complexity per time unit of 2 GOPS is observed with this timing configuration. This information enables the designer to deduce the number of logic resources required for the architecture. The simulation time to execute the performance model for one 8-point FFT took about 50 us on a 2.66 GHz Intel Core2 duo machine. This simulation is fast enough to compare different architecture configurations and to analyze different tradeoffs between data throughput and resource utilization.

A second performance model has been defined at a lower level of granularity following a transaction-based modeling approach as presented in Section 4. The goal is to evaluate the execution speedup obtained with the first model by reducing the number of context switches during simulation. At the data granularity considered, a transaction corresponds to one complex symbol. The second model initiates 32 transactions, whereas 4 transactions were required in the first model. Properties Cc_{Stage1}, Cc_{Stage2}, and Cc_{Stage3} evolve also each time a transaction is received. The same results previously presented with the first model can be observed. For the second performance model, the simulation speed is significantly improved by reducing the numbers of transactions required and the related context switches between threads. The first performance model enables to obtain a simulation speedup of 7.62. A minor difference with the theoretical factor speedup of 8 is due to weak effect of algorithm added to obtain the same observation about resource utilization. In [23], it is shown how the measured simulation speedup evolves linearly with the reduction factor of thread switches.

5.2. LTE Receiver System Case Study. This section highlights application of the proposed generic execution model to favor creation of transaction level models and to perform architecture exploration. The case study considered here concerns the creation of a transaction level model for analysis of processing functions involved at the physical layer of a communication receiver implementing part of the LTE protocol. This protocol is considered for next

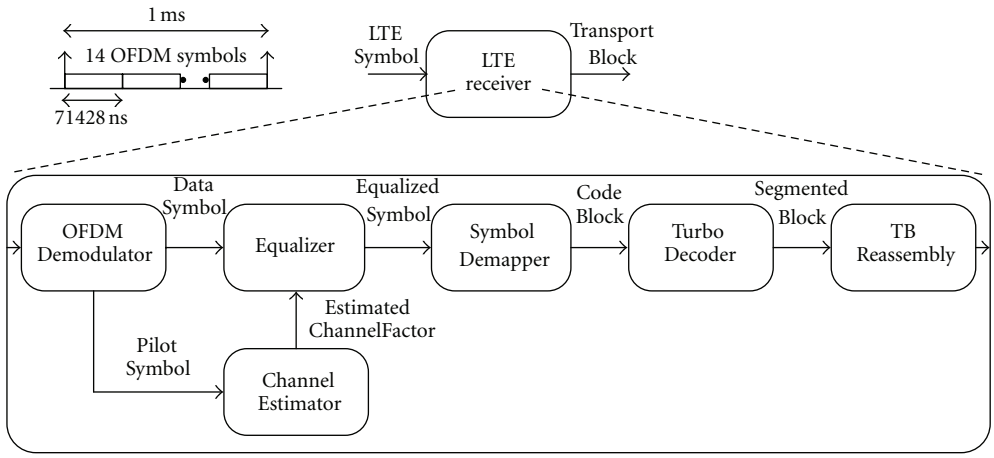


Figure 12: Activity diagram of the LTE receiver studied.

generation of mobile radio access [26]. Associated baseband architecture demands high computational complexity under real-time constraints and multiprocessor implementation is then required [27]. The aim of the proposed model is a comparison of performances achieved with two potential architectures. Information obtained by simulation enables to identify and compare the amount of logic resources and memory capacity required for potential architectures. Figure 12 describes the structural description of the studied LTE receiver, captured with the activity diagram notation adopted.

A single-input single-output (SISO) configuration is analyzed. Figure 12 depicts the activities proposed for performance analysis of baseband functions of the LTE receiver [28], namely, OFDM demodulation, channel estimation, equalization, symbol demapping, turbo decoding, and transport block (TB) reassembling. Different configurations exist for a LTE subframe. The relation *LTESymbol* represents one OFDM symbol and processing of a symbol takes 71428 ns [29]. An OFDM demodulation is performed on data associated to the relation *LTESymbol*. It is typically performed using FFT algorithm. In a LTE subframe known complex symbols, called pilots and denoted by *PilotSymbol*, are inserted at a distance of 4 and 3 OFDM symbols from one another to facilitate channel estimation. Data associated to relation *DataSymbol* are equalized to compensate effects of propagation channel. The *Symbol Demapper* activity represents interface between symbol level processing and bit processing. Channel decoding is performed through turbo decoder algorithm. The activity *TBReassembly* receives binary data block through the relation *SegmentedBlock*. Data blocks are then transmitted through relation *TransportBlock* to the Medium Access Control (MAC) layer each 1 ms when 14 OFDM symbols have been received and processed by the different functions related to the physical layer.

Figure 13 shows the performance model defined to compare the studied architectures.

The lower part of Figure 13 depicts the two studied architectures. Architecture I corresponds to an heterogeneous architecture. P_1 is considered as a processor and P_2 as a dedicated hardware resource. Architecture II consists in implementing each function as a set of dedicated hardware resources denoted by P_3. The upper part of the figure depicts the model to analyze and compare performances obtained with these two architectures. Time constraints are denoted in Figure 13 by $TC_{OFDMDemod}$, $TC_{ChanEstim}$, $TC_{Equalizer}$, and $TC_{TurboDecoder}$. TC_{P1} is the time constraint related to the sequential execution of the functions allocated on P_1. Its value is set to the reception period of input transactions, which is equal to 71428 ns. The behavior of each activity is specified in order to evaluate the memory cost and the computational complexity each function causes on the resources when executed. Table 3 illustrates the specified behavior to analyze the computational complexity related to the channel estimation function.

States s_0 and s_2 represent waiting states on relation *PilotSymbol*. Operations performed for one channel estimation iteration are performed during states s_1 and s_3. Cc_{s1} and Cc_{s3} correspond to the number of arithmetic operations executed during these processing states. According to the OFDM symbol received, 4 or 3 iterations of channel estimation are required. T_{Proc} is the computation duration of processing states. It is fixed according to the time constraint $TC_{ChanEstim}$. The behaviors related to the other activities have also been specified with state-action tables. Based on a detailed analysis of resources required for each function, we have defined analytical expressions to give relations between functional parameters related to the different configurations of LTE subframes and the resulting computational complexity in terms of arithmetic operations [30].

Each activity has been captured by using the proposed generic execution model. Activities *ChannelEstimator*, *SymbolDemapper*, *TurboDecoder*, and *TBReassembly* have been captured using the same description given in Figure 4. Activity *OFDMDemodulator* corresponds to the case of two output relations, whereas activity *Equalizer* has been described with two input relations. A design test environment has been defined to produce LTE subframes with different configurations. We captured the model in the CoFluent Studio tool. Each activity was described in a way

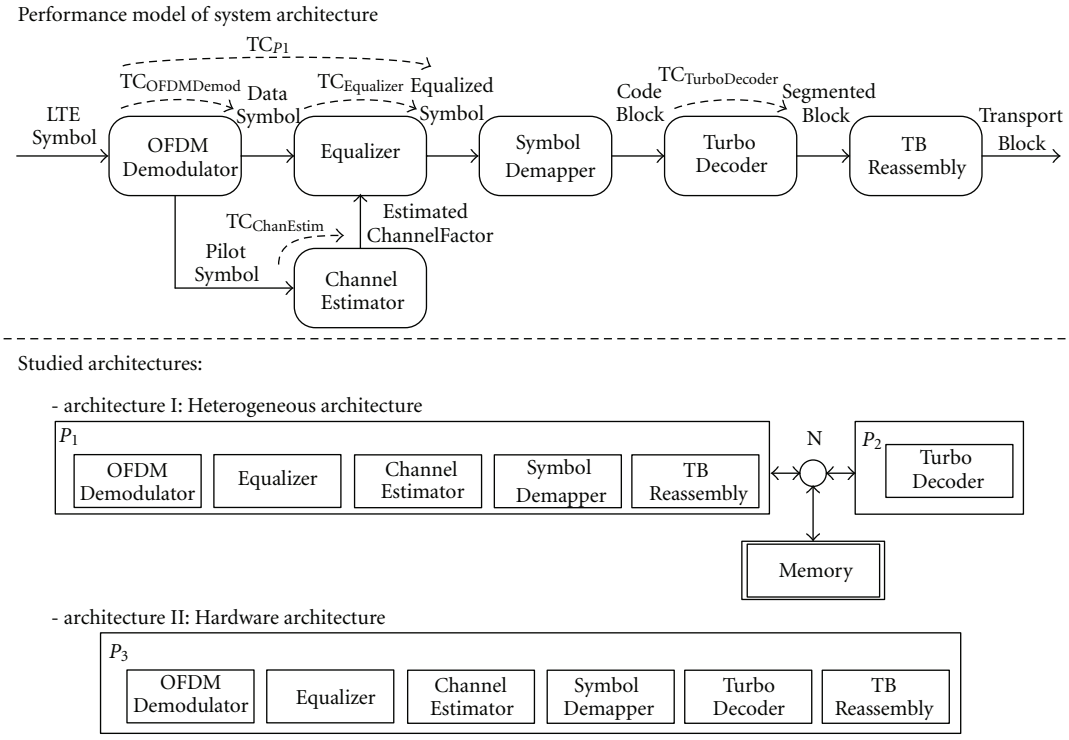


Figure 13: Proposed model to analyze performances obtained with different architectures.

Table 3: State-action table for specification of the *ChannelEstimator* activity.

Current state	Next state		Actions
	Condition	State	Action
s_0	PilotSymbol	s_1	$Cc_{Estim} = 0$
s_1	$t = T_{Proc}$ AND $k < 4$	s_1	$Cc_{Estim} = Cc_{s1}$ EstimatedChannelFactor
	$t = T_{Proc}$ AND $k = 4$	s_2	
s_2	PilotSymbol	s_3	$Cc_{Estim} = 0$
s_3	$t = T_{Proc}$ AND $k < 3$	s_3	$Cc_{Estim} = Cc_{s3}$ EstimatedChannelFactor
	$t = T_{Proc}$ AND $k = 3$	s_0	

similar to one presented in Section 4. Modeling effort was then significantly reduced due to the adoption of a generic model. The model instances are parameterized according to the specifications defined in state-action tables. The complete architecture model corresponds to 3842 lines of SystemC code, with 22% automatically generated by the tool. The rest of the code is sequential C/C++ code defined for computation and display of properties studied. Table 4 shows the time constraints considered in the following for the simulation of the two architectures studied.

These values have been set to guarantee processing of a LTE subframe each 1 ms. In case of architecture I, OFDM demodulation, channel estimation, and equalization functions are executed sequentially on the processor P_1. $TC_{OFDMDemod}$, $TC_{Equalizer}$, and $TC_{ChanEstim}$ are fixed to meet time constraint TC_{P1} and to limit the computational complexity per time unit required by processor P_1. $TC_{TurboDecoder}$ is also fixed to limit the computational complexity per time unit required by P_2.

In case of architecture II, each function can be executed simultaneously. $TC_{OFDMDemod}$ is equal to the reception period of input transaction *LTESymbol*. $TC_{Equalizer}$ and $TC_{ChanEstim}$ are defined to perform a maximum of 4 iterations of channel estimation and equalization during OFDM demodulation. $TC_{TurboDecoder}$ is set to perform turbo decoding of 1 or 2 blocks of data during equalization.

Figure 14 shows the observations obtained for architecture I with the time constraints set. Results observed correspond to the reception of a LTE subframe with the following configuration: number of resource blocks allocated: 12, size of FFT: 512, modulation scheme: QPSK, and number of iterations of turbo decoder: 5.

The Figure 14(a) depicts the evolution of the computational complexity per time unit on processor P_1. The Figure 14(b) shows the evolution of the computational complexity per time unit on P_2 to perform operations related to turbo decoding. A maximum of 1,6 GOPS is estimated for processor P_1. With the time constraints set, it is mainly

Table 4: Time constraints applied for each activity according to architectures.

Architecture	$TC_{OFDMDemod}$	$TC_{ChanEstim}$	$TC_{Equalizer}$	$TC_{TurboDecoder}$
I	$0{,}2 * TC_{P1}$	$0{,}12 * TC_{P1}$	$0{,}04 * TC_{P1}$	$0{,}08 * TC_{P1}$
II	71428 ns	14285 ns	14285 ns	7142 ns

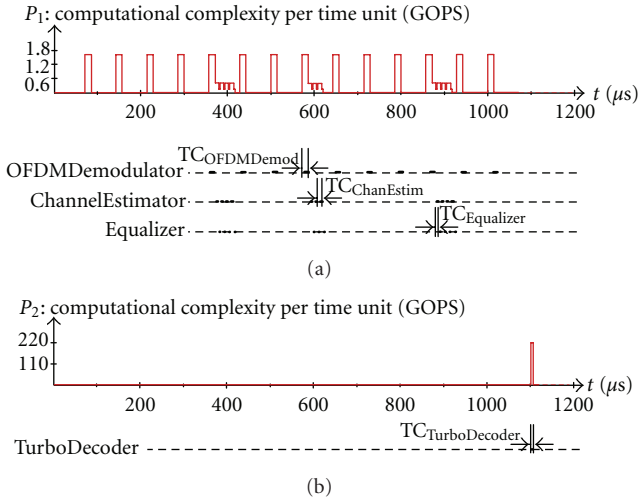


Figure 14: Evolution of the computational complexity per time unit of (a) processor P_1 and (b) P_2.

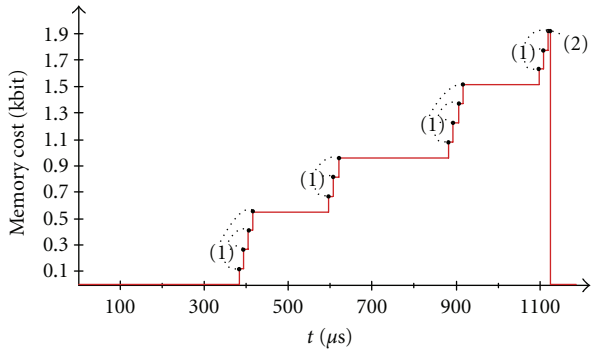


Figure 15: Evolution of the estimated memory cost (in Kbit) for architecture I.

due to OFDM demodulation. A maximum of 221 GOPS is estimated to perform turbo decoding. Figure 15 describes the evolution of the memory cost associated to the symbol demapper and transport block reassembly functions.

The memory cost evolves each time a transaction is received or sent by one of the three functions studied. Instants (1) correspond to the evolution of the memory cost after reception of a packet of data by the symbol demapper function. The amount of data stored in memory increases each time a packet of data is received by the symbol demapper. At instant (2), the complete LTE subframe has been processed at the physical layer level and the resulting data packet is produced. The maximum value achieved with the considered LTE subframe is estimated to be 1920 bits.

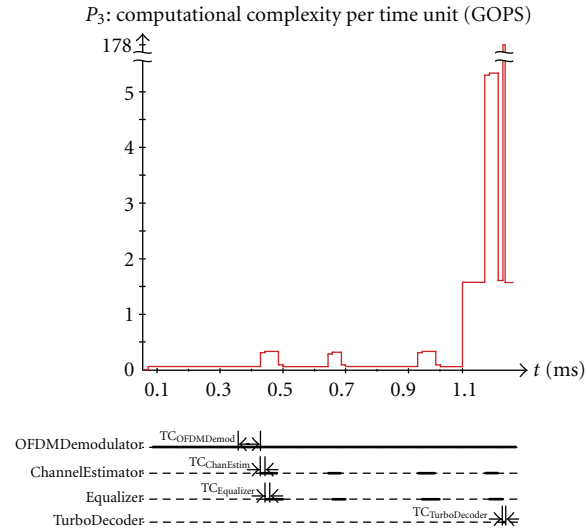


Figure 16: Evolution of the computational complexity per time unit for architecture II.

Figure 16 illustrates the observation obtained with time constraints considered for architecture II and for the LTE subframe configuration evaluated previously.

The upper part of Figure 16 shows evolution of the computational complexity per time unit for P_3. The lower part depicts the resource utilization of P_3. A maximum computational complexity per time unit of 177,812 GOPS is observed when turbo decoding and OFDM demodulation are performed simultaneously.

Table 5 summarizes the simulation results obtained for architectures I and II. The maximal computational complexity metric is considered here because it directly impacts area and energy consumption of the two candidate architectures. With architecture II, we note that simultaneous execution of functions on the processing resources enables to reduce the maximum computational complexity from 223,382 GOPS to 177,812 GOPS. For architecture I, the resource usage metric enables to express the percentage of time used by P_1 to execute each function. For architecture II, the resource usage metric expresses the usage of each dedicated resource for processing one OFDM symbol.

Observations given in Figures 14, 15, and 16 are used to estimate expected resources of the architecture. Similar observations can be obtained for different subframe configurations. The simulation time to execute the performance model for 1000 input frames took 11 s on a 2.66 GHz Intel Core2 duo machine. This is fast enough for performing performance evaluation and for simulating multiple configurations of architectures. Time properties and quantitative

Table 5: Maximal computational complexity and utilization of the processing resources observed for the two architectures considered.

	Architecture I				Architecture II	
	P_1		P_2		P_3	
	Maximal computational complexity (GOPS)	Resource usage (%)	Maximal computational complexity (GOPS)	Resource usage (%)	Maximal computational complexity (GOPS)	Resource usage (%)
OFDMDemodulator	1,612	20	—	—	0,322	100
ChannelEstimator	0,406	48	—	—	0,244	80
Equalizer	0,151	12	—	—	0,03	80
TurboDecoder	—	—	221,77	8	177,49	10
Application	1,612	80	221,77	8	177,812	—

properties defined for each activity can be easily modified to evaluate and compare various architectures.

The main benefit of the presented approach comes from the adoption of a generic modeling style. The modeling effort is then significantly reduced. We estimate that the creation of the complete model of the LTE Receiver architecture took less than 4 hours. Once the model created, parameters can easily be modified to address different architectures and simulation time is fast enough to allow exploration. In the presented modeling approach the simulation method used makes possible to run ahead evolution of the studied properties in a local time until activities need to synchronize. This favours creation of models at higher abstraction level. Synchronization points are defined by transactions exhibited in the architecture model. Then, this approach is sensitive to estimates related to each function and further works should be led in order to evaluate the simulation speed-accuracy tradeoff.

6. Conclusion

Abstract models of system architectures represent a convenient solution to maintain the design complexity of embedded systems and to enable architecting of complex hardware and software resources. In this paper, we have presented a state-based modeling approach for the creation of transaction level models for performance evaluation. According to this approach, system architecture is modeled as an activity diagram and description of activities incorporates properties relevant to resources used. The presented contribution is about a generic execution model defined in order to facilitate the creation of performance models. This model relies on a specific computation method to significantly reduce the simulation time of performance models. The experimentation of this modeling style has been illustrated through the use of the framework CoFluent Studio. However, the approach is not limited to this specific environment and it could be applied to other SystemC-based frameworks. Further research is directed towards validation of estimates provided by simulation and applying the same modeling principle to other nonfunctional properties such as dynamic power consumption.

References

[1] C. H. van Berkel, "Multi-core for mobile phones," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 1260–1265, April 2009.

[2] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli, "A platform-based taxonomy for ESL design," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 359–373, 2006.

[3] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration, the VLSI Journal*, vol. 38, no. 2, pp. 131–183, 2004.

[4] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[5] C. Haubelt, J. Falk, J. Keinert et al., "A systemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 22 pages, 2007.

[6] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer, 2005.

[7] L. Gai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 19–24, Newport Beach, Calif, USA, October 2003.

[8] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*, Kluwer Academic, Dodrecht, The Netherlands, 2000.

[9] Open SystemC Initiative (OSCI), "Functional specification for SystemC 2.0," http://www.systemc.org.

[10] Open SystemC Initiative TLM Working Group, Transaction Level Modeling Standard 2 (TLM 2), June 2008.

[11] R. Dömer, A. Gerstlauer, J. Peng et al., "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 647953, 2008.

[12] Soclib: a modelisation and simulation platform for system on chip design, 2003, http://www.soclib.fr.

[13] G. Schirner and R. Dömer, "Quantitative analysis of the speed/accuracy trade-off in transaction level modeling," *Transactions on Embedded Computing Systems*, vol. 8, no. 1, article 4, 2008.

[14] P. Lieverse, P. van Der Wolf, K. Vissers, and E. F. Deprettere, "A methodology for architecture exploration of heterogeneous

signal processing systems," *The Journal of VLSI Signal Processing*, vol. 29, no. 3, pp. 197–207, 2001.

[15] T. Wild, A. Herkersdorf, and G. Y. Lee, "TAPES-Trace-based architecture performance evaluation with SystemC," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005.

[16] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "Performance evaluation of UML2-modeled embedded streaming applications with system-level simulation," *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 826296, 16 pages, 2009.

[17] J. Kreku, M. Hoppari, T. Kestilä et al., "Combining UML2 application and systemC platform modelling for performance evaluation of real-time embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2008, Article ID 712329, 18 pages, 2008.

[18] Object Management Group (OMG), "A UML profile for MARTE, beta 1 specification," August 2007.

[19] A. Viehl, B. Sander, O. Bringmann, and W. Rosenstiel, "Integrated requirement evaluation of non-functional system-on-chip properties," in *Proceedings of the Forum on Specification, Verification and Design Languages (FDL'08)*, pp. 105–110, Stuttgart, Germany, September 2008.

[20] J. P. Calvez, *Embedded Real-Time Systems: A Specification and Design Methodology*, John Wiley & Sons, New York, NY, USA, 1993.

[21] D. Harel and M. Politi, *Modeling Reactive Systems with Statechart*, McGraw-Hill, 1993.

[22] D. D. Gajski, *Principles of Digital Design*, Prentice-Hall, 1996.

[23] S. Le Nours, A. Barreteau, and O. Pasquier, "Modeling technique for simulation time speed-up of performance computation in transaction level models," in *Proceedings of the Forum of specification and Design Languages (FDL '10)*, vol. 2010, pp. 136–141, Southampton, UK, September 2010.

[24] CoFluent Design, http://www.cofluentdesign.com/.

[25] Xilinx. LogiCORE Fast Fourier Transform v7.1., http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf.

[26] E. Dahlman, S. Parkvall, J. Skold, and P. Beming, *3G Evolution, HSPA and LTE for Mobile Broadband*, Academic Press, 2008.

[27] C. Jalier, D. Lattard, A. A. Jerraya, G. Sassatelli, P. Benoit, and L. Torres, "Heterogeneous vs homogeneous MPSoC approaches for a mobile LTE modem," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '10)*, pp. 184–189, Dresden, Germany, March 2010.

[28] 3GPP. TS 36.201, Evolved Universal Terrestrial Radio Access (E-UTRA); LTE physical layer; General description, March 2010, http://www.3gpp.org/ftp/Specs/html-info/36201.htm.

[29] 3GPP. TS 36.211 V9.1.0, Evolved Universal Terrestrial Radio Access (E-UTRA); Physical channels and modulation, April 2010, http://www.3gpp.org/ftp/specs/html-info/36211.htm.

[30] J. Berkmann, C. Carbonelli, F. Dietrich, C. Drewes, and W. Xu, "On 3 G LTE terminal implementation—standard, algorithms, complexities and challenges," in *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC '08)*, pp. 970–975, August 2008.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 906350, 14 pages
doi:10.1155/2012/906350

Research Article

High-Level Synthesis under Fixed-Point Accuracy Constraint

Daniel Menard, Nicolas Herve, Olivier Sentieys, and Hai-Nam Nguyen

IRISA/ENSSAT, University of Rennes, 6 rue de Kerampont, 22305 Lannion, France

Correspondence should be addressed to Daniel Menard, daniel.menard@irisa.fr

Received 18 July 2011; Revised 12 December 2011; Accepted 4 January 2012

Academic Editor: Philippe Coussy

Implementing signal processing applications in embedded systems generally requires the use of fixed-point arithmetic. The main problem slowing down the hardware implementation flow is the lack of high-level development tools to target these architectures from algorithmic specification language using floating-point data types. In this paper, a new method to automatically implement a floating-point algorithm into an FPGA or an ASIC using fixed-point arithmetic is proposed. An iterative process on high-level synthesis and data word-length optimization is used to improve both of these dependent processes. Indeed, high-level synthesis requires operator word-length knowledge to correctly execute its allocation, scheduling, and resource binding steps. Moreover, the word-length optimization requires resource binding and scheduling information to correctly group operations. To dramatically reduce the optimization time compared to fixed-point simulation-based methods, the accuracy evaluation is done through an analytical method. Different experiments on signal processing algorithms are presented to show the efficiency of the proposed method. Compared to classical methods, the average architecture area reduction is between 10% and 28%.

1. Introduction

Implementing signal processing applications in embedded systems generally requires the use of fixed-point arithmetic [1, 2]. In the case of fixed-point architectures, operators, buses, and memories need less area and consume less power compared to their equivalent using floating-point arithmetic. Furthermore, floating-point operators are more complex and lead to longer execution time.

However, the main problem slowing down the hardware implementation flow is the lack of high-level development tools to target these architectures from algorithmic specification language using floating-point data types. In this design process, mainly two kinds of high-level Computer Aided Design (CAD) tools are required for reducing the time-to-market: floating-point to fixed-point conversion and High-Level Synthesis (HLS).

For hardware implementation like FPGA or ASIC, the floating-point to fixed-point conversion is a complex and an error prone task that converts an application specified with high-precision floating-point data into an algorithm using fixed-point arithmetic, usually under an accuracy constraint. Then, HLS automatically translates the algorithm specified with fixed-point data into an optimized dedicated architecture. In the processing part of this architecture, the number and the type of operators must be defined Moreover, each operator input and output word-length must be determined. For complex designs, the word-length search space is too large for a manual exploration. Thus, time-to-market reduction requires high-level tools to automate the fixed-point architecture synthesis process.

The aim of HLS, handling multiple word-length, is to minimize the implementation cost for a given fixed-point accuracy constraint. This process leads to an architecture where each operator word-length has been optimized. Best results are obtained when the word-length optimization (WLO) process is coupled with the HLS process [3, 4]. HLS requires operator word-length to correctly execute its allocation, scheduling, and resource binding steps. But the word-length optimization process requires operation-to-operator binding. To deal with this optimization issue, an iterative refinement process should be used. Many published methodologies [5–9] do not couple data WLO and HLS processes. Moreover, simulation-based accuracy evaluation is used, which leads to prohibitive optimization time.

In this paper, a new method for HLS under accuracy constraint is proposed. The WLO process and the HLS

are combined through an iterative method. Moreover, an efficient WLO technique based on tabu search algorithm is proposed to obtain solutions having better quality. Compared to existing methods, the HLS synthesis process is not modified and thus this method can take advantage of existing academic and commercial tools. Furthermore, the proposed method benefits from an analytical accuracy evaluation tool [10], which allows obtaining reasonable optimization times. Experiments show that good solution can be obtained with a few iterations.

This paper is organized as follows. In Section 2, related work in the area of multiple word-length architecture design is summarized. Then, the proposed fixed-point conversion method for hardware implementation is presented in Section 3. The multiple word-length architecture optimization is detailed in Section 4. In Section 5, different experiments on various signal processing algorithms are presented to show the efficiency of the proposed method. Finally, Section 6 draws conclusions.

2. Related Works

The classical method used to optimize data word-length relies on handling uniform word-lengths (UWL) for all data that reduces the search space to one dimension and simplifies the synthesis because all operations will be executed on operators with the same word-length. However, considering a specific fixed-point format for each data leads to an implementation with a reduced power, a smaller area, and a smaller execution time [9].

In the sequential method [5–9], the word-lengths are first optimized and then the architecture is synthesized. The first step gives a fixed-point specification that respects the accuracy constraint. For this purpose, a dedicated resource is used for each operation. So, the HLS is not considered first, because there is no resource sharing. The second step of the process corresponds to HLS. In [9], a heuristic to combine scheduling and resource sharing is proposed for a data flow graph with different word-lengths. This method implements a fixed-point application, which leads to a numerical accuracy greater than the constraint. In the first step, the data WLO gives a numerical accuracy close to the accuracy constraint, but, in the second step, the binding to larger operators will improve the global numerical accuracy. Consequently, the obtained solution may not be optimized exactly for the specified accuracy constraint given that the two steps are not coupled.

A method combining word-length optimization and HLS has been proposed in [11]. This method is based on a Mixed Integer Linear Programming (MILP). This MILP formulation leads to an optimal solution. Nevertheless, some simplifications have been introduced to limit the number of variables. This method is restricted to linear time-invariant systems, and the operator latency is restricted to one cycle. Moreover, the execution time to solve the MILP problems can become extremely long and several hours could be needed for a classical IIR filter.

In [3], the authors propose a method where the HLS is achieved during the WLO phase. The authors take account of the resource sharing to reduce the hardware cost but also to reduce the optimization time. Indeed, the accuracy evaluation is obtained through fixed-point simulations. Therefore, heuristics are used to limit the search space and to obtain reasonable optimization time. A first step analyzes the application SFG and groups some data according to rules. For example, addition inputs will be specified with the same fixed-point format. The second step determines the required minimum word-length (MWL) for each data group. The MWL of a group corresponds to the smallest word-length for the data of the group allowing fulfilling the accuracy constraint when the quantization effect of the other groups is not considered. This MWL is used as a starting point because its computation can be achieved in a reasonable execution time when simulation-based methods are used to evaluate the accuracy. In the third step, the fixed-point specification is scheduled and groups are bound to operators using the word-length found in the previous step. During the combined scheduling-binding, some operations are bound to larger operators. Finally, the last step corresponds to the operator WLO.

The synthesis and WLO processes have to be interactive and have to be finally terminated with a synthesis to exactly implement the fixed-point specification optimized for the given accuracy constraint. Indeed, the last step of the method proposed in [3] optimizes the operator word-length. But this process can challenge the scheduling obtained in the previous step.

In [4], a method combining WLO and HLS through an optimization process based on simulated annealing is proposed. In the following, a movement refers to a modification in the system state of the simulated annealing optimization heuristic. This method starts with the solution obtained with uniform word-length (UWL). In this optimization process based on simulated annealing, movements on the HLS are carried-out by changing the mapping of the operations to the operators. An operation can be mapped to a nonmapped or to another already mapped resource or operations can be swapped. Movements on WLO are carried out by modifying the operation word-length. The movement can increase or decrease the word-length of a signal of one bit or make more uniform the word-length of the operation mapped to a same operator. A movement is accepted if the implementation cost is improved compared to the previous solutions and the accuracy constraint fulfilled. If the accuracy constraint is fulfilled but the implementation cost is not improved, the movement is accepted with a certain probability decreasing with time. Thus, for each movement, the implementation cost, the fixed-point accuracy, and the total latency of the current solution must be computed. Stochastic algorithms lead to good quality solutions for optimization problems with local minimum. But they are known to require a great number of iterations to obtain the optimized solution. Given that each iteration requires an architecture synthesis and an accuracy evaluation, the global optimization time can be very high.

In this paper, a new HLS method under accuracy constraint is proposed. An iterative process is used to link HLS and WLO processes, and good results are obtained

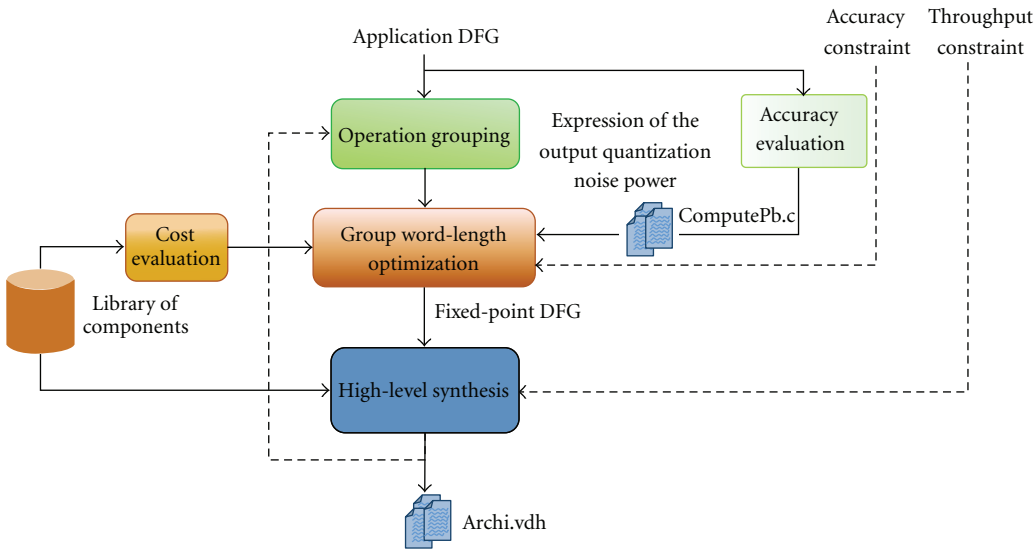


Figure 1: Combined HLS and WLO environment.

with a few iterations. The accuracy evaluation is carried-out through an analytical method leading to reasonable optimization time. Compared to [3, 4], a classical HLS tool can be used and no modification of this tool is required. Thanks to the analytical method, the optimized word-length (OWL) associated to each operation can be computed in a reasonable time and is used as starting point, as opposed to the MWL like in [3]. It is obvious that the MWL is less relevant than the OWL since that the quantization effects of the other operations are not taken into account.

3. High-Level Synthesis under Accuracy Constraint

A fixed-point data is composed of two parts corresponding to the integer part and the fractional part. The number of bits for each part is fixed. iwl is the number of bits for the integer part integrating the sign bit, fwl is the number of bits for the fractional part, while $w = iwl + fwl$ is the total number of bits. The scaling factor associated with the data does not evolve according to the data value as in floating-point arithmetic. So, the aim of fixed-point conversion process is to determine the optimized number of bits for the integer part and the fractional part.

The proposed HLS method under accuracy constraint is detailed in Figure 1. The aim is to implement an algorithm specified with floating-point data types into an architecture using fixed-point arithmetic. This method is based on the definition of a co-synthesis and WLO environment. The input of the framework, for multiple word-length high-level synthesis, is the Data Flow Graph (DFG) of the application. Nevertheless, Control Data Flow Graph can be used if the HLS tool supports this intermediate representation. For each operation operand, the binary point-position has been determined. The binary point-position must allow the representation of the data extreme values without overflow and the minimization of the number of bits used for the integer part.

In our multiple word-length high-level synthesis approach, the operator WLO and the HLS are coupled. The goal is to minimize the architecture cost as long as the accuracy constraint is verified. The multiple word-length HLS is an iterative process as explained in Section 3.1. The HLS, under throughput constraint, is carried out with the tool GAUT [12]. The HLS and the WLO process usees a library composed of various types of operator characterized in terms of performances for different operand word-lengths.

3.1. Multiple Word-Length Architecture Optimization. To obtain an optimized multiple word-length architecture, the operator word-length determination and the HLS must be coupled. Indeed, for the HLS, the operation word-length must be known. The operator propagation time depends on the input and output word-length. For the operator WLO, the resource sharing must be taken into account. A group g_i is defined as a set $\mathcal{S}_{g_i}$ of operations o_k that will be computed on the same operator α_i. To determine a group, the operation assignment must be known. The operations executed by the same operator α_i must have the same word-length w_i, and this condition must be taken into account during the optimization of the group word-length.

To couple HLS and WLO, the proposed method is based on an iterative process with the aim of finding the optimized operation binding, which minimizes the cost through word-length minimization. The method efficiently combines resource sharing obtained through HLS and WLO search. This process is presented in Figure 2. This process is in four steps.

The first step defines the number of groups needed for each type of arithmetic operation. For the first iteration, the number of group for each operation type is set to one. Indeed, the number of operators required to execute

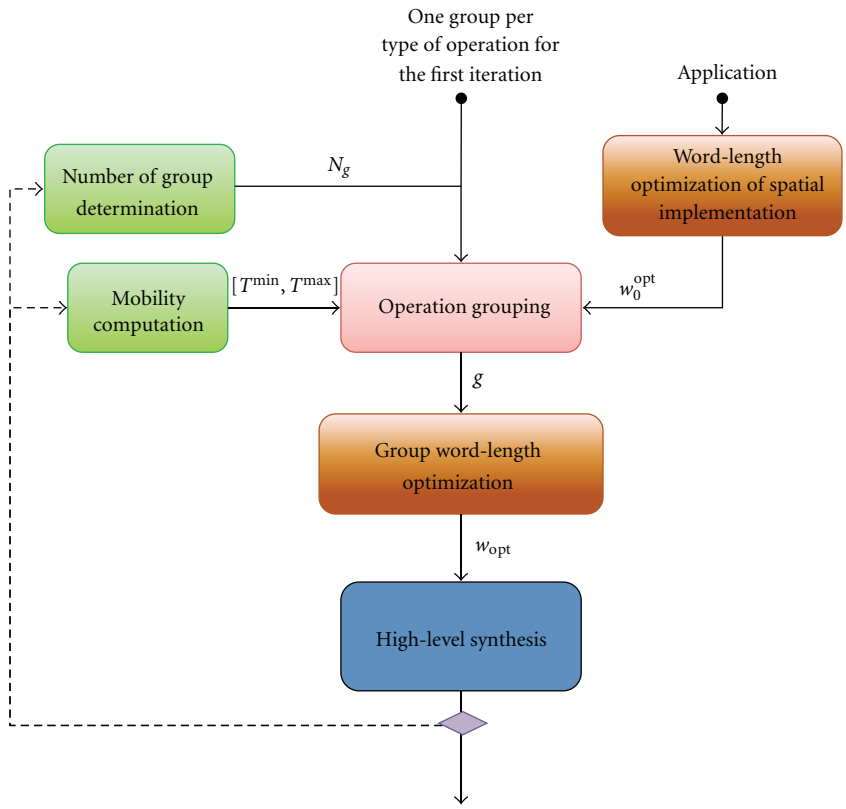


Figure 2: Iterative process for combining word-length optimization and high level synthesis.

each operation type is unknown and can be defined only after an architecture synthesis. For the other iterations, the number of groups for each operation type is defined from the HLS results obtained in the previous iteration. The group number for an operation type is fixed to the number of operators used for this operation type. In the second step, a grouping algorithm is applied. This step, relatively similar to clustering [6, 9], aims to find the best operation combinations, which would lead to interesting results for the WLO process and HLS. The technique is presented in Section 3.1.1. The third step searches the optimal word-length combination for this grouping, that minimizes the implementation cost and fulfills the accuracy constraint. This optimization process is detailed in Section 4. The fourth step is the architecture processing part synthesis from the fixed-point specification obtained in the third step. After this synthesis, the number of operators used for each operation type has to be reconsidered. Indeed, operation word-lengths have been reduced leading to operator latency decrease. This can offer the opportunity to reduce the number of operators during the scheduling. Thus, an iterative process is necessary to converge to an optimized solution, and the algorithm stops when successive iterations lead to the same results or when the maximal number of iterations is reached.

3.1.1. Operation Grouping. Operation grouping is achieved from an analysis of the synthesis result. A group g_i is defined for each operator α_i of the synthesized architecture. Each group g_i is associated with a word-length w_{g_i} that corresponds to the maximal word-length of the operations o_k associated to the group g_i

$$w_{g_i} = \max_{o_k \in \delta_{g_i}} \left(w_{o_k}^{\text{opt}} \right), \tag{1}$$

where $w_{o_k}^{\text{opt}}$ is the optimized word-length associated with each operation o_k. $w_{o_k}^{\text{opt}}$ corresponds to optimized word-length obtained with a spatial implementation, that is, where each operation has a dedicated fixed-point operator. This word-length $w_{o_k}^{\text{opt}}$ is obtained with the optimization algorithm presented in Section 4.3, when a group g_k is assigned at each operation o_k.

For each operation a mobility interval $[T_i^{\min}; T_i^{\max}]$ is computed. The mobility index m_{α_i} is defined as the difference between the execution dates, $T_i^{\min}$ and $T_i^{\max}$, obtained for two list schedulings in the direct and reverse directions. Operations are treated with a priority to least mobility operations. The mobility index m_{α_i} is used to select the most appropriate group for operation o_i.

To group the operations, the optimized word-length $w_{o_k}^{\text{opt}}$ associated with each operation o_k is considered. An operation o_k is preferably associated to the group g_i with the word-length w_{g_i} immediately greater than to the optimized operation word-length $w_{o_k}^{\text{opt}}$ and compatible in terms of mobility interval. In case of mobility inconsistency, the grouping algorithm tries, firstly, to make the operation o_k take the place of one or more operations o_j having a smaller word-length $w_{o_j}^{\text{opt}}$, secondly, to place o_k in another group

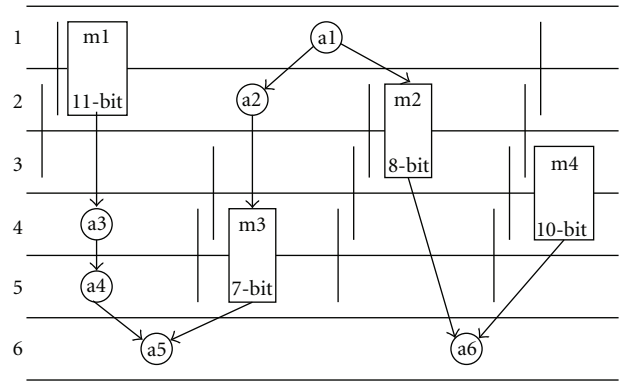


Figure 3: Application DFG showing alternative scheduling possibilities meeting timing constraints for the multiplications (no resource constraint).

Table 1: Priority list of multiplication operations with mobility index and spatio-optimal word-length.

Operation	Mobility index	$w_{o_k}^{\text{opt}}$	Priority index
m1	3	11-bit	1 (highest)
m2	3	8-bit	2
m3	4	7-bit	3
m4	5	10-bit	4

having a greater word-length and finally creates a new group with this operation o_i if other alternatives have failed. The idea of this method is to obtain for each operation the smaller word-length, and to favor placement in smaller word-length groups. When an operation o_j has been removed from a group, this operation returns to the top of the priority list of operations to be assigned. The convergence of the algorithm is ensured by the fact that operations are placed one by one in groups according to their priority and can be returned in the priority list only by operations having strictly a higher word-length.

3.1.2. Illustrative Example. The following example illustrate, the concepts for operation grouping described above. The DFG presented in Figure 3 is considered. The circles represent addition operations (named a1,…,a7), rectangles represent multiplication operations (named m1,…,m4), arrows represent data dependency, number in italics are the optimized word-length of corresponding operation, and vertical bars represent scheduling alternatives for given multiplication. The time constraint is set to 6 clock cycles. Table 1 gives the optimized word-length $w_{o_k}^{\text{opt}}$, the initial mobility index, and the associated priority for multiplication operations.

In the rest of the example, scheduling and cost of additions are not considered to simplify illustration. The latency of the multiplications is equal to two clock cycles. For the second iteration of the iterative process, the algorithm proceeds as follows.

Step 1. The ready operation m1 with highest priority is scheduled and assigned to a resource. As there is no resource selected yet, this operation defines the first resource for type m. This resource is named **M1** (with a word-length of 11 bits).

Step 2. The ready operation m2 (with highest priority now) is scheduled and assigned to a resource. As there is scheduled time on resource **M1** compatible with m2, m2 is assigned on **M1**.

Step 3. The ready operation m3 (with highest priority now) is scheduled and assigned to a resource. As there is no compatible time on **M1** regarding the mobility of m3, a second resource **M2** is created with a word-length of 7 bits.

Step 4. Operation m4 was always ready but with a lowest priority due to its highest mobility compared to the other operations. Standard list scheduling algorithm would have allocate this operation on resource **M2** since there is no more place on resource **M1**, increasing word-length of group **M2** to 10 bits. The proposed algorithm allows operation m4 to deallocate operation m2 that have smaller optimized word-length and m4 is scheduled on resource **M1**.

Step 5. This step try to realocate operation m2 on resource of immediately superior word-length, corresponding to resource **M1**. As there is no place and no operation with smaller optimized word-length, operation m2 is placed on resource **M2** and **M2** word-length is updated to 8 bits. Observe that mobility of present operation m3 is used to maximize use of resources and let operations m2 and m3 fit on resource **M2**.

After Step 5, there is no more operation to schedule and allocate, so the algorithm finished. Resource **M1** will execute operations m1 and m4 with an effective word-length of 11 bits, and resource **M2** will execute operations m2 and m3 with an effective word-length of 8 bits resulting in a smaller architecture, while a more naive algorithm would have required an 11-bit and 10-bit multiplier.

Figure 4 presents the various steps of assignments. The step number of operation assignment is indicated by circled numbers. Figure 4(a) presents Steps 1 to 3 and Figure 4(b) Steps 4 to 5 after reassignment of operation m2.

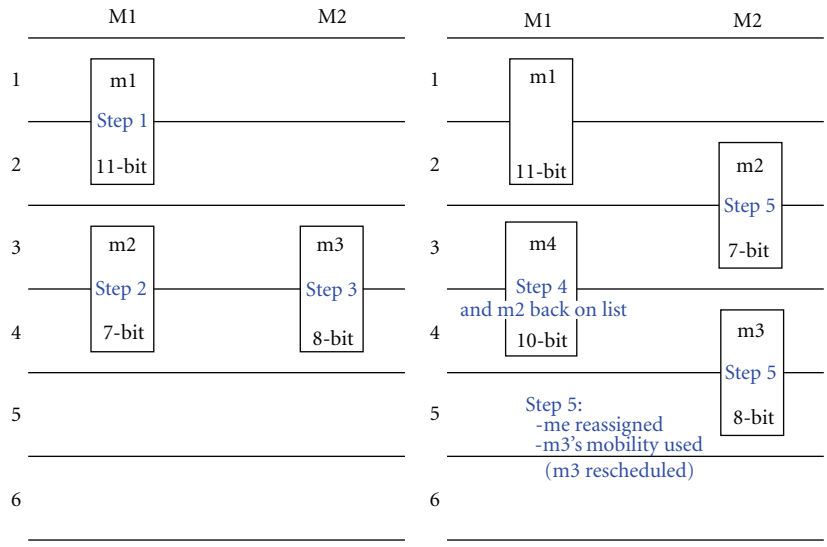


Figure 4: Scheduling with resource assignments with step information.

3.2. High-Level Synthesis Process. The high-level synthesis tool Gaut [12] is used to generate an optimized architecture from the DFG of the application. The aim is to minimize the architecture area for a given throughput constraint. The high-level synthesis process is composed of different steps. The selection module selects the best operator for each operation from the library. This library is the same as the one used for word-length optimization. Each component is characterized in terms of area, latency, cadence, and energy consumption. A list scheduling is used to schedule the operations. The algorithm is based on a mobility heuristic depending on the availability of allocated operators. Operation assignment to operators is carried out simultaneous to the scheduling task. Finally, the architecture is globally optimized to obtain a good trade-off between the storage elements (register, memory) and the interconnection elements (multiplexer, demultiplexer, tristates, and buses). Different algorithms can be used. The best results for complex applications are obtained with a variant of the left-edge algorithm.

4. Word-Length Optimization

The aim of the WLO is to find the best group word-lengths, which minimize the architecture cost $\mathcal{C}$ as long as the accuracy constraint $\lambda_{\min}$ is fulfilled. This optimization problem is described with the following expression:

$$\min(\mathcal{C}(\mathbf{w})) \text{ such as } \lambda(\mathbf{w}) \geq \lambda_{\min} \tag{2}$$

with $\mathbf{w} = [w_{g_0}, w_{g_1}, \ldots, w_{g_{N_g-1}}]$, the vector containing the word-length associated to each group. $\lambda(\mathbf{w})$ corresponds to the numerical accuracy obtained for a given group word-length $\mathbf{w}$. The evaluation of the numerical accuracy is summarized in Section 4.2.1. The cost function $\mathcal{C}$ is evaluated with the method presented in Section 4.1. For each tested combination, the accuracy and the cost function are evaluated with mathematical expressions, so that, the optimization time will be significantly reduced compared to a simulation-based method.

4.1. Cost Function. The aim of the HLS is to obtain an optimized architecture from the application functional specification. The architecture processing part is built by assembling different logic entities corresponding to arithmetic operators, multiplexers, and registers. These elements come from a library associated to the targeted FPGA or ASIC technology.

4.1.1. Generation of Operator Library. In the case of multiple word-length synthesis, the arithmetic operator library contains operators with different input and output word-lengths. The library generation flow is described in Figure 5. First, the different library elements are synthesized to obtain placed-and-routed blocks. Then, these elements are characterized by the information collected after operation synthesis.

A parameterized VHDL description is written for each operator type. From this description, a logic synthesis is achieved separately for each library element. The script-based method is used to automatically generate the library elements for different word-lengths. This logic synthesis is achieved with the Synplify Pro tool (Synopsys) for FPGA and with Design Compiler (Synopsys) for ASIC.

Let E_{Lib} denote the set containing all the library elements. Each operator α_j is characterized by the number of resources used n_{α_j} (logic cells and dedicated multipliers, for FPGA, and standard cells and flip-flops for ASIC), the propagation time t_{α_j}, and the energy consumption e_{α_j} for different input and output word-lengths. For the HLS,

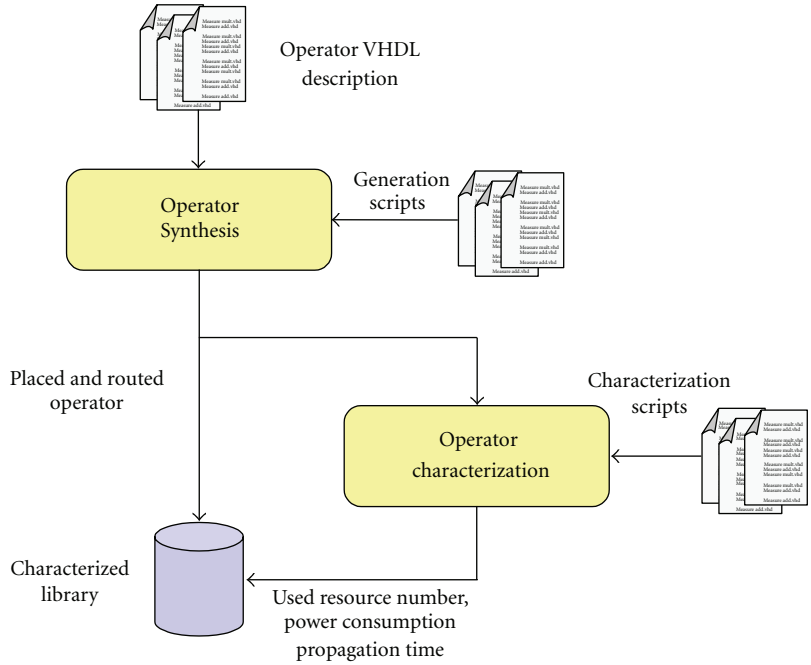


Figure 5: Operator synthesis and characterization flow.

the latency l_{α_j} of the operator is expressed in number of cycles

$$l_{\alpha_j} = \left\lceil \frac{t_{\alpha_j}}{t_{\mathrm{CLK}}} \right\rceil, \tag{3}$$

where t_{CLK} is the clock period used for the system.

This different information is used in the HLS and WLO processes. The mean power consumption of these components is characterized at the gate level with several random input vectors. The number of vectors is chosen to ensure the convergence to the mean value. This characterization is finally saved as an XML database exploited in the proposed method.

4.1.2. Model of Cost Function. The aim of the WLO is to minimize the architecture cost $\mathcal{C}$. Let S_{Opr} denote the subset of operators used for the architecture from the library ($S_{\mathrm{Opr}} \subset S_{\mathrm{Lib}}$). Let c_i denote the cost associated with each operator α_i. This cost depends on $\mathbf{w}(i)$, the word-length of operator α_i. The global cost for the architecture processing part is defined as the sum of the different costs of the operator o_i used for the architecture

$$\mathcal{C}(\mathbf{w}) = \sum_{i \in S_{\mathrm{Opr}}} c_i(\mathbf{w}(i)). \tag{4}$$

The cost used in the proposed method corresponds to the architecture area evaluated through the number of LUTs of a functional unit. For FPGA integrating dedicated resources, the user has to define a maximum number for each dedicated resource type. Moreover, other cost functions can be used to optimize energy consumption.

4.2. Constraint Function. The constraint function of the optimization problem corresponds to the fixed-point numerical accuracy. The use of fixed-point arithmetic leads to unavoidable error between the results in finite precision and in infinite precision. The fixed-point implementation is correct only if the application quality criteria are still fulfilled. Given that the link between the fixed-point operator word-lengths and the application quality criteria is not direct, an intermediate metric λ is used to define the fixed-point accuracy. The most commonly used metric is the Signal to Quantization Noise Ratio (SQNR) [3]. This metric corresponds to the ratio between the signal power and the quantization noise power.

The accuracy constraint λ_{min}, corresponding to the minimal value of the SQNR, is determined from the system performance constraints. This accuracy constraint is defined such as the system quality criteria will be still verified after the fixed-point conversion process.

4.2.1. Fixed-Point Accuracy Evaluation. Two kinds of method can be used to determine the fixed-point accuracy. These methods are either based on fixed-point simulations or analytical. Simulation-based methods estimate the quantization noise power statistically from signal samples obtained after fixed-point and floating-point simulations [3]. The floating-point result is considered as the reference because the associated error is negligible compared to the fixed-point one. The fixed-point simulation requires to emulate all the fixed-point arithmetic mechanisms. Moreover, to obtain an accurate evaluation, an important number of samples is necessary. The combination of these two phenomena leads to an important simulation time. In the WLO process, the fixed-point accuracy is evaluated

at each iteration. For complex systems, where the number of iterations is important, the fixed-point simulation time becomes prohibitive, and the search space cannot be explored.

An alternative to simulation-based methods is the analytical approach, which determines a mathematical expression of the noise power at the system output according to the statistical parameters of the different noise sources induced by quantization. In this case, the execution time required to evaluate the noise power values is definitely lower. Indeed, the SQNR expression determination is done only once. Then, the SQNR is evaluated quickly at each iteration of the WLO process through a mathematical expression. The method used in this paper to compute the accuracy allows obtaining automatically the quantization noise power expression from the signal flow graph (SFG) of the application. The SFG is obtained from the DFG by inserting the delay operations between data.

An analytical approach, to evaluate the fixed-point accuracy, has been proposed for linear time-invariant systems in [10] and for systems based on smooth operations in [13]. An operation is considered to be smooth if the output is a continuous and differentiable function of its inputs, as it the case for arithmetic operations. In the analytical expression of the output quantization noise power, the gains between the different noise sources and the output are computed from the impulse response of the system between the output and the noise sources. This approach has been implemented in a software tool to automate this process. Our numerical accuracy evaluation tool generates the analytical expression of the output quantization noise from the signal flow graph of the application. This analytical expression is implemented through a C function having the word-length $\mathbf{w}$ of all data as input parameters. This C code can be compiled and dynamically linked to the fixed-point conversion tool for the optimization process.

4.3. Optimization Techniques. In the proposed method, deterministic optimization approach is retained to lead to reasonable optimization times. However, classical greedy algorithms based on steepest-descent ($\max -1$ [14]) or mildest-ascent ($\min +1$ [14]) can lead to weak quality solutions. To improve the solution quality, a tabu search algorithm is used. The proposed method is based on three main steps. First, an initial solution $\mathbf{w}_{\min}$ is determined by computing the minimal word-length associated to each optimization variable $\mathbf{w}(i)$. Then, a mildest-ascent greedy algorithm is used to optimize the word-length. This algorithm starts with the initial solution $\mathbf{w}_{\text{mwc}}$ and leads to the optimized solution $\mathbf{w}_{\min}$. Finally, this optimized solution is refined by using a tabu search algorithm to obtain a better quality solution $\mathbf{w}_{\text{opt}}$.

In the first step, the minimum word-length combination $\mathbf{w}_{\text{mwc}}$ is determined with the algorithm presented in Algorithm 1. For that, all variable word-lengths $\mathbf{w}(i)$ are initially set to their maximal value $w_i^{\max}$. In that case, the accuracy constraint is satisfied. Then, for each variable, the minimum word-length still satisfying the accuracy is determined, all other variable word-lengths staying at their maximum value.

```
for i = 0 ... N_g − 1 do
    w ← (w_0^max ... w_i^max ... w_{N_g−1}^max)
    while λ(w) < λ_min do
        w(i) ← w(i) − 1
    end while
    w_mwc(i) = w(i) + 1
end for
```

Algorithm 1: Determination of the minimal word-length combination.

The mildest-ascent greedy algorithm presented in Algorithm 2 is used to optimize the word-length. Each variable $\mathbf{w}(i)$ is set to its minimal value $\mathbf{w}_{\text{mwc}}(i)$. With this combination, the accuracy constraint will surely not be satisfied anymore. But the advantage of this starting point is that word-lengths only have to be increased to get the optimized solution. At each step of the algorithm, the word-length of one operator is modified to converge to the optimized solution obtained when the accuracy constraint is fulfilled. A criterion has to be defined to select the best direction, that is, the operator for which the word-length has to be modified. The criterion is based on the computation of the discrete gradient of the cost and the accuracy. Let $\nabla_{k/\lambda}$ denote the gradient of the accuracy function

$$\nabla_{k/\lambda} = f_{\text{dir}}(\mathbf{w}_\Delta, \mathbf{w}) = \frac{\lambda(\mathbf{w}_\Delta) - \lambda(\mathbf{w})}{\mathbf{w}_\Delta(k) - \mathbf{w}(k)}, \tag{5}$$

with $\mathbf{w} = (\mathbf{w}_1, \ldots, \mathbf{w}_k, \ldots, \mathbf{w}_{N_g-1})$ and $\mathbf{w}_\Delta = (\mathbf{w}_1, \ldots, \mathbf{w}_k + 1, \ldots, \mathbf{w}_{N_g-1})$.

This gradient on the accuracy is used as a criterion for finding the best direction in the $\min +1$ *bit* algorithm [14]. Amongst deterministic algorithm, $\min +1$ *bit* does not always give a good result. It takes sometimes the wrong direction and returns poor quality results. To improve this criterion, the cost and the accuracy are taken into account as follows:

$$\nabla_k = \frac{\nabla_{k/\lambda}}{\nabla_{k/C}} = \frac{\lambda(\mathbf{w}_\Delta) - \lambda(\mathbf{w})}{C(\mathbf{w}_\Delta) - C(\mathbf{w})}. \tag{6}$$

This criterion selects the direction, which minimizes the cost increase and maximizes the accuracy increase.

Currently, all greedy algorithms used in WLO are monodirection, either steepest-descent ($\max -1$) or mildest-ascent ($\min +1$). To improve the solution obtained with these monodirection algorithms, the proposed algorithm is based on tabu search [15] and allows the movement in both directions.

The set T is the tabu list and contains *tabu variables*. When a variable $\mathbf{w}(k)$ is added in the tabu list, its value will not be modified afterwards and thus this variable is no longer considered in the optimization process. The term d represents the direction, ascending direction is used with $d > 0$, and descending direction is used with $d < 0$. The vector $\mathbf{w}_{\text{opt}}$ corresponds to the best combination of word-lengths

```
w ← w_mwc
while λ(w) < λ_min do
  for i = 0...N_g − 1 do
    w_Δ ← w
    w_Δ(i) ← w(i) + 1
    ∇_i ← ((λ(w_Δ) − λ(w)))/((C(w_Δ) − C(w)))
  end for
  k ← argmax_i(∇_i)
  w(k) ← w(k) + 1
end while
w_min = w
```

Algorithm 2: Mildest-ascent greedy algorithm for WL Optimization.

which have been obtained and C_{opt} is the cost associated with $\mathbf{w}_{\text{opt}}$.

The algorithm starts with the solution $\mathbf{w}_{\text{min}}$ obtained with the mildest-ascent greedy algorithm presented in Algorithm 2. This algorithm iterates until all the variable, $\mathbf{w}(k)$ are not in the tabu list (lines 22–23).

For each variable $\mathbf{w}(k)$, the possibility of a movement is analyzed in lines 8–15. If a variable $\mathbf{w}(k)$ reaches its maximal value $\mathbf{w}(k)^{\text{max}}$ in the ascending direction, or its minimal value b_k^{min} in the descending direction, this variable is added to the tabu list. In the other cases, a movement is possible and the metric for finding the best direction is computed in the lines 16–21. During this metric computation, the cost and the accuracy are compared, respectively, to the best cost C_{opt} and the accuracy constraint λ_{min}, and the best solution is updated if necessary.

After the computation of the metric ∇_k for each variable, the best possible direction is selected. For the ascending direction, the solution leading to the highest value of ∇_k is selected (lines 26–28). It corresponds to the solution leading to the best trade-off between the increase of accuracy and the increase of cost. For the descending direction, the solution leading to the lowest value of ∇_k is selected (lines 33–35). The aim is to reduce the cost without reducing too much the accuracy.

As soon as the accuracy constraint is crossed, the direction is inverted (lines 29–31 and 36–38). In this case, the operator is added to the tabu list if the direction is ascending (lines 29–31). This algorithm iterates until all the variables $\mathbf{w}(k)$ are not in the tabu list.

5. Experiments

5.1. Word-Length Optimization Technique. First the quality and the efficiency of the WLO technique based on the tabu search algorithm (Algorithms 1, 2 and Algorithm 3) is evaluated on different benchmarks. The tested applications are a eight-order Infinite Impulse Response filter (IIR) implemented through four second order cells as presented in Figure 6, a Fast Fourier Transform (FFT) on 128 points using a radix-2 and decimation in frequency (DIF) structure and a Normalized Least Mean Square adaptive filter (NLMS) with 128 taps using an adaptation step of 0.5. The implementation cost C_{tabu} and the optimization time T_{tabu} are measured for the proposed technique and compared with the results C_{greedy} and T_{greedy} obtained with only the greedy algorithm corresponding to Algorithms 1 and 2. The number of variables inside the optimization problem is adjusted by grouping together operations. Let I_{tabu} denote the improvement of the solution quality due to the tabu search algorithm such as

$$I_{\text{tabu}} = \frac{C_{\text{greedy}} - C_{\text{tabu}}}{C_{\text{greedy}}}. \tag{7}$$

Let OC_{tabu} denote the over-cost in terms of optimization time due to the tabu search algorithm

$$OC_{\text{tabu}} = \frac{T_{\text{tabu}} - T_{\text{greedy}}}{T_{\text{greedy}}}. \tag{8}$$

For the different experiments, the input signal is normalized in the interval $]-1, 1[$ and different values for the SQNR are tested between 40 to 60 dB by step of 1 dB. The results presented in Table 2 show the improvement obtained with the tabu search algorithm. In our experiments, the improvement can reach up to 65%. The optimization time is significantly increased compared to the greedy algorithm, but the execution time is still reasonable and low compared to other combinatorial optimization approaches like stochastic algorithms.

5.2. Illustrative Example for HLS under Accuracy Constraint. To illustrate the proposed method, an infinite impulse response (IIR) filter example is detailed. This filter is an eight-order IIR filter implemented as four cascaded second-order cells. The signal flow graph (SFG) of this IIR filter, presented in Figure 6, contains 20 multiplications and 16 additions. The method presented in Section 3 is used to obtain the data dynamic range and the binary point-position and thus, a correct fixed-point specification. The SQNR analytical expression is determined and the accuracy constraint is set to 60 dB. The Stratix FPGA is used for the experiments with no dedicated resources.

Firstly, the different operation word-lengths is optimized for a spatial implementation. In this case, an operator is used for each operation. The obtained word-lengths $w_{o_i}^{\text{opt}}$ are presented in Figure 7 (number between parentheses). For the first iteration, a group is defined for each operation type and the group word-lengths are optimized. Thus, multiplications are executed on a 17×17-bit multiplier and additions on 20-bit adders. The minimal system clock frequency is set to 200 MHz, so the operator latency is a multiple of 5 ns. The multiplier and adder propagation times are, respectively, equal to 10.3 ns and 2.5 ns, so the latency of the multiplier and adder is set, respectively, to 3 and 1 clock cycles. The hardware synthesis for this fixed-point specification leads to the scheduling presented in Figure 7. For a 70 ns time constraint, five multipliers and two adders are needed. In the next step, five new groups for multiplications and two new groups for the additions are defined. These groups,

(1): $\mathbf{w} = \mathbf{w}_{\min}$
(2): $C_{\text{opt}} = C(\mathbf{w})$
(3): $\mathbf{w}_{\text{opt}} \leftarrow \emptyset$
(4): $T \leftarrow \emptyset$ {tabu operators}
(5): $d \leftarrow (\lambda(\mathbf{w}) > \lambda_{\min})\ ?\ -1\ :\ 1$ {direction selection}
(6): **while** $|T| < N$ **do**
(7): **for all** $1 \le k \notin T \le N$ **do** {computation of ∇k}
(8): $\mathbf{w}_\Delta \leftarrow \mathbf{w}$
(9): **if** $\mathrm{d} > 0 \wedge \mathbf{w}(k) < \mathbf{w}^{\max}(k)$ **then**
(10): $\mathbf{w}_\Delta(k) \leftarrow \mathbf{w}(k) + 1$
(11): **else if** $\mathrm{d} < 0 \wedge \mathbf{w}(k) > \mathbf{w}^{\min}(k)$ **then**
(12): $\mathbf{w}_\Delta(k) \leftarrow \mathbf{w}(k) - 1$
(13): **else**
(14): $T \leftarrow T \cup \{k\}$
(15): **end if**
(16): **if** $k \notin T$ **then** {Computation of the metric for direction selection}
(17): $\nabla_k \leftarrow f_{\text{dir}}(\mathbf{w}_\Delta, \mathbf{w})$
(18): **if** $\lambda(\mathbf{w}_\Delta) > \lambda_{\min}$ **then**
(19): update of C_{opt} and $\mathbf{w}_{\text{opt}}$ if necessary
(20): **end if**
(21): **end if**
(22): **end for**
(23): **if** $|T| = N$ **then** {all the variable are in the tabu list}
(24): stop
(25): **end if**
(26): **if** $\mathrm{d} > 0$ **then** {selection of the best solution}
(27): $j \leftarrow \text{argmax } \nabla_k$
(28): $\mathbf{w}(j) \leftarrow \mathbf{w}(j) + 1$
(29): **if** $\lambda(\mathbf{w}) > \lambda_{\min}$ **then**
(30): $\mathrm{d} \leftarrow -1$ {change of direction if $\lambda_{\min}$ is now fulfilled}
(31): $T \leftarrow T \cup \{j\}$
(32): **end if**
(33): **else**
(34): $j \leftarrow \text{argmin } \nabla_k$
(35): $\mathbf{w}(j) \leftarrow \mathbf{w}(j) - 1$
(36): **if** $\lambda(\mathbf{w}) < \lambda_{\min}$ then
(37): $d \leftarrow 1$ {change of direction if $\lambda_{\min}$ is no longer fulfilled}
(38): **end if**
(39): **end if**
(40): **end while**
(41): **return** $\mathbf{w}_{\text{opt}}$

Algorithm 3: Tabu search for solution improvement in WL optimization.

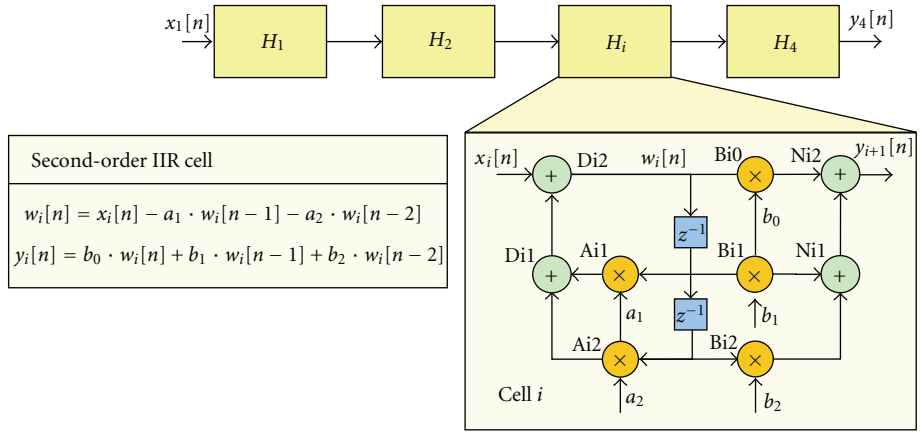


Figure 6: Signal flow graph of the 8th-order IIR filter.

Table 2: Comparison of the tabu search and greedy algorithms in terms of quality improvement I_{tabu} and execution time over-cost (OC_{tabu})

Benchmark	N_g	I_{tabu}	T_{greedy} (s)	T_{tabu} (s)	OC_{tabu}
IIR	14	2.9%	4.5	14.6	219%
	18	6.5%	35.1	83.2	137%
	36	6.6%	78.3	177.1	126%
FFT	8	65.7%	26.1	62.8	141%
	12	62.4%	57.3	163.4	185%
	20	0.6%	57.4	128.8	124%
NLMS	13	12.5%	16.8	37.1	120%
	25	16.3%	76.5	152.4	99%
	49	9.65%	286.5	579.6	102%

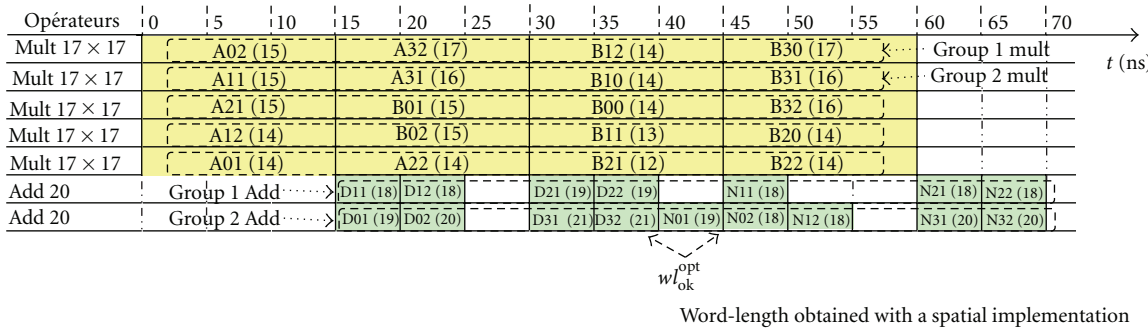


Figure 7: Scheduling of the 8th-order IIR filter obtained for the first iteration.

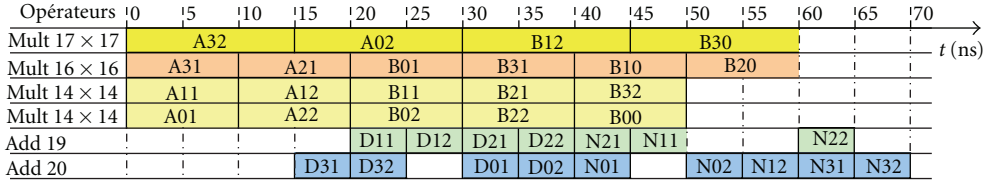


Figure 8: Scheduling of the 8th-order IIR filter obtained for the second iteration.

presented in Figure 7, are built depending on the word-lengths obtained for the spatial implementation and the operation mobility.

A group WLO under accuracy constraint is carried-out for these seven groups. This optimization results in lower word-lengths. The five multiplication group word-lengths are, respectively, 17, 16, 15, 14, and 14 bits. The HLS for this new fixed-point architecture leads to the scheduling presented in Figure 8. Given that, below 16 bits, multipliers have a critical path lower than 10 ns, that is, 2 clock cycles, and so only four multipliers are now needed. Therefore, this architecture uses one less multiplier than the previous one. The word-length reduction combined with the decrease in the number of operators reduces the area by 35%.

A uniform word-length architecture optimization leads to five multipliers and two adders with a precision of 19 bits. Compared to this architecture, the total area saved on operators, with the proposed method, is 47%. A sequential approach carrying-out a word-length optimization in the case of spatial implementation and a high level synthesis leads to the same number of operators as our approach. Nevertheless, the word-length of the operators is higher than those obtain with our approach. Indeed, the operator word-length is imposed by the operation with the greater word-length. Consequently, compared to this sequential approach, the total area saved on operators, with the proposed method, is 6%. These results show the interest of using multiple word-length architecture and efficiency of the proposed method, which couples HLS and WLO.

5.3. Pareto Frontier. The proposed method for multiple word-length HLS generates, for a given timing constraint (latency or throughput) and accuracy constraint, an architecture optimized in terms of implementation cost. By testing different accuracy and timing constraints, the Pareto frontier associated to the application can be obtained. The different trade-off between implementation cost, accuracy, and latency can be analyzed from this curve.

The results obtained for the searcher module of a WCDMA receiver are presented in Figure 9. The data flow graph of the application can be found in [16]. The targeted architecture is an FPGA and only LUTs are considered. The results show an evolution by plateau. For the latency, the plateaus are due to the introduction of one operator or

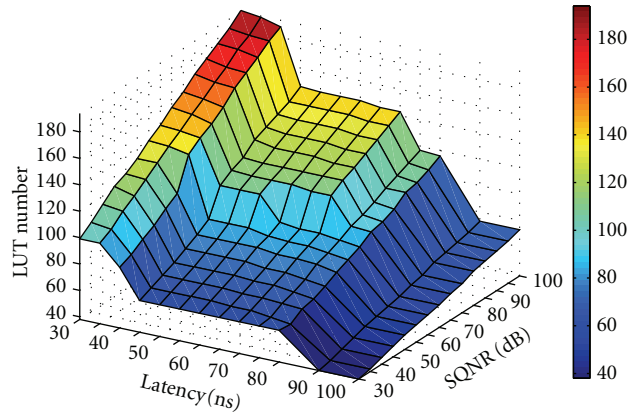


Figure 9: Implementation cost according to the accuracy and timing constraint for WCDMA searcher module.

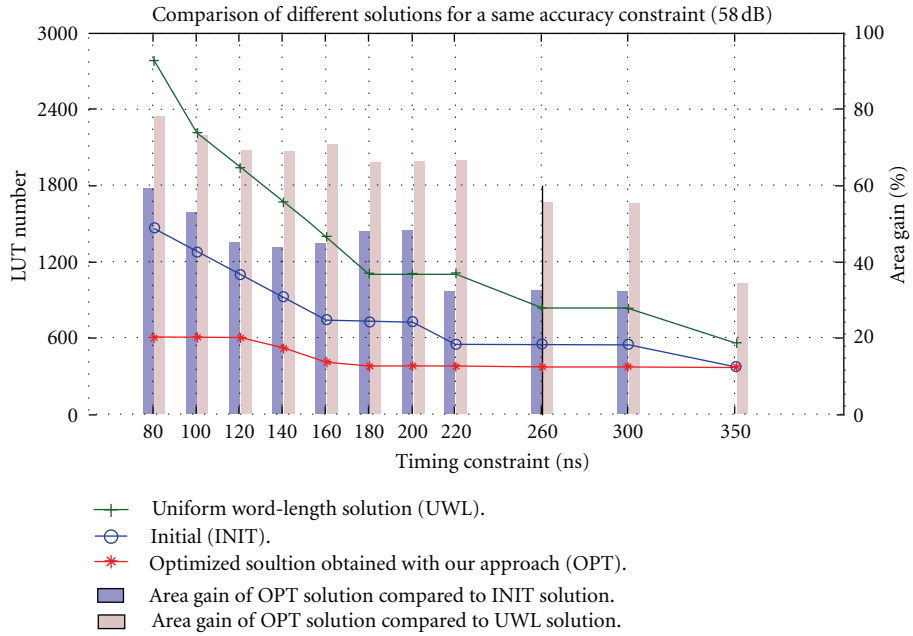


Figure 10: Comparison of the solution obtained by the proposed method (OPT) with a UWL solution and with the initial solution (INIT) for an FFT with an SQNR of 58 dB.

several in parallel, to reduce the application latency. For the accuracy, the evolution is piecewise linear. The smooth evolution is due to the gradual increase of the operator word-length to reach the accuracy constraint. The evolution is linear for this application because the architecture is dominated by addition and subtraction operations, which have a linear implementation cost according to the operand word-length. As for the latency, the abrupt changes in the evolution are due to to the introduction of one operator or several in parallel to reach the constraints. The accuracy increase requires operators with greater word-length and thus leads to higher operator latency. Consequently, when the operator latency increase does not satisfy the global timing constraint, one or more additional operators are required. The location of these abrupt changes in the Pareto frontier is tightly linked to the clock period. The discretization of the operator latency in an integer number of cycles leads to the occurrence of abrupt changes.

5.4. Comparison with Other Solutions. In this section, the solution obtained with the proposed method is first compared with a classical method based on a uniform word-length (UWL) and then with the solution using a single word-length for each type of operation. As in [4, 8], to evaluate the efficiency of the proposed method, the obtained solutions are compared with the UWL solutions. In this last case, a single word-length is used for all data. For a Fast Fourier Transform (FFT), the UWL solution with a 16-bit precision leads to a SQNR of 58 dB. The cost is evaluated with the proposed method (OPT) and with the UWL method for this accuracy constraint of 58 dB and for different timing constraints. The results are presented in Figure 10. For this

Table 3: Area reduction obtained after several iterations compared to the first iteration. Mean and maximal values obtained for different accuracy and timing constraints

Application	$l_{\min}$ (ns)	$\lambda_{\min}$ (dB)	Area Reduction	
			Mean	Max
FIR 32-taps	[100, 200]	[30, 100]	18%	35%
FFT radi-2, DIF, 8 points	[80, 500]	[30, 100]	28%	50%
IIR 8th-*order*	[50, 100]	[30, 100]	22%	47%
Complex correlator	[70, 250]	[10, 100]	10%	20%

application, the proposed method performs better with a gain on the implementation cost between 33% and 78%. When the timing constraint is low, several operators are used for the same kind of operations and the multiple word-lengths approach benefits from the possibility to distribute different word-lengths to each operator. When the timing constraint is high, the number of operators in the architecture is lower and the difference between the OPT and MWL solutions decreases. In the sequential method used in [8, 17], the word-lengths are first optimized and then the architecture is synthesized. The results presented in [8] lead to a gain of up to 52% compared to the UWL solution, and the results presented in [17] leads to a gain of up to 45% compared to the UWL solution. These results show that the combination of the WLO and the HLS in the proposed method gives better results than the sequential method. In [4], the WLO and HLS processes are combined through a simulated annealing optimization and the gain obtained compared to the UWL solution is between 22% and 77%. The proposed method leads to similar gains but with significantly less iterations required for a good solution. Moreover, in our case the HLS process is not modified and existing academic or commercial tools can be directly used.

To analyse the efficiency of the proposed iterative method and the interest of coupling WLO and HLS, the optimized solution obtained after several iterations and the solution obtained at the first iteration are compared. The solution obtained at the first iteration (INIT) corresponds to the case where a single word-length is used for all the operators of the same type. In this case, the operation binding on operators is not taken into account. The architecture area reduction compared to the first iteration is measured and the results obtained for the Stratix FPGA are given in Table 3 for different digital signal processing kernels. The complex correlator computes the correlation between complex signal like in a WCDMA receiver. The experiments are carried out for different accuracy and timing constraints and the maximal and mean values are reported. The proposed method can reduce the architecture area up to 50% in the case of the FFT. In average, the reduction is between 10% and 28%. These results show the efficiency of the iterative method to improve the cost implementation. Gradually, the information collected from the previous iterations allows the convergence to an efficient operation grouping, which improves the HLS.

6. Conclusion

In this paper, a new HLS method under accuracy constraint is proposed. An iterative process is used to link HLS and WLO. This coupling is achieved through an iterative process. To reduce significantly the optimization time compared to the simulation-based methods, the accuracy is evaluated with an analytical method.

The efficiency of proposed method is shown through experiments. Compared to classical implementations based on a uniform word-length, the proposed method reduces significantly the number of resources used to implement the system. These results show the relevance of using multiple word-length architecture. The interest of coupling HLS and WLO is shown on different digital signal processing kernels. This technique reduces the number of operators used in the architecture and also reduces the latency.

References

[1] D. Novo, B. Bougard, A. Lambrechts, L. Van Der Perre, and F. Catthoor, "Scenario-based fixed-point data format refinement to enable energy-scalable software defined radios," in *Proceedings of the IEEE/ACM Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 722–727, Munich, Germany, March 2008.

[2] D. Novo, M. Li, B. Bougard, L. Van Der Perre, and F. Catthoor, "Finite precision processing in wireless applications," in *Proceedings of the IEEE/ACM Conference on Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 1230–1233, April 2009.

[3] K. 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, pp. 921–930, 2001.

[4] G. Caffarena and C. Carreras, "Architectural synthesis of DSP circuits under simultaneous error and time constraints," in *Proceedings of the 18th IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC '10)*, pp. 322–327, Madrid, Spain, June 2010.

[5] B. Le Gal, C. Andriamisainat, and E. Casseau, "Bit-width aware high-level synthesis for digital signal processing systems," in *Proceedings of the IEEE International SOC Conference (SOCC '06)*, pp. 175–178, September 2006.

[6] P. Coussy, G. Lhairech-Lebreton, and D. Heller, "Multiple word-length high-level synthesis," *Eurasip Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 916867, 11 pages, 2008.

[7] B. Le Gal and E. Casseau, "Latency-sensitive high-level synthesis for multiple word-length DSP design," *Eurasip Journal on Advances in Signal Processing*, vol. 2011, Article ID 927670, 11 pages, 2011.

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

[9] G. A. Constantinides, P. Y. K. Cheung, and W. Luk., *Synthesis and Optimization of DSP Algorithms*, Kluwer Academic, 2004.

[10] D. Menard, R. Rocher, and O. Sentieys, "Analytical fixed-point accuracy evaluation in linear time-invariant systems," *IEEE Transactions on Circuits and Systems I*, vol. 55, no. 10, pp. 3197–3208, 2008.

[11] G. Caffarena, G. Constantinides, P. 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.

[12] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, *High-Level Synthesis From Algorithm to Digital Circuit, chapter GAUT: A High-Level Synthesis Tool for DSP Applications From C Al-gorithm to RTL Architecture*, Springer, Amsterdam, The Netherlands, 2008.

[13] R. Rocher, D. Menard, P. Scalart, and O. Sentieys, "Analytical accuracy evaluation of Fixed-Point Systems," in *Proceedings of the European Signal Processing Conference (EUSIPCO '07)*, Poznan, Poland, September 2007.

[14] 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, 2002.

[15] F. Glover, "Tabu search—part I.," *INFORMS Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[16] H.-N. Nguyen, D. Menard, R. Rocher, and O. Sentieys, "Energy reduction in wireless system by dynamic adaptation of the fixed-point specification," in *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing (DASIP '08)*, Bruxelles, Belgium, November 2008.

[17] 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.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 862469, 26 pages
doi:10.1155/2012/862469

Research Article

Automated Generation of Custom Processor Core from C Code

Jelena Trajkovic,[1,2] Samar Abdi,[3] Gabriela Nicolescu,[2] and Daniel D. Gajski[1]

[1] *Center for Embedded Computer Systems, University of California, Irvine CA 92697, USA*
[2] *École Polytechnique de Montréal, Montreal, QC, Canada H3C 3A7*
[3] *Electrical and Computer Engineering Department, Concordia University, Montreal, QC, Canada H4B 1R6*

Correspondence should be addressed to Jelena Trajkovic, jelena.tr@gmail.com

Received 7 June 2011; Revised 14 November 2011; Accepted 21 November 2011

Academic Editor: Yuan Xie

We present a method for construction of application-specific processor cores from a given C code. Our approach consists of three phases. We start by quantifying the properties of the C code in terms of operation types, available parallelism, and other metrics. We then create an initial data path to exploit the available parallelism. We then apply designer-guided constraints to an interactive data path refinement algorithm that attempts to reduce the number of the most expensive components while meeting the constraints. Our experimental results show that our technique scales very well with the size of the C code. We demonstrate the efficiency of our technique on wide range of applications, from standard academic benchmarks to industrial size examples like the MP3 decoder. Each processor core was constructed and refined in under a minute, allowing the designer to explore several different configurations in much less time than needed for manual design. We compared our selection algorithm to the manual selection in terms of cost/performance and showed that our optimization technique achieves better cost/performance trade-off. We also synthesized our designs with programmable controller and, on average, the refined core have only 23% latency overhead, twice as many block RAMs and 36% fewer slices compared to the respective manual designs.

1. Introduction

In order to implement an application, the designer typically starts from an application model and a set of constraints, such as performance and cost (Figure 1). One of the primary design decisions is to define the hardware-software (HW/SW) partitioning of the application [1]. In case of noncritical applications, the entire application may be implemented in software. The designer selects the target processor and compiles the applications for the fixed instruction set of the selected processor. The generated binary is executed on the target processor. This is the most flexible solution, since the target processor is programmable. As such, the designer may change the application code and recompile to modify the implementation. In case when part of an application, or the entire application, has tight performance, power, or area constraints, the designer may decide to implement the critical part of the application in hardware. For hardware implementation, the designer may use pre-existing intellectual property (IP), use high-level synthesis (HLS) tools or manually design the hardware. The generated hardware is highly optimized, but nonprogrammable.

In order to obtain the advantages of both approaches, that is, programmability and high design quality, the designer may opt for application-specific processor (ASP). Application-specific processor cores are being increasingly used to address the demand for high performance, low area, and low power consumption in modern embedded systems. In case of application-specific instruction set processors (ASIPs), the designer first decides on the instruction set (IS) then generates the hardware and the compiler to support the chosen IS. The other possibility is to generate data path first, and then to generate the controller either manually or automatically [2]. The advantage of the latter method is the possibility to remove IS and decoder from the design process and the design itself, respectively. Removing the decoder significantly simplifies the design and verification. Furthermore, the controller generates a set of control signals that directly drives the data path, which allows for having any combination of control signals possible, not just the

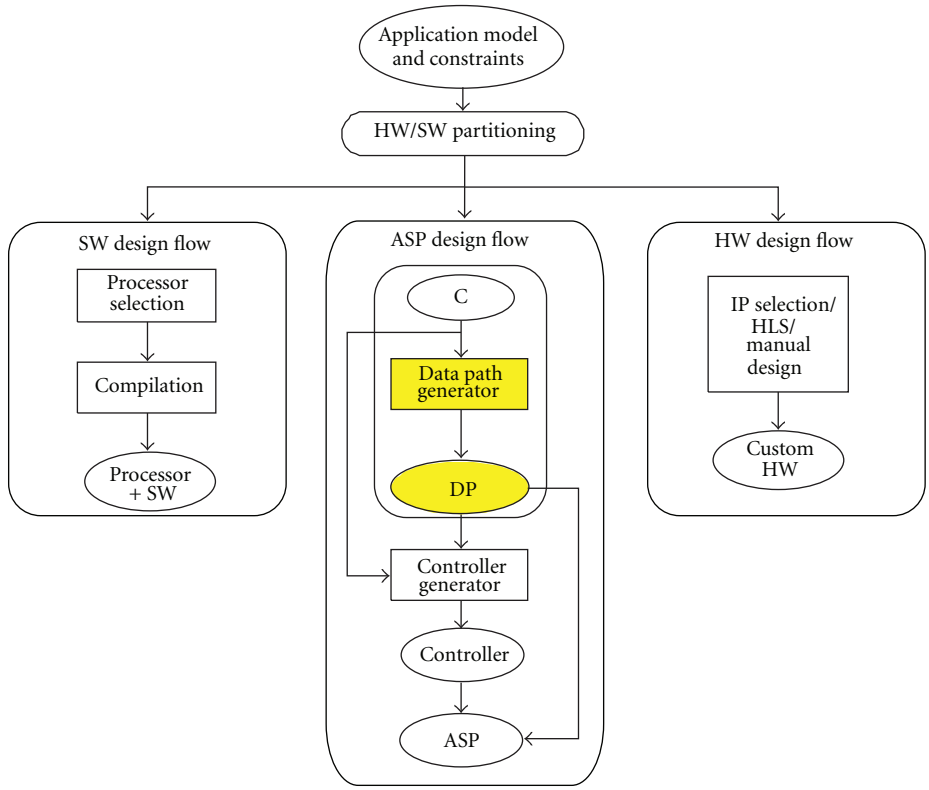


Figure 1: Proposed design technique within system-level design flow.

combinations that are allowed by instructions from a predefined IS. Moreover, it facilitates having compiler [3] that does not need to be modified every time that IS changes. Therefore, in this work, we adopt the methodology where the data path gets created separately from the controller.

In general, the challenges of ASP design are as follows:

(1) design of the data path, that is, selection of the best set of components (functional units, register files, buses, memories, etc.) and connections;

(2) design of the controller;

(3) ability to scale well with large code size;

(4) controllability of the design process.

In this work, we deal with (1) and (4), and the aspect of (3) that applies to data path creation, where (2) has been published in [3, 4].

1.1. Custom Processor Design Methodology. The design of application-specific cores is nontrivial and the problem is further exacerbated by the size and complexity of the application as well as the short time to market. Traditional C to RTL automation tools have so far had little impact. This is because they combine the optimization of data path architecture and controller, which makes the process unscalable and uncontrollable. We overcome this problem by applying a design strategy where the data path and controller are designed separately. The architecture is derived by analyzing the C code and designer-specified constraints. The C code is then compiled into either control words (control word is a collection of control signals that drive the data path) for programmable cores or FSM for hardwired cores.

Our target processor core is similar to an ASIP, with the exception of the instruction decoder. The C application is directly compiled into microcode that drives the data path. This approach removes the unnecessary restriction of an instruction set and provides the abovementioned advantages. The target processor core template is shown in Figure 2. First, we construct the data path on the right-hand side by selecting, configuring, and connecting various functional and storage units. Then, we develop the microcoded or hardwired controller (on the left-hand side) for the constructed data path. During core construction, the data path is automatically refined to meet the given performance and resource constraints. The generated data path is *pareto-optimal* for the given application, but it may execute any other application if it contains all the components required by the application. However, the execution may not be optimal. By separating data path and controller design, we allow simplified optimization of the controller by removing the data path optimization parameters from the equation. The scheduling and binding, that is, controller generation, are performed once the data path has been designed. The controller generation may be manual or automated, as presented in [3]. Details of separation of data path generation from scheduling and binding may be found in [2, 5].

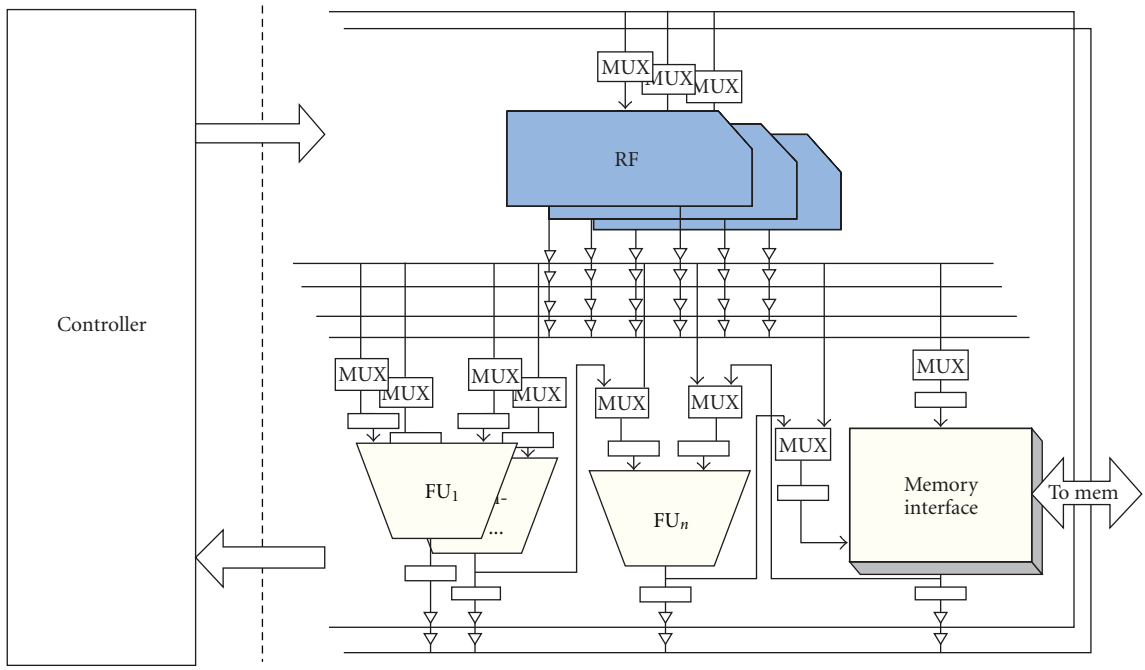


Figure 2: Proposed processor core template: the data path and the controller are created separately.

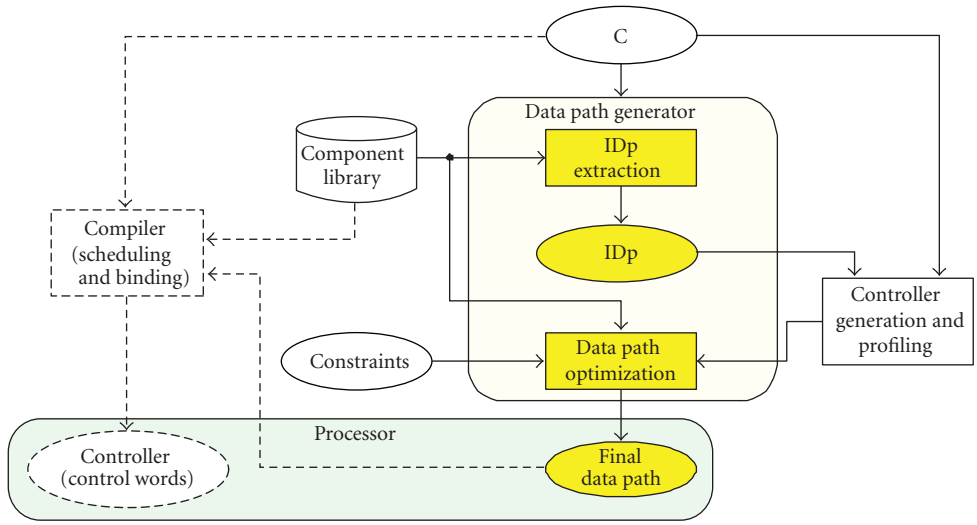


Figure 3: Data path extraction steps.

Automating the design process in the proposed way has several advantages.

(1) Designers use their expertise to guide the tool instead of performing cumbersome and error-prone HDL coding.

(2) The tool automatically produces a design in a fraction of time while having only marginal performance degradation and almost the same total area as the manual design.

(3) The designer may change *only* the constraints and iterate over the optimization phase.

Figure 3 shows the steps of proposed extraction technique. In the first step, called *Initial Data path* (IDp) *Extraction* (Section 2), source code for a given application is analyzed in order to extract code properties that will be mapped onto hardware components and structures and the *Initial Data path* is created. The Initial Data path is used for controller generation and profiling of given application code. The profiled code is then analyzed and the data path undergoes several iterations of refinement and estimation during the *Data path Optimization* (Section 3) step. Data path Optimization step first selects the portion of code to be optimized (Section 3.1), converts partial resource constraints to timing overhead (Section 3.3), and estimates

execution characteristics of intermediate candidate designs (Section 3.4) until the specified overhead is met. The final data path is used for scheduling and binding, and controller generation. As explained above, the controller generation is out of scope of this work.

2. Initial Data Path Extraction

The stepwise process of creating the initial data path is shown in Figure 4. Based on a target-independent scheduling of the C-code instructions, we identify a set of properties for the given code. Then, we use the mapping function to map the code properties to available hardware elements in the Component Library and to generate a set of hardware components and structures/templates that correspond to the reference C code.

The code properties include operators and their frequency, data types and their values, control and data dependencies, existence of loops, loop nests, size of the loop body, and the available parallelism. The hardware elements include different types of parameterizable functional units, like adders, multipliers, and comparators; storage units, such as registers, register files, and memory; interconnect components, like busses, multiplexers, and tristate buffers. We consider different data path structures/templates: templates with a different connectivity schemes, like bus-based, multiplexer-based, and dedicated connections; templates with pipelined data path, pipelined units or both, and so forth. For instance, the available parallelism translates into the number of instances of functional units, the number of registers, and the number of register file ports. It may also determine, together with type of dependencies in the code, if the data path should be pipelined. Also, the data types and number of used bits to represent the data determines the bitwidth of used components and buses.

The maximum requirements of a given application is obtained from the application's "as late as possible" (ALAP) schedule (produced by Pre-Scheduler). The ALAP schedule is in a form of Control Data Flow Graph (CDFG) and is target independent. The underlying assumption is that there are sufficient resources to execute all existing operations. Front end of any standard compiler may be used as a Pre-Scheduler. In this case, the CDFG is output of front end of the Microsoft Visual C++ compiler. We choose ALAP because it gives good notion of the operations that may be executed in parallel. Please note that only the properties required for the data path generation (described in this section) are extracted from the CDFG.

The IDp *Extractor* traverses the given ALAP schedule, collecting the statistics for each operation. For each operation (addition, comparison, multiplication, etc.) maximum number of its occurrences in each cycle is computed. The tool also records the number of potential data transfers for both source and destination operands. For example, if the ALAP schedule assigns three additions in the same cycle, then three units that perform addition will be allocated, together with sufficient number of busses, register files, and memories to provide for reading and writing of all operands. However, since the resources are selected from the elements available in the Component Library, the resulting architecture may have different number of instances than desired. One such example would be the case where the application requires three adders, but only a register file with one input and two output ports is available, and no forwarding is possible. Therefore, it would be reasonable to allocate *only one* adder, because there would be no sources for operands of the remaining two adders.

In addition to application's schedule, Component Library (CL) is another input of the Initial Data path Extraction. The Component Library consists of hardware elements, that are indexed by their unique name and identifier. The record for each component also contains its type, number of input and/or output ports, and name of each port. In case of a functional unit, a hash table is used to link the functional unit type with the list of all operations it may perform. Later in the section, we also present a set of heuristics that measure how well the available storage components match the given requirements. The heuristics use quantifiers for the code properties, translate them to the required number of hardware elements, and map them to the available components in the Component Library.

We chose a load-store data path model. This widely used model has specialized operations that access memory (load and store instructions) and any other operation reads operands from the register file and writes the results back to the register file. Also, we explore the data path models that do not have any forwarding path. Therefore, to ensure that the interconnect is not a bottleneck, for the Initial Data path, we perform a greedy allocation of connectivity resources (Figure 7). This means that the output ports of all register files are connected to all the source buses. Similarly, the input ports of all register files are connected to all the destination busses. The same connection scheme applies to the functional units and the memory interface.

2.1. Application Requirements. In order to design an application-specific processor core, we must first determine the application requirements. These requirements must be quantized in some form for analysis and eventual selection of matching components. There are primarily three aspects of the application that we are concerned with, namely, computation, communication, and storage. In terms on the architecture, these aspects translate to operator usage, operand transfer, and type of storage.

The set of properties of the C code include data types, operators, variables, parallelism, loops, and dependencies. The components include functional units, storage elements (registers, register files, memories), and interconnect like buses, multiplexers, and tristate buffers, while structures/templates refer to different connectivity templates, like bus-based interconnect, multiplexer-based interconnect, dedicated connections, data path or component pipelining, and load/store architecture model. For example, the data types and number of bits used to represent them would determine the bit width of used components; the available parallelism would influence the number of instances of

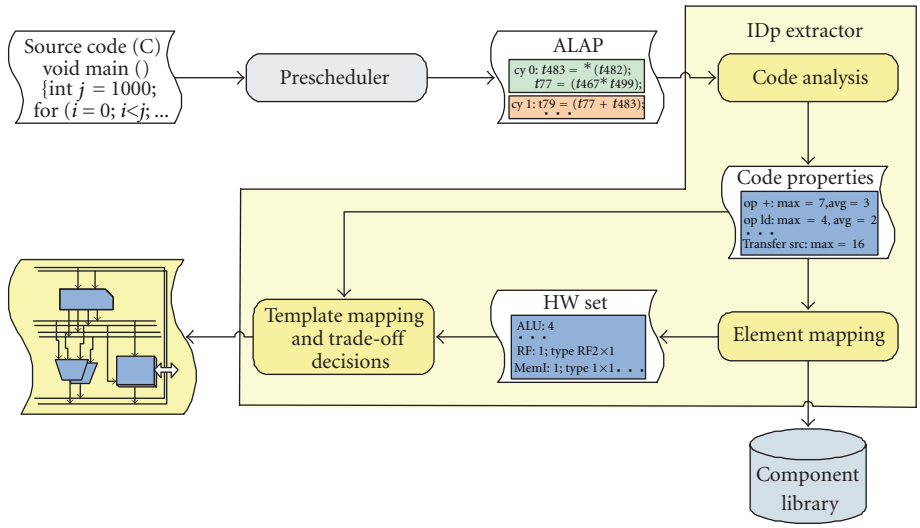


Figure 4: Initial data path (IDp) extraction flow.

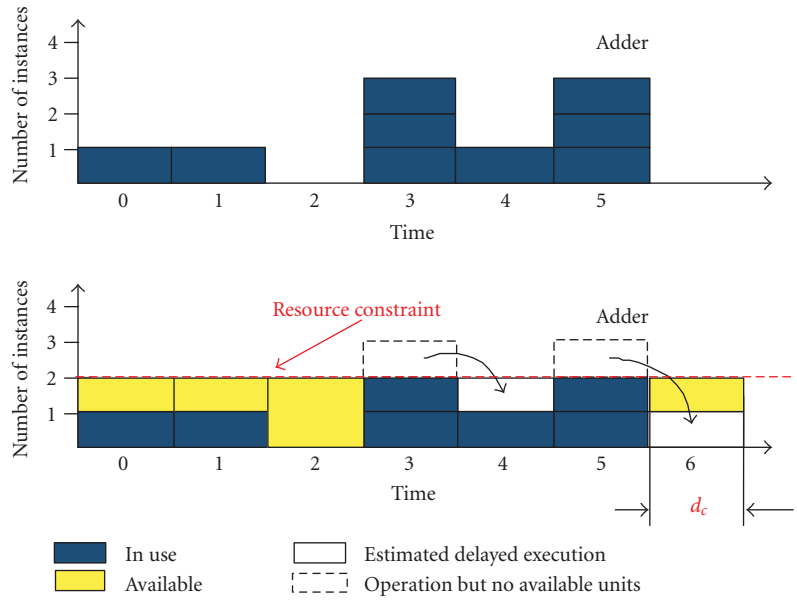


Figure 5: d_c Computation for given resource constraint.

functional units, number of registers or register file ports, and pipelining.

While the selection of function units and register files corresponds directly to the operations and variables, respectively, the communication needs more analysis. The operands for each operation are classified into either *source operands* or *destination operands*. The simplest case is the arithmetic operation, which typically has two or more source operands and one destination operand. The load operation has one source operand, the memory address, and one destination operand, the read variable. On the other hand, the store operation has two source operands: the address and the variable.

In a typical load/store architecture, like the one used in our work, the source and destination operands correspond to buses that go in and out of the register file. Each transaction, such as the fetching or storing of operand, must be performed using a bus. If multiple operations are scheduled to execute in parallel, multiple source and destination buses are required. One of the code properties we consider for design is the maximum number of source and destination buses that will be used. These numbers can be obtained easily by analyzing the prescheduled C source code.

The first step is to extract properties form the C code. Code property may be extracted from high level code, its

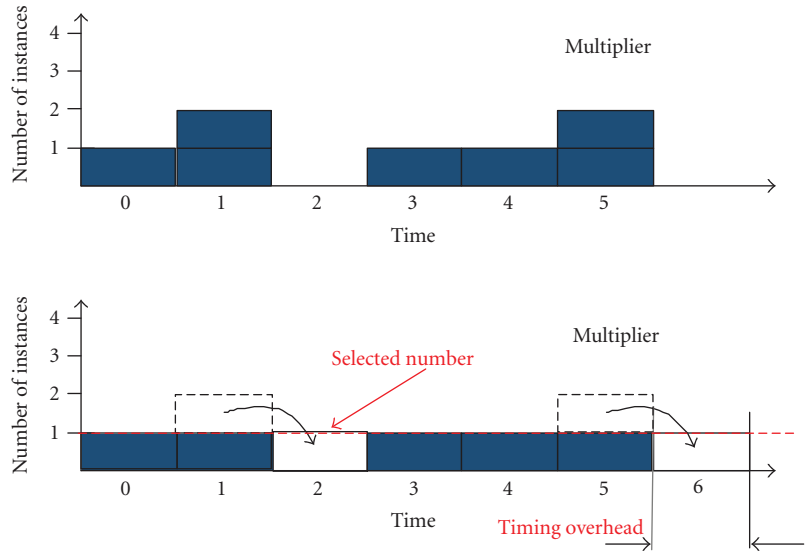


Figure 6: Balancing the number of instances of multiplier for given timing overhead.

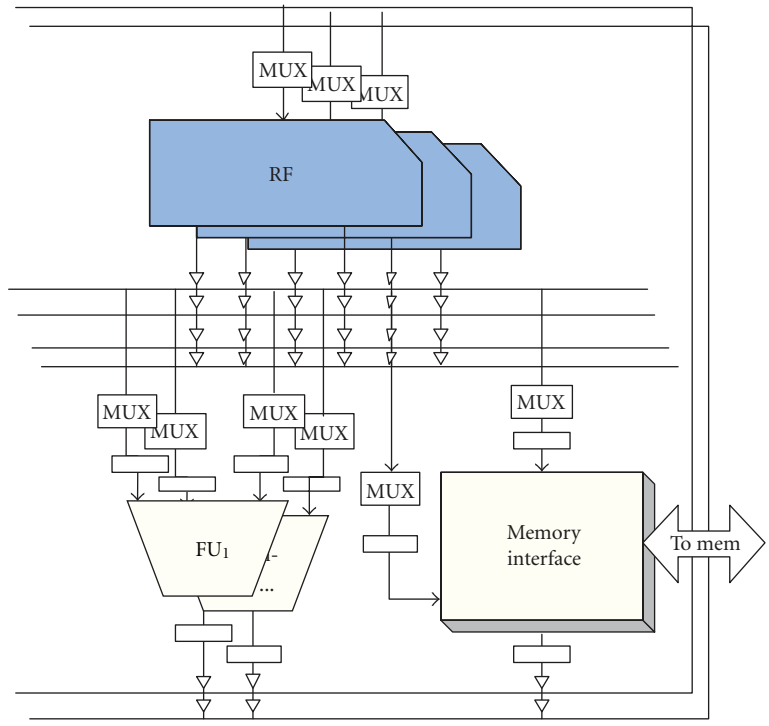


Figure 7: Used data path template.

CDFG, or any other intermediate representation generated by a compiler front end. We chose to start form an architecture-independent schedule that assumes no resource constraints, such as ASAP and ALAP. We chose ALAP as the starting point since our experiments show that the parallelism is more evenly distributed across cycles than in corresponding ASAP. The following properties are extracted from code's ALAP schedule:

(i) *OP*: a set of operations for the given application code;

(ii) m_{op}: the maximum number of concurrent usages of operand *op*;

(iii) m_s and m_d: the maximum number of concurrent data transfers of the source and destination operands, respectively.

2.1.1. Example. We present results for m_{op} value for various operations in typical multimedia functions to provide an idea of the application profile. The example shown here is

Mp3. The size is over 10000 lines (Table 4). The *Mp3* decoder is the example for decoding mp3 frames into pulse-code modulated (PCM) data for speaker output.

The operations are divided into separate classes depending on their type and implementation requirements. We classify the operations into *arithmetic, logic, comparisons, and load/store*. The tables below present the maximum number of concurrent operations (m_{op}) of each operation type.

The m_{op} values for the arithmetic operations, such as *add, subtract, multiply, divide,* and *remainder* are shown in the left-hand side of Table 1. It can be seen that *Mp3* has division operation that may require a divider or be implemented as software library in C. We can also note the high degree if concurrency. This is because the loop in the function is completely unrolled and all the array indexes are changed to different variables for optimization in software. A tool that performs component allocation for maximum performance based on available concurrency would produce a huge design. This extreme example points to the need for designer controllability in making component allocation decisions. The example also points to the fact that application optimizations for traditional processor architectures may not necessarily produce better processor designs. Indeed, the quality of application-specific processor design strongly depends on the structure of the C code in consideration.

The right-hand side of Table 1 shows the concurrency available for logic or bit-wise operations. We can see that not many of thebit-wise operations have been used, even though they are computationally less expensive than arithmetic operations. Some compiler optimizations make transform arithmetic operations involving constants into shifts and other logic operations. This is called strength reduction and is also useful code optimization that potentially may result in lower cost processor designs.

The left-hand side of Table 2 shows the concurrency available in load/store operations for the benchmarks. This concurrency typically does not translate into execution since memories typically have 1-2 ports for reading or writing data.

There is no concurrency available in the comparison operations as seen from right-hand side of Table 2. This is to be expected from the C code structure. Comparison operations are typically used for condition evaluation in if-then-else statements or loops. In the case of loops, the comparison is the only operation in a basic block. Since we are analyzing an ALAP for each basic block of the application, we consider concurrency inside the basic block only. In the case of if-then-else statements, the comparison is the last operation in a basic block that does not have any other comparisons. Therefore, comparisons cannot be executed in parallel due to the ALAP data structure constraints.

Available concurrency in variable reads and writes is shown by the number of concurrent Read Variable operation (262) and the number of Write Variable operations (196). For each basic block ALAP, the operations in the starting state are read operations that fetch the operands from the register file. Similarly, the terminating states write the variables back to the register file. The larger the basic block, the higher the possible concurrency in reading and writing variables, especially if the variables inside the basic block are independent. For our example, we have large basic blocks. This is due to the loop unrolling optimization on the source code, as discussed earlier. Furthermore, the *array elements*

Table 1: Application requirement for arithmetic and logic operations for Mp3 application.

OP—operation	m_{op}	OP—operation	m_{op}
Add	223	And	2
Sub	109	Or	1
Mul	73	Xor	1
Div	1	Neg	2
Div_Un	1	Not	1
Rem	1	Shr	90
Rem_Un	1	Shr_un	2
		Shl	15

Table 2: Application requirement for load and store and comparison operations for Mp3.

OP—operation	m_{op}	OP—operation	m_{op}
Ldind_I1	1	LessThan	1
Ldind_I2	1	LessThan_Un	1
Ldind_I4	32	LessOrEqual	1
Ldind_U1	3	LessOrEqual_Un	
Ldind_U2	2	Equal	1
Ldind_U4	10	NotEqual	1
Stind_I1	1	GreaterOrEqual	1
Stind_I2	1	GreaterOrEqual_Un	1
Stind_I4	20	GreaterThan	1
		GreaterThan_Un	1

Table 3: Register file (RF) mapping function values and functional unit mapping.

RF configuration	Value	Functional unit	Value
RF 2×1	715	Alu	223
RF 3×1	713	Multiplier	73
RF 4×1	711	Divider unsigned	1
RF 4×2	710	Divider signed	1
RF 6×3	705	Comparator	1
RF 8×4	700		
RF 16×8	**668**		

Table 4: Parameter values and code size (LOC).

Benchmark	(P_l,P_f,P_{fl})[%]	LoC	Gen. T [sec]	
			Nonpipe	Pipe
bdist2	(60,50,45)	61	0.2	0.8
Sort	(80,60,45)	33	0.1	0.1
dct32	(18,65,50)	1006	1.3	2.3
Mp3	(30,55,50)	13898	15.6	42.6
inv/forw 4×4	(50,45,55)	95	0.2	0.8

are represented as independent variables which lead to high concurrency available for reading and writing variables. Since the variables are read from output port of register files and written to the input ports, these numbers are indicative of a desirable I/O configuration for the register file.

Finally, the available concurrency in data transfers is shown by the number of data transfers of source m_s and the data transfers of destination m_d operands. The values for Mp3 applications are 1360 and 742, respectively. These numbers correspond to the concurrency in transaction of source and destination operands that were defined earlier in the section. Again, we find that high concurrency in the operations translates into high values of m_s and m_d. This is to be expected since the number of source and destination operand transactions correspond directly to the number of operations in most cases. The only exception is the store operation that only has destination operand transactions. However, most other operations, particularly arithmetic and logic operations, typically have more source operand transactions than destination operand transactions.

2.2. Mapping Functions and Heuristics. For component allocation, we have derived several heuristic that measure how well the selected components match the given requirements. In order to allocate a register file, we have derived a function that measures how good any of the register files available in Component Library match the application requirement. The function used to evaluate the register files is given by the following formula:

$$H_{rf(x)} = 2 * \text{abs}(x\_in - rq\_in) + \text{abs}(x\_out - rq\_out), \quad (1)$$

where x is the candidate component form the library, and x_in and x_out are the number of input and output ports of the component x. Also rq_in and rq_out are the number of required inputs and outputs, respectively. Required number of outputs correspond to the number of source operands read using *Read Variable* operation, and number of inputs correspond to the number of *Write Variable* operations. The heuristic is chosen to give priority to the input port requirement in order to allow storage of as many results as possible. The value of the function is computed for each candidate register file component from the Component Libry, and the one with the smallest value of the function H_rq is chosen and allocated to the data path. Determining the register file size requires estimation of the maximum number of variables and temporary variables that may occur in the application. While counting local and global variables is straight forwarder task, estimating the temporaries require scheduling and binding to be done for the given data path. Therefore, the register file size determination is out of the scope of this work. For all the practical purposes, we use the rule of thumb: for the Initial Data path, we count the number of all the variables in *ALAP* code and allocate additional 25% registers in the register file.

Allocation of the source and the destination buses depends on the number of source operands and destination operands that the application may require at the same time (in the same cycle). Therefore, the number of source busses equals to m_s and the number of the destination busses equals to m_d that was recorded while traversing the application *ALAP* schedule.

As for the memory interface allocation, we first consider number of sources and destination buses and chose the maximum of them to serve as the required number of ports rq_mi. We compute the value of H_mi for each candidate x component according to

$$H\_mi(x) = x\_in - rq\_mi, \quad (2)$$

where x_in is the number of input ports of memory interface, and rq_mi is a required number of memory interface input ports. The component with the minimum value of the heuristic is selected. In the corner case, where rq_mi is less than any of x_in, the memory interface with the minimum (or maximum) number of ports is chosen. Similarly, in case where rq_mi is less greater than any of x_in, the memory interface with the maximum number of ports is chosen.

Furthermore, in order to select functional units for available operators, we define

(i) $Ops(FU)$ as a set of operations that a functional unit FU performs,

(ii) *Selected* set of selected functional units for the implementation of a final data path,

(iii) n_{FU} the number of instances of the functional unit FU in the *Selected*.

A matching heuristics $H(OP, Ops(FU)) \rightarrow Selected$ maps the set of operations OP to a set of functional units *Selected* to implement the custom data path. The heuristics H determines both the type and the number of instances of a functional unit. Algorithm 1 describes the heuristics used in this work. $|\text{Ops}(FU_i)|$ represents the cardinal number of set $\text{Ops}(FU_i)$. According to heuristics H, a functional unit FU_i will be selected to perform an operation op_i if it performs the greatest total number of operands alongside the chosen one. Therefore, this heuristics prioritizes functional units with a higher possibility for sharing. As for the number of instances, the heuristics includes additional units of a chosen type to the set *Selected* only if the maximum number of occurrences of operand op_i is greater than the number of units n_{FU} of that type currently in *Selected*, that is, if $m_{\text{op}} > n_{FUi}$. For example, if application requires 3 units to perform additions and 4 units to perform subtractions, and an ALU is chosen for both addition and subtraction operator, the tool will allocate only 4 instances of the ALU.

To ensure that the interconnect is not a bottleneck in the Initial Data path, the connection resources are allocated in greedy manner. This means that output ports of all register files are connected to all source buses. Similarly, input ports of all register files are connected to all destination busses. The same connection scheme applies to the functional units and the memory interface, making it possible to transfer any source operand to any functional unit or memory interface and the result back into a register file. Note that data forwarding is not presented in this paper and that it is a part of our current research.

```
for all op_i ∈ OP do
    Select FU_i such that
    op_i ∈ Ops(FU_i) &&
    |Ops(FU_i)| = max(|Ops(FU_k)|, ∀k such that op_i ∈
    Ops(FU_k))
     if m_op > n_FUi then
        Add [(m_op − n_FUi) ×FU_i] to Selected
    end if
end for
return Selected
```

Algorithm 1: H(OP,Ops(FU)).

2.2.1. Example. The heuristic described in this section was applied to compute the initial estimates of components and their configuration for our application examples. In particular, we computed H_rf values corresponding to each application for all the register files in the component database. We also computed the required number of source and destination buses for each design. The functional unit mapping heuristic was used to select the type and instances of functional units from the component database. The initial design decisions for the given applications are shown in this section.

Left-hand side of Table 3 shows the register file heuristic values (H_rf) for all possible I/O configurations available in the database. The configuration is indicated in the name of the register file type. For example, RF 4 × 2 refers to a register file with 4 outputs and 2 inputs. The minimum heuristic value for each application is highlighted in bold. In other words, the configuration that produces the lowest H_rf value is selected. Typically, for larger applications with higher available parallelism in computation, we find that the register file with most input and output ports is selected.

The desirable number of source and destination busses exactly matches the number of data transfers: it is 1360 for the source and 742 for the destination busses. However, it may be unreasonable to have a huge number of buses. In most of such cases, design decisions on the number of function units or the configuration of register files may limit the number of source or destination buses. We will discuss such structural dependencies and their impact on bus selection in the following section.

Memory selection and organization is an important issue in processor design. In general, it is possible to have architectures with multiple memories. However, the task of the memory manager becomes quite complicated in such scenario. For the target processor, the memory is not considered to be the part of the datapath itself. Instead, a fixed single-port memory interface is assumed to provide access to all addressable data. Therefore, in this work we have not focused on memory configuration. The available memory interface with one read and one write ports has been selected for all the applications.

The functional units selected for our example applications are listed in the right-hand side of Table 3. The selection is based on the heuristics described earlier. We have mapped all arithmetic and logic operations to ALUs. Multipliers and dividers are expensive components and have been assigned selectively. We also distinguish between signed divider and unsigned divider. All comparison operations have also been mapped to a common comparator. As in the case of buses, some desirable numbers for function unit allocation require very large area. However, as we will shortly see, the allocation is adjusted based on the structural dependencies between the components.

2.3. Structural Restrictions and Dependencies. In the previous sections, we discussed matching of C code structures to architectural components and templates. Since the components must be connected to each other, there exist several structural dependencies between them. As a result, the number of components may not be independently decided. There are several cases in which we must make trade-offs based on structural restrictions and dependencies. We will discuss some of these restrictions and dependencies in this section. Based on the structural restrictions, we will define an initial data path for the application-specific processor corresponding to each C example.

One of the primary bottlenecks is the number of register files and their configuration. If multiple or distributed register files were used, we would need to provide algorithms for mapping of both variables and temporaries to each register file or each partition. This is a significant problem because we must know the variables and all temporaries in the generated code and provide binding rules for them. Furthermore, we must have the scheduling information for the operations. Scheduling, binding, and mapping are complex problems that are out of the scope of this work. Indeed, one of the fundamental assumption of our processor design paradigm is the separation of allocation from scheduling and code generation. We make the trade-off in optimal concurrency and restrict ourselves to a single register file. However, concurrency is still available by configuring the number of input and output ports of the register file. Therefore, the restriction stems from our methodology and the computational complexity of data partitioning.

Another restriction that we imposed on the data path template is that it does not support forwarding. Therefore, the source operands may come only from register file output

ports and from the *constant* value stored in the microcode itself (i.e., in the control word). It must be noted that this restriction is only to manage the problem size and there does not exist any technical reason for restricting forwarding. Program analysis can give us information for deciding where to insert the forwarding paths. Also, in this work, only one *constant* value is assumed to be stored in any microcode for a single cycle (i.e., in one control word). Hence, the total maximum number of source buses equals the number of register file output ports (+1 in case the number of output ports is odd). In all of the cases of the benchmarks presented here, we have imposed an upper limit on the number of source buses as the number of register file output ports. By the same token, the number of destination buses is at most equal to the number of register file input ports.

We have assumed that the output of the function units is written directly to the register file. Therefore, for the practical purposes, it makes sense to limit the number of instances of any functional unit type to be not more than the number of register file input ports. This decision has been made because in case where all of the units of the same type are utilized at the same time, they need to be able to store their values after executing the operation. Due to the above reason, we also limit the total number of functional units to be not more than the total number of register file input ports. Therefore, we may need to adjust the number of functional unit with the highest number of instances, for a given application. For example, the $Mp3$ application requires 223 ALUs to exploit all the available parallelism. However, since there are total of four other types of functional units, we allocate only four ALUs.

We can conclude from the above arguments that the register file allocation imposes the most stringent constraints on the data path structure. Therefore, it makes sense to start with allocating the register file, followed by all other component allocations. Once all the components are allocated, we may readjust the number of instances of functional units, if needed.

The resulting initial data path for $Mp3$ consists of one register file with 16 outputs and 8 inputs (Rf 16×8), 4 ALUs, 2 Multipliers, 1 Comparator, 1 Divider Unsigned, and 1 Divider Signed. The number of source and destination buses is equal to the number of register file output and input ports, respectively. As it can be seen, because the applications has high level of parallelism, it utilizes the register files with the largest number of ports and the large number of instances of functional units.

3. Data Path Optimization

The initial data path is used for compiling and profiling the given application code. The compiled code is then analyzed: first, the basic blocks (BBs) to be optimized are selected (Section 3.1). The designer may chose from a single BB to all BBs in the code. Nevertheless, the entire application runs on the generated data path. Then, for each selected BB, a usage plot (usage per cycle) for each component is created (Section 3.2). For each specified resource constraint, the estimation algorithm computes the number of extra cycles that the application needs when the number of instances is as specified by the constraint. As multiple resource constraints may be specified, the smallest estimated number of extra cycles is selected to be the Timing Overhead (Section 3.3). A subset of possible designs is created and the number of execution cycles is estimated for each design [6]. The optimization goal is to determine the number of unconstrained components and the structure of the data path such that the execution is extended by no more than the Timing Overhead (Section 3.4).

3.1. Critical Code Extraction. We define critical code as the set of BBs in the source code that contributes the most to the execution time and have the largest potential for optimization. Our selection criterion is based on the relative size and execution frequencies of the BBs in the application. Large BBs have high potential for optimization, since they have several operations that may potentially be performed in parallel. On the other hand, even minor optimization of BBs that have high frequency will yield high overall performance gains. Therefore, we keep 3-ordered lists of BBs, sorted by length (l_i), frequency (f_i), and frequency-length product, where i is the block id. For user-defined parameters P_{fl}, P_l, and P_f, a BB is considered critical if it meets any of the below conditions:

$$f_i \cdot l_i \geq P_{fl} \cdot \sum_{j=0}^{N} f_j \cdot l_j, \tag{3}$$

$$l_i \geq P_l \cdot \max_{j=0}^{N} \left(l_j\right), \tag{4}$$

$$f_i \geq P_f \cdot \max_{j=0}^{N} \left(f_j\right). \tag{5}$$

If any of the parameters equals to 100%, the data path will be optimized for the entire code.

3.2. Usage Plot Creation. A usage plot shows cycle-by-cycle usage for each data path resource. One usage plot is created for each selected BB for each component type (in case of functional units and buses), and for data ports of the same type, that is, input or output ports (in case of the storage units). Usage plots are extracted from the schedule generated for the Initial Data path. The usage plot is annotated with the frequency of a corresponding BB. The example of a usage plot for adders is shown in Figure 5. If we assume that the type and number of instances of all other components do not change, we can conclude that we need 3 instances of adder to execute this BB in $\leq$6 cycles.

3.3. Timing Overhead Computation. Changing the number of instances of a resource may affect the application execution time. If the designer specifies resource constraint for adders to be two (Figure 5), the number of cycles required for execution of the BB would increase. There would be one

extra operation in cycle three and in cycle five that could not execute in the originally specified cycle. The estimation algorithm finds a consecutive cycle that has an available adder (such as cycle four and cycle six) and uses it for delayed execution of the extra operation. By modeling the delayed execution the tool estimates, the extra number of cycles, d_c. If this is the only resource constraint specified, *Timing Overhead* equals to d_c (one cycle in Figure 5). For all other unconstrained resources, the number of instances needs to be sufficient so that the given BB executes in maximum seven cycles.

The estimation is done for a single resource type at a time and, therefore, the input is a set of usage plots for that resource for selected basic blocks. The task of estimation step is to quantify the effect of change in the number of instances of the resource to the application execution time. In order to do so, we compute the number of extra cycles (d_c) that is required to execute a single basic block with the desired number of units (*NewNumber*) using "Spill" algorithm (Algorithm 2).

We keep a counter (*DemandCounter*) of operations/data transfers that were originally scheduled for the execution in the current cycle on an instance of the resource r but could not possibly be executed in that cycle with the *NewNumber* of instances. For example, in both cycles 3 and 5 (in bottom of the Figure 5), there is one operation (shown in dashed lines) that cannot be executed if only two adders are used. Those operations need to be accounted for by the *DemandCounter.*

In each cycle, we compare the number of instances in use in a current cycle ($X.InUse$) to the *NewNumber*. If the number in the current cycle is greater, the number of "extra" instances is added to the *DemandCounter*, counting the number of operations/transfers that would need to be executed later. On the other hand, if the number in the current cycle is less than the *NewNumber*, there are available resources that may execute the operations/transfers that were previously accounted for with *DemandCounter*. In the bottom of Figure 6, the available resources are shown in yellow and the "postponed" execution of "extra" operations is shown by arrows. The "Spill" algorithm models in this way the delayed execution of all "extra" operations. After going through all cycles in a given block, the *DemandCounter* equals to the number of operations that need to be executed during the additional cycles d_c.

The "Spill" algorithm uses only statically available information and provides the overhead for a single execution of a given basic block. In order to estimate the resulting performance, we incorporate execution frequencies in the estimation. The estimated total execution time equals sum of products of block's d_c and block's frequency for each block selected for the optimization. We must note that this method does not explicitly account for interference while changing the number of instances of other resources than the specified ones.

3.4. Balancing Instances of Unrestricted Components. We assume that it is acceptable to trade off certain percentage of the execution time in order to reduce number of used resources (hence to reduce area and power and increase component utilization). Therefore, we select a subset of all possible designs to be estimated. The designs in the selected subset are created as follows:

(i) set the number of instances for the constrained resources to their specified value,

(ii) set the number of instances of unconstrained resources (functional units, buses, storage elements, and their ports) to the same values as in the Initial Data path,

(iii) assume the same connectivity as in the Initial Data path,

(iv) randomly select a resource type and vary its number of instances.

For the example depicted in Figure 6, where a multiplier is selected to have its number of instances varied, there will be two candidate designs created: with one and with two multipliers. The candidate design with no multipliers would not make sense, since there will be no units to perform the operations that were originally performed by the multiplier (Algorithm 1). Also, there is no need to consider three multipliers, since two already satisfy the constraints. It may happen that even if we increase the number of instances of some component, the *Timing Overhead* cannot be met. The algorithm then backtracks to the minimal number that causes violation and reports the *"best effort"* design. In the simple case, shown in the Figure 6, if the Timing Overhead is 1 cycle, having only 1 unit results in the acceptable overhead.

4. Experimental Setup

We implemented the IDp Extraction and the Data path Optimization in C++. We used programmable controller unit, NISC compiler [2, 3] to generate schedule and Verilog generator [7, 8] for translating architecture description from ADL to Verilog. For synthesis and simulation of the designs, we used Xilinx ISE 8.1i and ModelSim SE 6.2 g running on a 3 GHz Intel Pentium 4 machine. The target implementation device was a Xilinx Virtex II FPGA xc2v2000 in FF896 package with speed grade −6. The benchmarks used were *bdist2* (from MPEG2 encoder), *Sort* (implementing bubble sort), *dct32* (from MP3 decoder), *Mp3* (decoder), and *inv/forw* 4×4 (functions *inverse4* $\times$ *4* and *forward4* $\times$ *4* are amongst top five most frequently executed functions from H.264). The functions and programs vary in size from 100 lines of C code to over 10000 lines (Table 4). Function *bdist2* is part of the MPEG2 encoder algorithm. *Sort* is a bubble sort implementation that is widely used in various applications. Function *dct32* is a fixed-point implementation of a discrete cosine transform. The *Mp3* decoder is the example for decoding mp3 frames into pulse-code modulated (PCM) data for speaker output. It also contains the *dct32* as one of the functions. Finally, *inv/forw* 4×4 is a forward and inverse integer transform that is used in H.264 decoder. Profiling information was obtained manually.

Table 4 shows input parameters, benchmark length, and generation time. Parameters P_l, P_f, P_{fl} are defined in

```
in: Usage Plot(UP) for a Resource r
in: Number of Instances: NewNum
out: d_c//in number of cycles
for all X = cycle ∈ UP do
   CycleBudget = NewNumber − X.InUse;
   if CycleBudget ≥ 0&&DemandCounter ≥ 0 then
      CanFit = min(CycleBudget, DemandCounter)
      DemandCounter+ = CanFit
   else
      DemandCounter+ = CycleBudget
   end if
end for
d_c = ⌈DemandCounter/NewNumber⌉
return d_c
```

Algorithm 2: Spill.

Section 3.1. We decided on parameter values using the profiling information. The selected values of parameters ensure that blocks that affect the execution time the most are selected for optimization. The following column shows the number of lines of the C code (LoC). The largest C code has 13,898 lines of code, proving the ability of the proposed approach to handle large industrial scale applications. The last two columns present average generation time for nonpipelined and pipelined designs. Even for industrial size application, generation time is less than one minute.

In this paper, we present three sets of experiments. The first set of experiments illustrates design space exploration using the automatic extraction of data path from application C code. The second set of experiments compares our selection algorithm to manual selection of components from C code. The last set of experiments compare the presented extraction technique to HLS and manual design in order to establish quality of generated designs.

5. Results: Interactive Design Exploration

Results of the exploration experiments are shown in Figures 8, 9, 10, 11, 12, and 13. Used baseline data path architectures are MIPS-style manual designs (pipelined and nonpipelined) [5] with an ALU, a multiplier, two source and one destination bus, and a 128-entry register file with one input and two output ports. Only for the *Mp3* application, we have added a divider unit to this processor for comparison with the generated data path. In order to perform fair comparison, the size of storage elements has been customized for every application such that the resources (area) are minimized. Also, for comparison with automatically generated pipelined design, the *pipelined version* of manual design was used as a baseline. In-house compiler is used to create schedule for all baseline and generated data paths. This guarantees that the execution time depends on the data path architecture and does not depend on the compiler optimizations.

While exploring different designs for selected applications, we specified the resource constraints on the number of ALUs and number of output and input ports of register file (RF). The tool extracts a data path from the C code such that it complies to the specified constraint, and resulting data paths are named as

(i) *ALU-N*, where $N \in \{1, 2, 3\}$ is the specified number of ALUs,

(ii) *RFOxI*, where $(O, I) \in \{(2, 1), (4, 2), (6, 3), (8, 4)\}$ are the number of output and input ports.

In case of data path denoted by *RFOxI*, *two* resource constraints were used to generate the design: one for the number of output and the other for the number of input ports while all the remaining elements (like functional units, memories, and connectivity) are decided by the tool as described in Section 3.

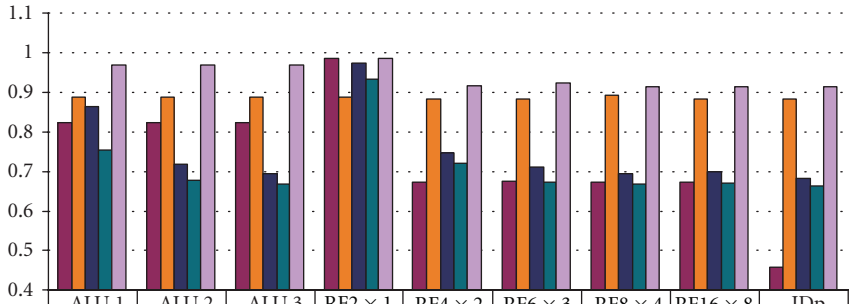
Figures 8 and 9 show the number of execution cycles for generated architectures normalized to the number of cycles for the baseline architecture. These graphs include two additional data paths that are generated only to illustrate tool behavior and to explore theoretical limits of the used data path model (Figure 7). Those additional configurations are

(i) *RF 16 × 8*: a configuration that was generated using RF 16 × 8 constraint,

(ii) *IDp*: an Initial Data path.

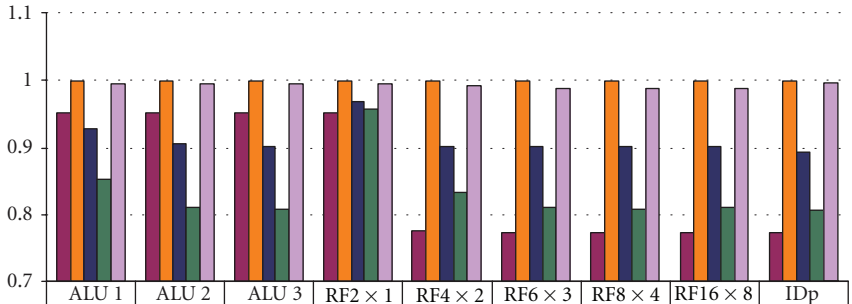
Table 5 summarizes generated architectures for all the configurations that are presented in this paper. Each benchmark and each configuration have a nonpipelined and a pipelined architecture generated, and those are presented in rows marked with N and Y in the column "Pipe." The table lists the difference from the corresponding baseline architecture. For example, "#R = 64" means that the generated data path has 64 registers in register file, "Rf 4 × 2" means that there is a register file with 4 output and 2 input ports, and "—" means that there is no change to the baseline data path parameters.

For generated nonpipelined data paths (Figure 8), normalized execution cycles range from 0.98 to 0.46. All of the benchmarks experience only a small improvement for RF2 × 1 configuration because this configuration is the most similar



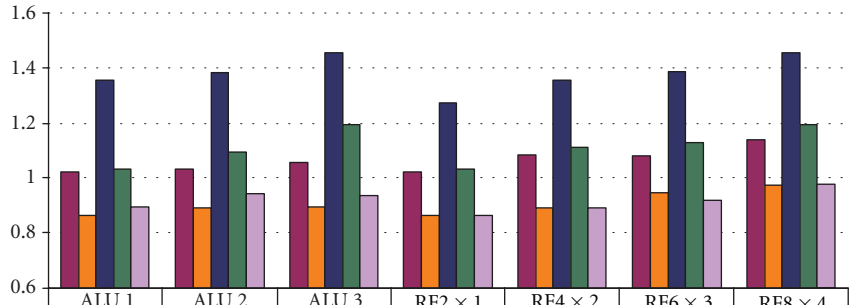
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4	RF16 × 8	IDp
bdist2	0.82	0.82	0.82	0.98	0.67	0.68	0.67	0.67	0.46
Sort	0.89	0.89	0.89	0.89	0.88	0.88	0.89	0.88	0.88
dct32	0.86	0.72	0.7	0.97	0.75	0.71	0.7	0.7	0.68
Mp3	0.76	0.68	0.67	0.93	0.72	0.67	0.67	0.67	0.67
inv/forw 4 × 4	0.97	0.97	0.97	0.98	0.92	0.92	0.92	0.92	0.92

Figure 8: Relative number of execution cycles on *nonpipelined* data paths generated for different resource constraints.



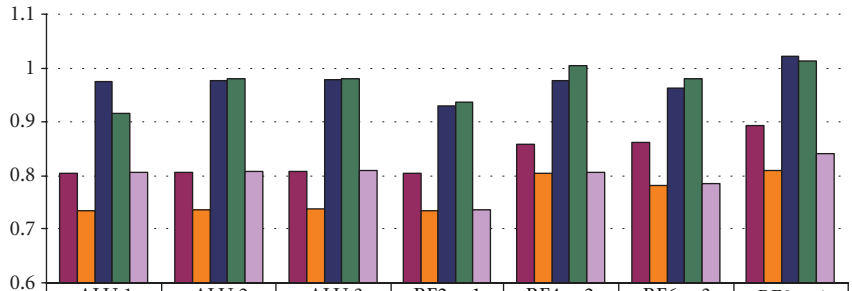
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4	RF16 × 8	IDp
bdist2	0.95	0.95	0.95	0.95	0.77	0.77	0.77	0.77	0.77
Sort	1	1	1	1	1	1	1	1	1
dct32	0.93	0.91	0.9	0.97	0.9	0.9	0.9	0.9	0.89
Mp3	0.85	0.81	0.81	0.96	0.83	0.81	0.81	0.81	0.81
inv/forw 4 × 4	1	1	1	0.99	0.99	0.99	0.99	0.99	1

Figure 9: Relative number of execution cycles on *pipelined* data paths generated for different resource constraints.



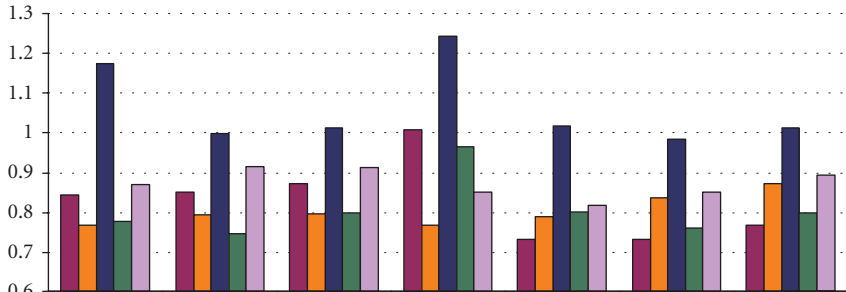
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
bdist2	1.03	1.03	1.06	1.03	1.09	1.08	1.14
Sort	0.87	0.9	0.9	0.87	0.89	0.95	0.98
dct32	1.36	1.38	1.46	1.28	1.36	1.39	1.46
Mp3	1.03	1.1	1.2	1.03	1.11	1.13	1.2
inv/forw 4 × 4	0.9	0.94	0.94	0.87	0.89	0.92	0.98

Figure 10: Relative cycle time for *nonpipelined* data paths generated for different resource constraints.



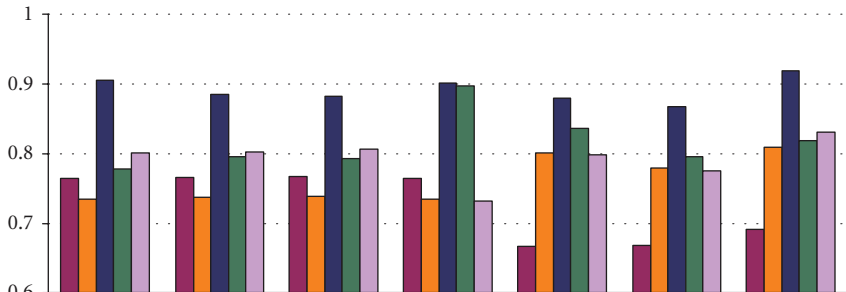
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
bdist2	0.8	0.81	0.81	0.8	0.86	0.86	0.89
Sort	0.74	0.74	0.74	0.74	0.8	0.78	0.81
dct32	0.98	0.98	0.98	0.93	0.98	0.96	1.02
Mp3	0.91	0.98	0.98	0.94	1	0.98	1.01
inv/forw 4 × 4	0.81	0.81	0.81	0.74	0.81	0.79	0.84

Figure 11: Relative cycle time for *pipelined* data paths generated for different resource constraints.



	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
bdist2	0.85	0.85	0.87	1.01	0.73	0.73	0.77
Sort	0.77	0.79	0.8	0.77	0.79	0.84	0.87
dct32	1.17	1	1.01	1.24	1.02	0.98	1.01
Mp3	0.78	0.74	0.8	0.97	0.8	0.76	0.8
inv/forw 4 × 4	0.87	0.91	0.91	0.85	0.82	0.85	0.89

Figure 12: Relative execution time for *nonpipelined* data paths generated for different resource constraints.



	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
bdist2	0.76	0.77	0.77	0.76	0.66	0.67	0.69
Sort	0.73	0.74	0.74	0.73	0.8	0.78	0.81
dct32	0.9	0.88	0.88	0.9	0.88	0.87	0.92
Mp3	0.78	0.79	0.79	0.9	0.84	0.79	0.82
inv/forw 4 × 4	0.8	0.8	0.81	0.73	0.8	0.78	0.83

Figure 13: Relative execution time for *Pipelined* data paths generated for different resource constraints.

Table 5: Difference in components and parameters between respective baseline and generated design.

Benchmark	Pipe	ALU1	ALU2	ALU3	RF 2 × 1	RF 4 × 2	RF 6 × 3	RF 8 × 4	RF 16 × 8	IDp
bdist2	N	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	Rf 8 × 4, 3 Alu, 2 Mul
	Y	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	Rf 8 × 4, 3 Alu, 2 Mul
Sort	N	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 6 × 3, 1 Alu
	Y	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 6 × 3, 1 Alu
dct32	N	Rf 4 × 2	Rf 6 × 3	Rf 8 × 4	—	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
	Y	Rf 4 × 2	Rf 4 × 2	Rf 6 × 3	—	2 Alu	2 Alu	2 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
Mp3	N	—	Rf 6 × 3	Rf 8 × 4	—	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
	Y	Rf 4 × 2	Rf 6 × 3	Rf 6 × 3	—	2 Alu	2 Alu	2 Alu	2 Alu	Rf 16 × 8, 4 Alu, 2 Mul
inv/forw 4 × 4	N	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 16 × 8, 4 Alu, 1 Mul
	Y	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 16 × 8, 4 Alu, 1 Mul

to the baseline. Also, the number of cycles is not exactly the same but slightly reduced. This effect is due to replacing of buses in architecture specification with multiplexers which allows for more efficient handling by the compiler. This effect is particularly emphasized in case of *bdist2*. For this benchmark, the improvements are small (0.82) for all ALU configurations and may be attributed to the effect of explicit multiplexer specification that results in more efficient compiler handling and shorter prologue/epilogue code. We see the further reduction to 0.67 for RF 4 × 2 configuration which exploits the parallelism of operations executed on different units. No further improvement is seen for further increase in the number of register file ports. However, execution on the IDp, which has two ALU and two multiplier units, experiences additional improvement in number of cycles to 0.47.

Benchmark *Sort* is sequential in nature and, therefore, it does not experience significant improvement regardless of the number of ALUs or register file ports that are introduced. Both benchmarks *dct32* and *Mp3* have abundance of available parallelism. The *dct32* benefits the most from having two ALUs (ALU2-0.72), and the increase in number of ports or adding more units (in IDp) contributes by only 4% of additional improvement. Similarly, *Mp3* executes in 0.68 of the number of baseline execution cycles for ALU2 configuration. Note that specifying two ALUs as resource constraint for both benchmarks results in an increase in the number of RF ports and buses: since both instances of ALU and one instance of multiplier are significantly utilized, the resulting configurations have RF 6 × 3 and full connectivity scheme. On the other hand, both benchmarks suffer from significant increase of prologue/epilogue code which sets back the savings in number of cycles that are obtained by the "body" of the benchmark. Adding more ALUs does not help in case of benchmark *inv/forw 4 × 4*. The benchmark benefits the most from additional register file ports, because this configuration exposes limited parallelism between the operations that execute on functional units of different type.

As for the pipelined configurations, shown in Figure 9, across all the benchmarks, maximum reduction in the number of execution cycles for generated data paths (0.77) is less than a maximum reduction for the nonpipelined designs since the pipelining itself exploits some degree of available parallelism. In case of *bdist2,* there is no improvement with increased number of ALUs since the tool allocates single RF 2 × 1. Same as for the nonpipelined configurations, the minimal normalized number of cycles is reached for RF 4 × 2 due to the increased simultaneous use of ALU and multiplier. On the other hand, benchmark *Sort* does not change the number of execution cycles since pipelining takes advantage of already limited available parallelism.

For configurations ALU 1, ALU 2, and ALU 3, the tool allocates, together with sufficient connectivity resources:

(i) for *dct32*: RF 4 × 2, RF 4 × 2 and RF 6 × 3, respectively;

(ii) for *Mp3*: RF 4 × 2, RF 6 × 3 and RF 6 × 3, respectively.

The most of the execution cycle reduction is brought by an increase in the number of register file ports in case of configuration ALU1. Increasing both number of ALUs and ports brings down the normalized cycles only by 0.02 and 0.04 down to 0.90 and 0.81 for *dct32* and *Mp3*, respectively. For all the RF configurations, both benchmarks have the same trend for allocation: the tool recognizes a potential for adding more ALUs and, therefore, two ALUs are allocated for all of them, except for RF 16 × 8 configuration of *dct32* where three ALUs are allocated. One would expect the tool would allocate more units for RF 8 × 4 and RF 16 × 8. However, the data dependencies limit concurrent usage of more units than allocated. The results for *IDp* illustrate this: even though *IDp* has three ALUs and two multipliers, further reduction in the number of normalized cycles is only by 0.01 to 0.02 for *dct32* and *Mp3*, respectively. Similarly to *Sort*, *inv/forw 4 × 4* has almost no improvement if the number of instances of increases.

Figures 10 and 11 show normalized cycle time for nonpipelined and pipelined automatically generated designs. We observed the cycle time in order to explain the total execution time of each benchmark. Current version of the tool does not take into account postsynthesis results. However, we believe that this feature is crucial for DFM and are currently working on incorporating prelayout information in data path optimization.

Results for normalized cycle time for designs are intuitive: as complexity of generated data path increases, so does

the cycle time. For nonpipelined designs (Figure 10), designs for all benchmarks except *Sort* have larger cycle time than for corresponding baseline. The main contributor to cycle time length is register file: as the number of ports increase, the decoding logic increases and so does the cycle time. In case of *Sort,* cycle time is lower because of the reduction of the register file size. For nonpipelined configurations, the normalized cycle time ranges from 0.85 to 1.46. Pipelined configurations (Figure 11) uniformly have smaller or equal cycle time as the baseline configuration. For each benchmark, there is almost no difference in cycle time across all ALU configurations. *Mp3* is the only benchmark that has significantly lower normalized cycle time for ALU1 configuration (0.91) than for the remaining two ALU configurations (0.98). RF configurations experience the increase in cycle time with an increase in complexity. For RF 6×3 in case of *Sort*, *dct32*, *Mp3,* and *inv/forw* 4×4, there is a small decrease in cycle time comparing to RF 4×2 configurations because the synthesis too manages to decrease combinational delay of interconnect.

Figures 12 and 13 show normalized total execution time. Across all configurations and all benchmarks, except all nonpipelined configurations for *dct32*, total execution time has been reduced. Nonpipelined *dct32* experiences increase in execution time for all but ALU2 configuration: the reduction in number of cycles is not sufficient to offset the large increase in the cycle time. The reduction in number of cycles is less than expected because of explosion of prologue/epilogue code. The nonpipelined configurations reduce the execution time up to 0.73, 0.77, 1.00, 0.74, and 0.82 for *bdist2*, *Sort*, *dct32*, *Mp3,* and *inv/forw* 4×4, respectively. Normalized execution times for all nonpipelined configurations, except for the *Sort*, are greater than the corresponding normalized number of cycles. The *Sort* has further decrease in execution time due to significant cycle time reduction (resulting from "minimized" data path comparing to the baseline). Furthermore, for *dct32* and *Mp3*, that perform the best for ALU2, several other configurations have minimum normalized number of cycles. Pipelined configurations uniformly experience smaller normalized execution time compared to the nonpipelined. The minimums are 0.66, 0.73, 0.88, 0.79, and 0.73 for *bdist2*, *Sort*, *dct32*, *Mp3,* and *inv/forw* 4×4, respectively. For all applications, each normalized execution time is smaller than the corresponding normalized number of execution cycles. Furthermore, the configurations that perform in minimal time are the same as the one that performs in minimal number of cycles.

In order to find a data path with a minimum execution time and the best configuration, we plot for each benchmark absolute execution time in Figure 14. The leftmost bar shows the execution time on a baseline architecture. The best implementation for *bdist2* is pipelined RF 4×2. RF 6×3 has only slightly longer execution time, but since it uses more resources, it is a less desirable (recommendable) choice. Benchmark *Sort* benefits from reduction of resources and, therefore, the best configuration is ALU1. For this benchmark, all of the pipelined configurations perform worse than corresponding nonpipelined. Benchmark *dct32*, despite having plethora of available parallelism, performs good only for nonpipelined *Baseline*, ALU2, and RF 6×3 configurations. The pipelined configurations do not perform as well as nonpipelined. To improve the current generated pipelined architectures, we may consider use of multicycle and pipelined functional units which may reduce the cycle time. Furthermore, if there is only single function to be performed on the generated hardware module, both prologue and epilogue code may be eliminated and the speedup of "parallel" architectures would increase. Here, we presented the results for all the applications with prologue/epilogue code because we believe that the application execution needs to have data received and sent proceeding and following the execution of the benchmark body. Therefore, benchmark is a function that needs to be called and, therefore, the prologue and epilogue codes are required. In this case, the number of registers that need to be stored and restored, and hence the length of prologue and epilogue code, needs to be estimated. Nonpipelined designs for *Mp3* perform better than pipelined, for the same reason. Overall, the best design would be for ALU2 configuration with 32% performance improvement over the baseline. Similarly, benchmark *inv/forw* 4×4 has smaller execution time for all nonpipelined configurations than for the pipelined ones. The peak performance improvement, 37%, is achieved for RF 4×2 configuration.

6. Results: Selection Algorithm Quality

Table 6 shows the comparison of manually designed architectures and the automatically generated ones. The manual designs were created by computer engineering graduate students. The students were asked to select the components for the templatized data path, as the one in Figure 7, based on the application C code. Running and profiling the code on the host machine with the same input as used for the automatic generation data were allowed.

There is the only one column for manual designs in Table 6 because the designers had *the same* component/parameter selection for nonpipelined and for pipelined data paths. However, our experiments in Section 5 show that often there is less resources required for pipelined configurations. Such examples are configurations ALU2, ALU3, RF 6×3, and RF 8×4 for *dct32* in Table 5. The nonpipelined and pipelined configurations presented in Table 6 are those that have the smallest number of execution cycles for the given benchmark, as seen in Figures 8 and 9.

It is interesting to notice that for manual designs, in most cases, the number of instances and parameters of selected register files and functional units outnumbers the one in the best generated architectures. For example, for benchmark *bdist2,* manual designer anticipated use of four ALUs. The nonpipelined IDp for benchmark *bdist2* needs only three and pipelined IDp only one ALU, which shows that the designer overestimates the number of ALUs. Also, the optimal automatically generated data path uses only one ALU in both nonpipelined and pipelined cases. However, for this benchmark, the designer underestimated register file

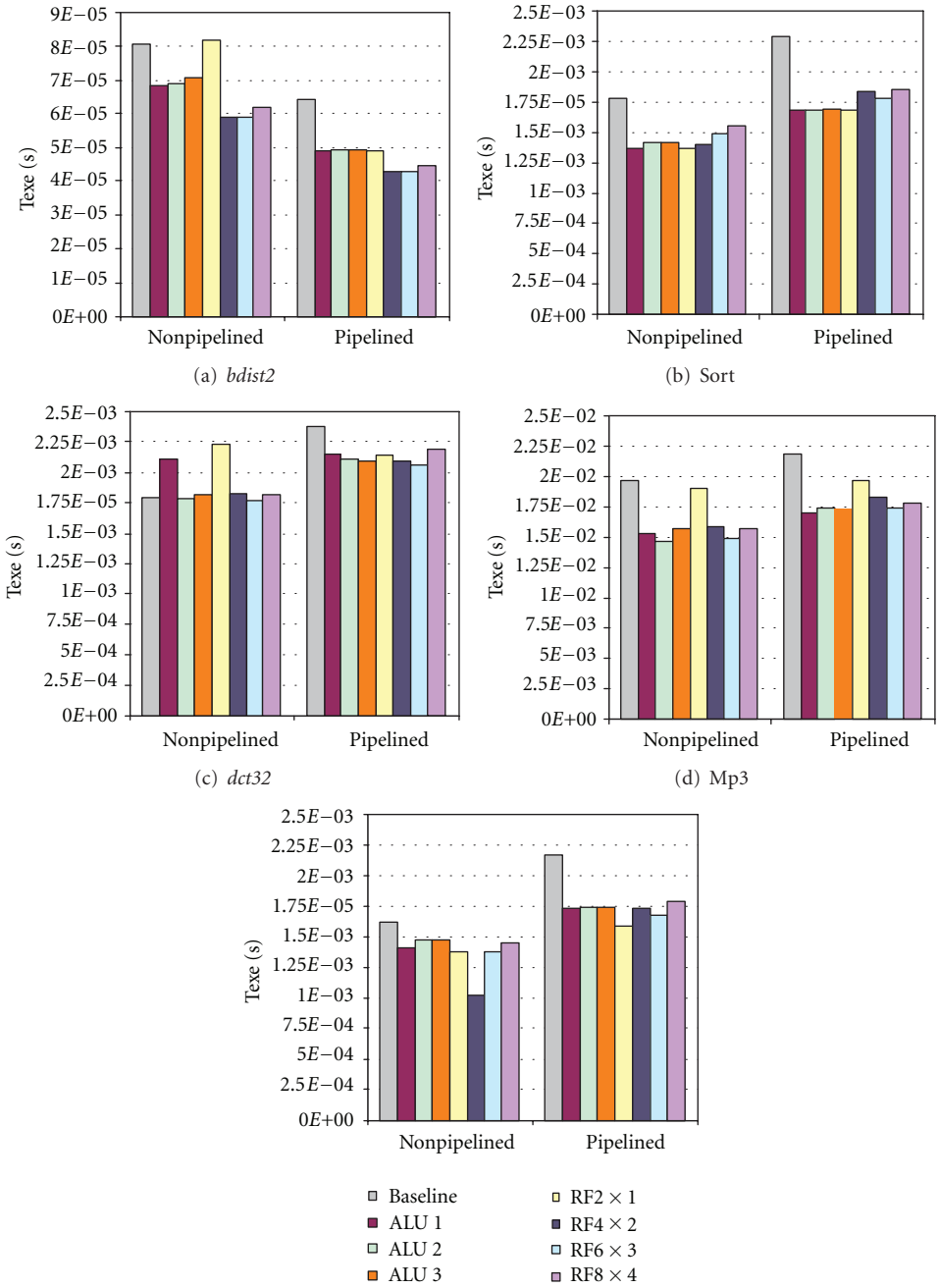


FIGURE 14: Total execution time on generated data paths.

size: in case where there are more functional units, more operations may be performed in parallel and, therefore, there will be more operands/registers required. The tendency to allocate manually more resources than actually required may be explain the best on the example of function *inverse4* × *4* from *H.264*, shown in Algorithm 3.

The designer allocates four ALUs based on an observation that code in lines 7, 8, 9, and 10 is not dependent and an assumption that once all of the operations from line 2 to line 5 are completed, the entire block of lines 7 to 10 will be executed at the same time. However, code in lines 2 to 5 has data dependencies, requires sequential execution, and performs memory access. Therefore, it makes sense to compute expressions in line 7 and line 8 as soon as t0 and t2 are available. Hence, no need for 4 ALU in the data path.

Similarly, for *dct32*, data dependencies are not "visible" from C code. Therefore, the designer allocates four ALUs, two multipliers, a comparator, and two adders. IDp for *dct32* has only three ALUs, two multipliers, and a comparator even though it has a register file RF 16 × 8. IDp configuration

Table 6: Comparison between components and parameters of manual and automatically generated design.

Benchmark	Manual	Automatic	
		Nonpipe	Pipe
bdist2	#R = 32, Rf 8 × 4, 4 Alu, 1 Mul	#R = 64, Rf 4 × 2, 1 Alu, 1 Mul, 1 Comp	#R = 32, Rf 4 × 2, 1 Alu, 1 Mul, 1 Comp
Sort	#R = 32, Rf 4 × 2, 1 Alu	#R = 32, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp	#R = 32, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp
dct32	#R = 48, Rf 8 × 4, 4 Alu, 2 Mul, 1 Comp, 2 Adders	#R = 128, Rf 8 × 4, 3 Alu, 1 Mul, 1 Comp	#R = 128, Rf 4 × 2, 2 Alu, 1 Mul, 1 Comp
Mp3	#R > 16, Rf 16 × 8, 4 Alu, 8 Mul 1 Or, 1 Comp, 1 NotEq Comp, 1 Div	#R = 128, Rf 8 × 4, 3 Alu, 1 Mul, 1 Comp, 1 Div	#R = 128, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp, 1 Div
inv/forw 4 × 4	#R = 32, Rf 8 × 4, 4 Alu, 1 Comp	#R = 16, Rf 4 × 2, 1 Alu, 1, Mul, 1 Comp	#R = 16, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp

```
1:  ...
2:  t0 = *(pblock++);
3:  t1 = *(pblock++);
4:  t2 = *(pblock++);
5:  t3 = *(pblock);
6:
7:  p0 = t0 + t2;
8:  p1 = t0 − t2;
9:  p2 = SHIFT(t1, 1) − t3;
10: p3 = t1 + SHIFT(t3, 1);
11:  ...
```

Algorithm 3: Part of *inverse4* × *4* C code.

performs in 0.68 of the baseline, which is only 2% better than configurations *ALU3*, RF 8 × 4, and RF 16 × 8 that have less resources. The number of registers in the register file is computed based on the number of units (eight without the comparator), the fact that each unit has two inputs and one output, and assumption that for each source/destination the data memory will be used twice. Therefore,

$$\#R = 8 \times (2 + 1) \times 2 = 48, \tag{6}$$

that is, the designer decides on 48 registers. Practically, with these many units, there are more than 48 registers required for temporary variables, if we want to avoid access to memory for fetching data and storing results.

Manual selection of components and parameters for *Mp3* shows the same properties: the number of functional units was overestimated, the number of registers in the register file was underestimated, and the pipelining was selected after the decision on units had been made. The designer profiled the application and found that among all computationally intensive functions, function *synth_full* contributes 35% to total execution. The designer identified eight multiplications and four additions that may be executed in parallel in this function. Also, only the lower bound for the number of registers in the register file was specified.

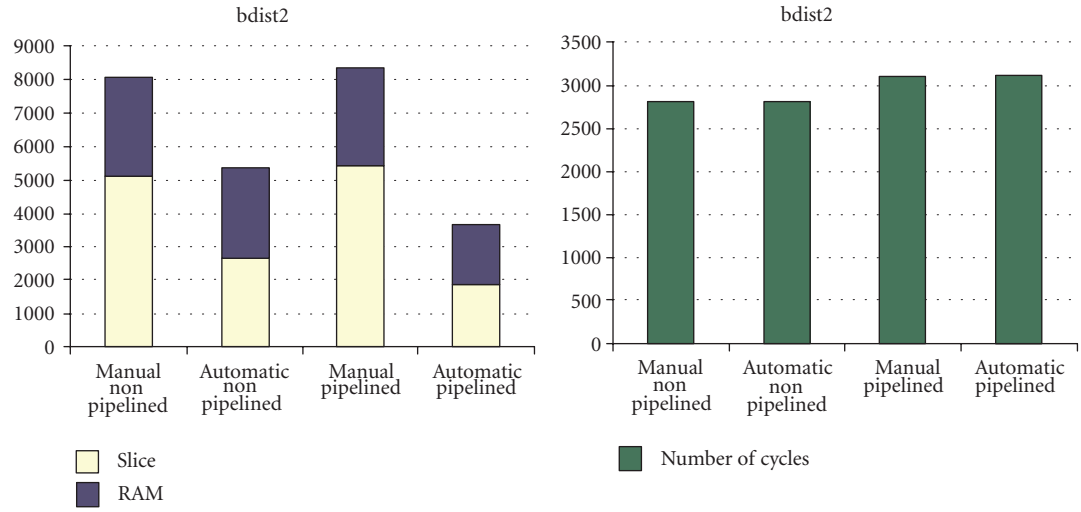
In order to better understand cost/performance trade-off for manually and automatically generated data paths, we defined a total cost of a design C_{design} as a sum of slices and a sum of RAMs for all the selected components:

$$C_{\text{design}} = \sum_{\text{components}} \text{slice} + \sum_{\text{components}} \text{RAM}. \tag{7}$$

We synthesized all available components and assigned cost in terms of slices and RAMs. We generated both nonpipelined and pipelined versions of manual design, so that we can perform fair comparison of cost and performance. We assumed that when pipelining was selected all inputs and outputs of functional units have a pipeline register (uniform pipelining, such as shown in Figure 7). Cost of pipeline registers was added to the total cost of pipelined data path designs. The performance is measured in the number of execution cycles, since neither the designers nor the tool were given any synthesis information as an input.

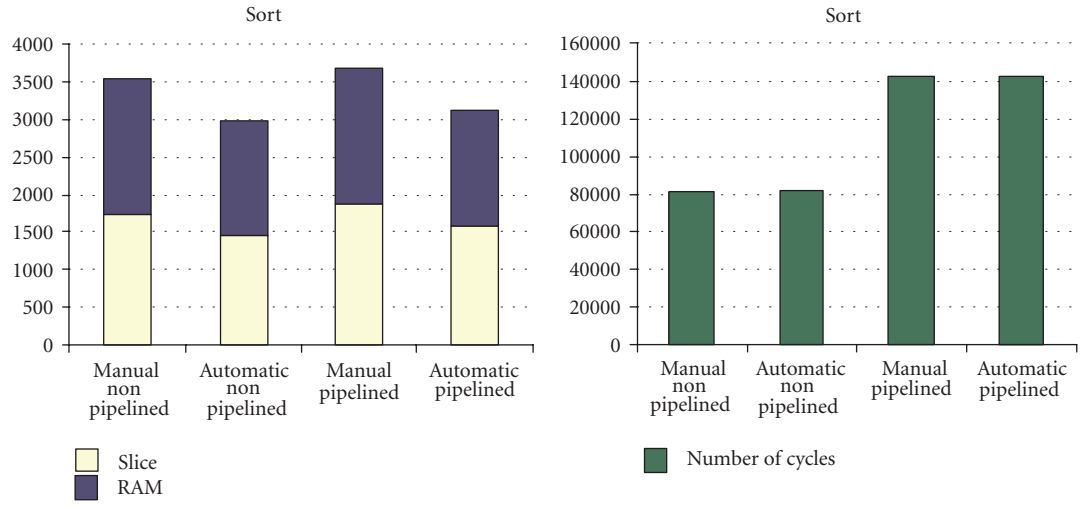
Figures 15, 16, 17, 18, and 19 show set of cost/performance graphs for all presented benchmarks. All automatically generated designs have significantly lower cost compared to the ones manually derived from C code. Relative to the corresponding manual designs, the automatically generated ones have from 7.41% (*dct32* nonpipelined) to 82.13% (pipelined *inv/forw 4 × 4*) less slices and from 8.22% (nonpipelined *bdist2*) to 87.46% (pipelined *Mp3*) less RAMs. As seen in the performance graphs, overhead in the number of cycles is negligible for all designs except for the above-mentioned pipelined *Mp3*. *Mp3* has 18.84% overhead, but 81.50% and 87.46% less slices and RAMs, which according to us is a reasonable trade-off. For all the remaining designs, overhead of the number of cycles ranges from 0.00% to 0.68% relative to the corresponding manual design.

This experiment showed that translating C code into simple hardware design is a nontrivial process. Selecting components and their parameters, tracking data and control dependencies, and estimating operation sequencing and available parallelism based on high level description results in underutilization of data path elements. On the other hand, using our tool, within the same time, a designer may explore many different alternatives and create working solution that satisfies his or her needs.



(a) Cost (b) Performance

Figure 15: *bdist2*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.



(a) Cost (b) Performance

Figure 16: Sort: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

7. Results: Design Refinement Quality

In this section, we present design refinement quality for two applications *dct32* and *Mp3* and compare automatically generated data paths to implementations using HLS tool and MicroBlaze soft processor.

7.1. Dct32. Figure 20 plots values for the number of cycles (No.cycle), clock cycle time (Tclk), and total execution time (Texe), while Figure 21 plots values for number of slices and bRAMs (Slice and BRAM) for several different data paths for *dct32* benchmark. All the values have been normalized to the corresponding values of a *manually* designed data path for the same application. Note that the same C code has been used as a starting point for all designs, including manual. The graphs show following data paths:

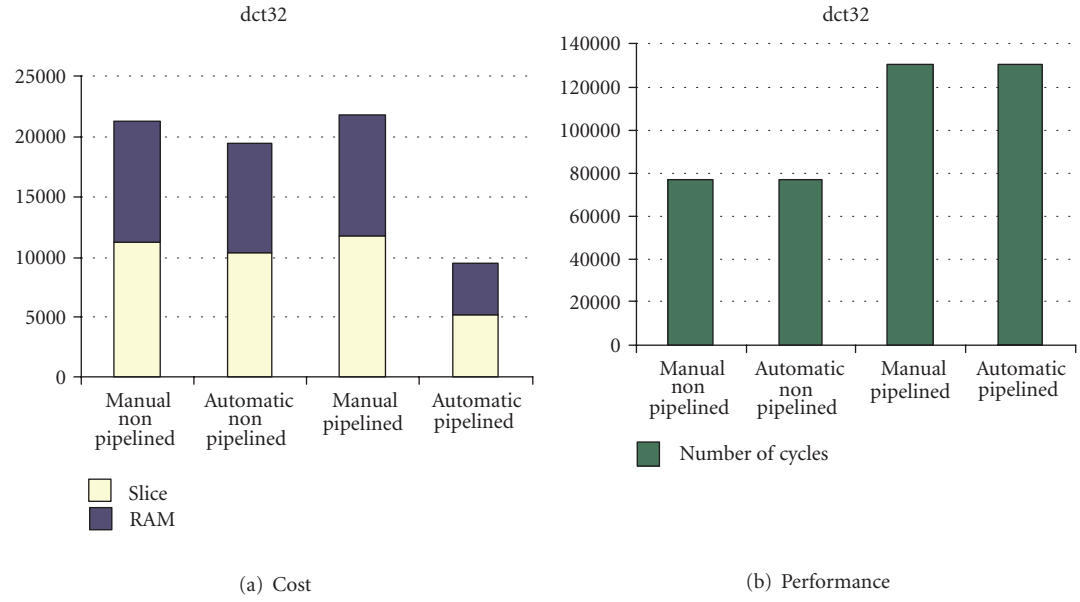
(i) *Baseline*—corresponds to a pipelined version of a baseline design for the *dct32* used in 5,

(ii) *ALU1-N and RF 4 × 2-N*—generated *nonpipelined* data paths for constraints ALU 1 and RF 4 × 2, respectively,

(iii) *RF 4 × 2-P*—generated *pipelined* data path for constraint RF 4 × 2,

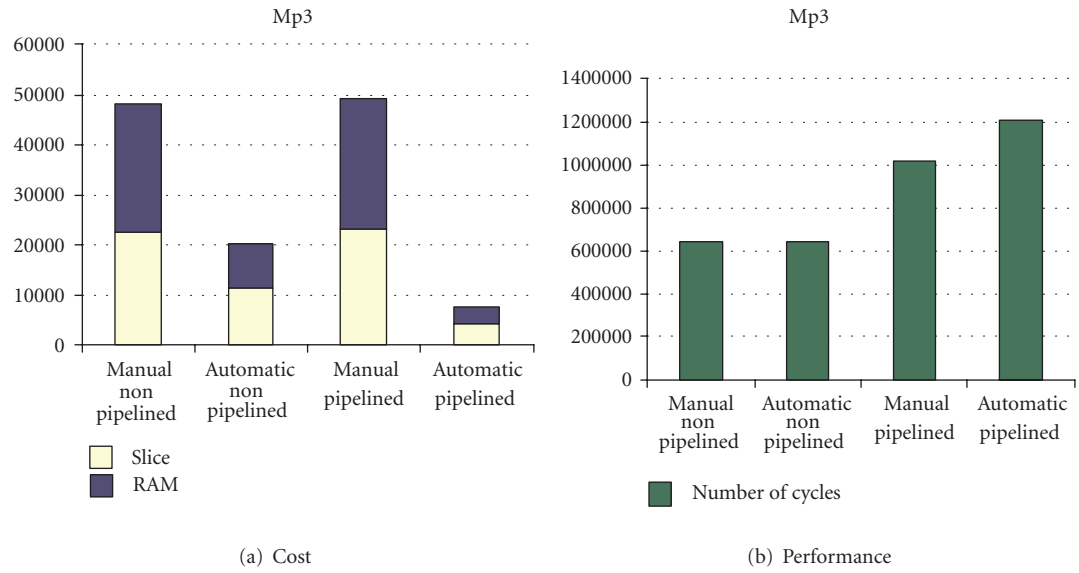
(iv) *HLS*—a design generated by academic high level synthesis tool [9],

(v) *MicroBlaze*—a design implemented on soft processor MicroBlaze [10].



(a) Cost (b) Performance

Figure 17: *dct32*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

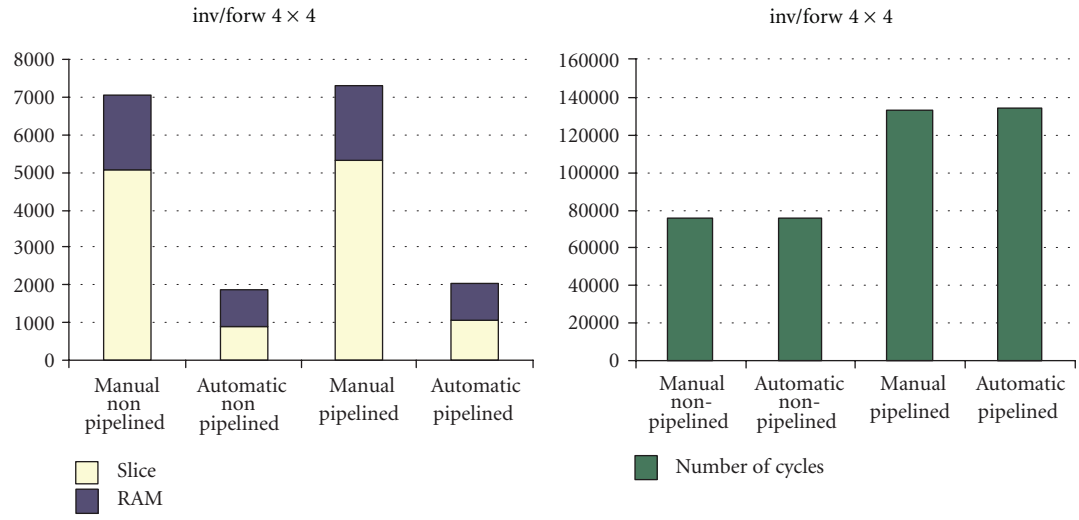


(a) Cost (b) Performance

Figure 18: Mp3: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

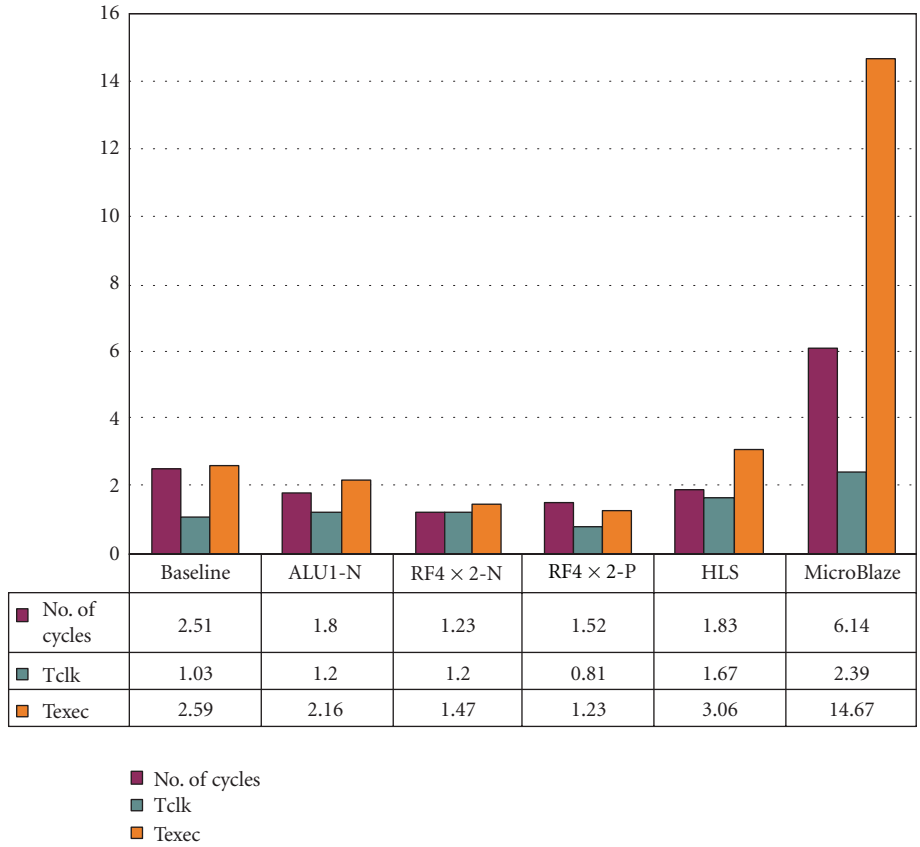
To alleviate different assumptions of different tools and designers for wrapping the function by send/receive primitives, we present here the results for the body of the *dct32* function, contrary to experiments in 5. The manual implementation has been designed by third-party RTL designer [11]. It is important to notice that the largest normalized value across all performance metrics for the *Baseline* and all the generated designs is 2.59 times the corresponding metric of the manual design, where HLS and MicroBlaze reach 3.06 and 14.67, respectively. Hence, for the generated designs, none of the compared metrics are several orders of magnitude larger than the manual design. The overhead of number of cycles for generated designs range from 23% (i.e 1.23 on the graph) to 80% of the manual design, while cycle time experiences from 25% (0.85 in the figure) speedup to 25% (1.25 in the figure) slowdown. The best generated design RF 4×2-P has 1.23 times longer execution time comparing to the manual.

Baseline and all generated architectures have from 0.53 to 0.64 times slices and 2.29 times block RAMs (bRAMs) compared to the manual design. This is because the tool attempts to map all storage elements to bRAM on FPGA. On the other hand, the design generated by HLS tool uses 1.36 times slices and only 0.14 times bRAMs due to the heavy use of registers and multiplexers. The generated designs outperform the design produced by HLS tool with



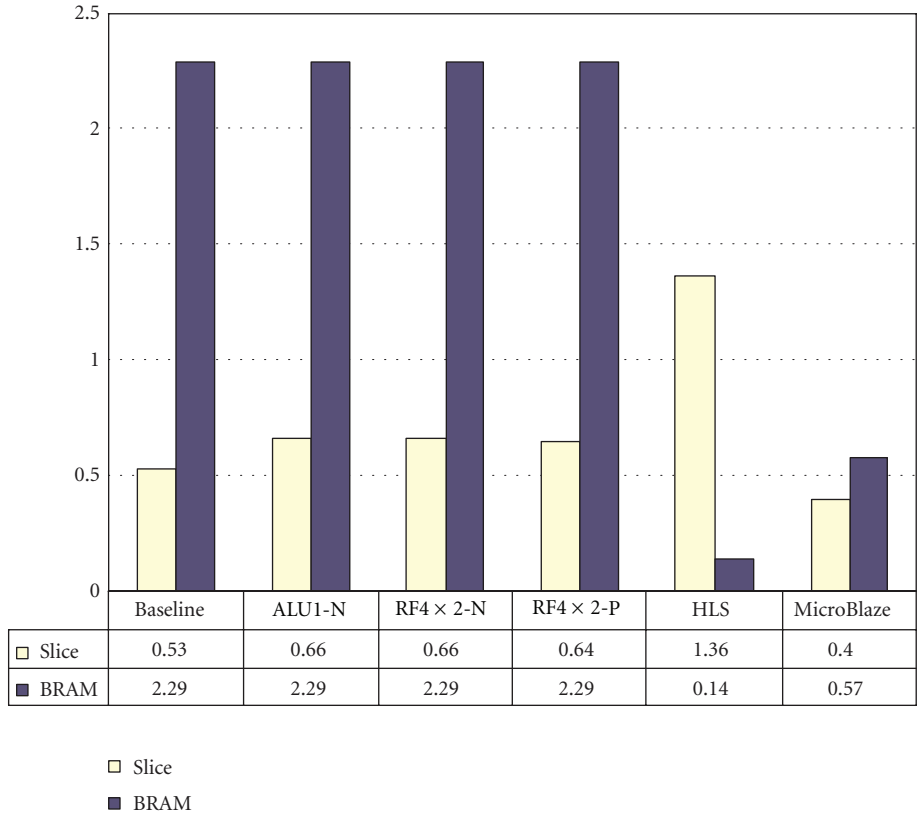
(a) Cost (b) Performance

FIGURE 19: *inv/forw 4 × 4*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.



	Baseline	ALU1-N	RF4 × 2-N	RF4 × 2-P	HLS	MicroBlaze
No. of cycles	2.51	1.8	1.23	1.52	1.83	6.14
Tclk	1.03	1.2	1.2	0.81	1.67	2.39
Texec	2.59	2.16	1.47	1.23	3.06	14.67

FIGURE 20: Performance comparison for *dct32* relative to manual implementation.



	Baseline	ALU1-N	RF4 × 2-N	RF4 × 2-P	HLS	MicroBlaze
Slice	0.53	0.66	0.66	0.64	1.36	0.4
BRAM	2.29	2.29	2.29	2.29	0.14	0.57

Figure 21: Area comparison for *dct32* relative to manual implementation.

respect to all the metrics except the number of used bRAMs. Moreover, the average generation time for *dct32* is 2.3 seconds while it took 3 man-weeks for the manual design. The fastest extracted design has only 23% of execution overhead and a negligible generation time compared to the manual design. Hence, we believe that the proposed data path extraction from C code is valuable technique for creation of an application-specific data path design. Moreover, all the generated designs outperform MicroBlaze: the best automatically generated design RF 4 × 2-*P* has 1.23 times longer execution time, where MicroBlaze has 14.67 times longer one. However, MicroBlaze utilizes smaller area: 0.4 slices and 0.57 bRAMSs, where our best design utilizes 0.64 slices ad 2.29 bRAMs.

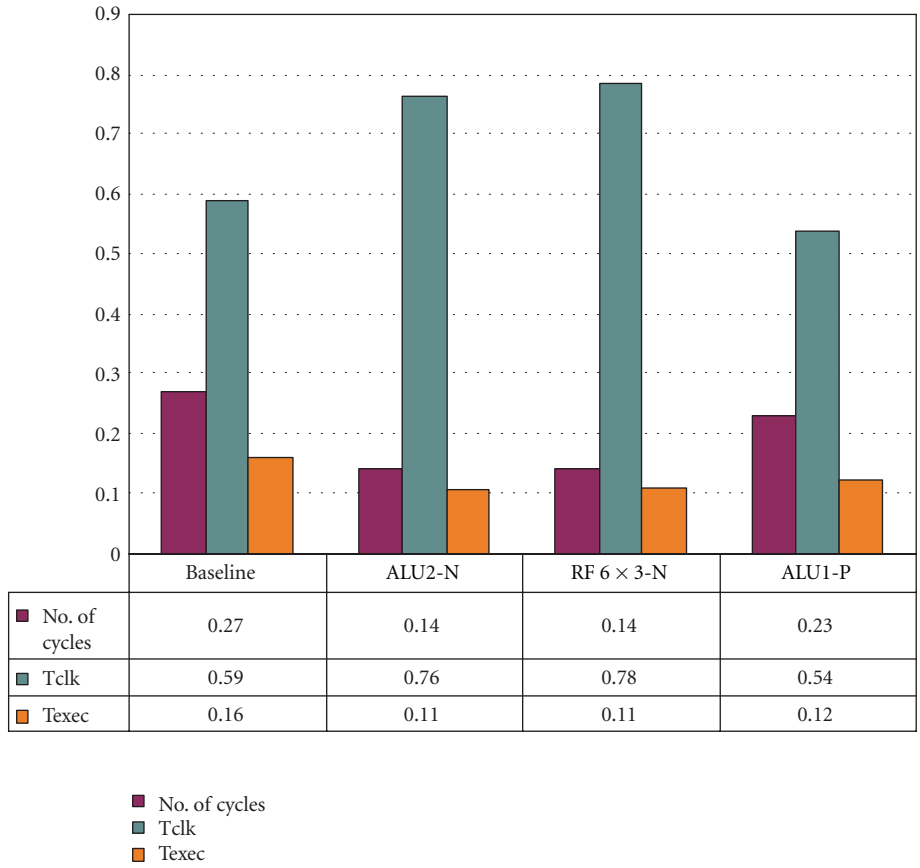
7.2. Mp3. Figures 22 and 23 plot the same performance and area metrics as the previous two figures, but relative to the implementation on *MicroBlaze*, because manual implementation for *Mp3* is not available. Also, the HLS tool could not be used, due to the capacity issues, hence we present the results for all the remaining data path designs. We, also, used the the same C code as an input to all tools. The graphs show following data paths:

(i) *Baseline*—corresponds to a pipelined version of a baseline design for the *dct32* used in Section 5,

(ii) *ALU2-N*—generated *nonpipelined* data paths for constraints ALU 2,

(iii) *RF 6 × 3-P and ALU1-P*—generated *pipelined* data path for constraint RF 4 × 2 and ALU 1, respectively.

With respect to performance, all the designs outperform the MicroBlaze implementation. The number of cycles improvements range from 73% to 88% (i.e., 0.27 to 0.12 on the graph), and the cycle time improvement form 24% to 46% (0.76 to 0.54). This significant improvement in both number of execution cycles and the cycle time directly translates to significant savings in total execution time. The best execution time is only a fraction on execution time on MicroBlaze (0.11, i.e., 89% improvement) and it is achieved while running on RF 4 × 2-*N* and RF 4 × 2-*P* data path configurations. The significant performance improvements are because of the use of extra resources: register file with more input/output ports and more functional units that facilitate efficient use of available parallelism. This directly translates into the use of more slices: from 3.53 to 6.12 times more than in the MicroBlaze implementation. Similarly to *dct32*, the tool maps all the storage elements to bRAMs on FPGA, and hence the high bRAM overhead compared to the MicroBlaze implementation.



	Baseline	ALU2-N	RF 6 × 3-N	ALU1-P
No. of cycles	0.27	0.14	0.14	0.23
Tclk	0.59	0.76	0.78	0.54
Texec	0.16	0.11	0.11	0.12

Figure 22: Performance comparison for *Mp3* relative to implementation on MicroBlaze.

8. Related Work

In order to accomplish performance goals, ASIPs and IS extension use configurable and extensible processors. One such processor is Xtensa [12] that allows the designer to configure features like memories, external buses, protocols, and commonly used peripherals [13, 14]. Xtensa also allows the designer to specify a set of instruction set extensions, hardware for which is incorporated within the processor. Our work presents algorithms that generate custom data path, which determines a set of (micro-) operations that can be preformed. This approach is orthogonal to ASIP/IS extension approach. The automatically generated data path can be used in ASIP or added to a processor, as custom hardware unit. In order to do so, one would need to identify the instructions that are to be executed on such a data path. The automated extraction of instructions from a set of operations is a topic of our future work.

The Tensilica XPRES (Xtensa PRocessor Extension Synthesis) Compiler [15] automatically generates extensions that are formed from the existing instructions in style of VLIW, vector, fused operations, or combination of those. Therefore, automated customizations are possible only within bound of those combinations of existing instructions. IS extensions also require the decoder modifications in order to incorporate new instructions. For example, having VLIW-style (parallel) instructions requires multiple parallel decoders [15], which not only increase hardware cost (that may affect the cycle time) but also limit the possible number of instructions that may be executed in parallel. Our approach also automatically generates data path which defines a set of possible (micro-) operations. However, in our approach, the decoding stage has been removed. Therefore, there is no increase in hardware complexity and no limitations on the number and type of operations to be executed in parallel. In case where the code size exceeds the size of on-chip memory, due to the number of operations that are specified to be executed in parallel, "instruction" caches and compression techniques may be employed, both of them have been in scope of our current research.

The IS extensions, in case of Stretch processor [16], are implemented using configurable Xtensa processor and Instruction Set Extension Fabric (ISEF). The designer is responsible for, using available tools, identifying the critical portion of the code "hot spot" and rewriting the code so the "hot spot" is isolated into the custom instruction. The custom instruction is then implemented in ISEF. Thus, the application code needs to be modified which

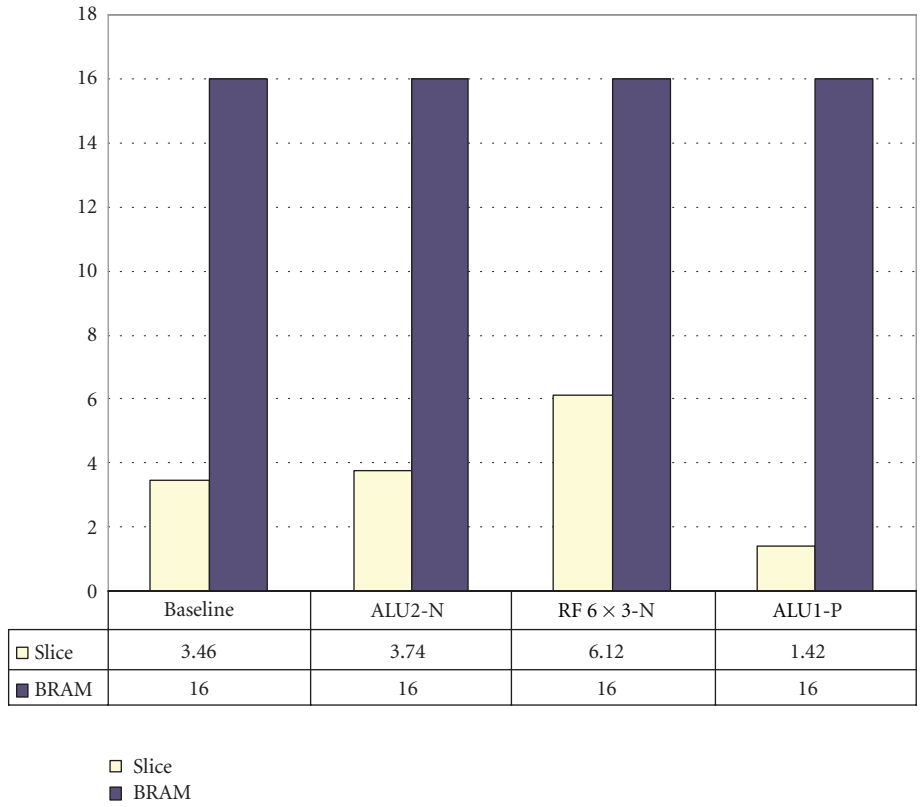


Figure 23: Area comparison for *Mp3* relative to implementation on MicroBlaze.

requires expertise and potentially more functional testing. The designer is expected to explicitly allocate the extension registers. In contrary, our approach allows but does not require C code modifications and does not require the designer to manipulate the underlying hardware directly. In both previous cases, it is required that the designer has expertise in both software and hardware engineering.

On the other hand, C-to-RTL tools, such as Catapult Synthesis [17], Behaviour Synthesizer Cyber [18], and Cynthesizer [19] generate the data path and the controller simultaneously, which may lead to capacity issues, like the one in the case of GSM algorithm [20]. Catapult [17] allows control over resource sharing, loop unrolling, and loop pipelining. It also provides technology-specific libraries [21] that allow specific hardware instances to be inferred from C code. However, this requires code modifications. As reported by Mentor Graphics, code modifications took one week while the synthesis took one day. Also, the biggest listed C code had 480 lines of code. Other examples published in [20] include W-CDMA 3G modem algorithm called EPC and 2D graphics acceleration algorithm IDCT (which has the same complexity as *dct32* used here). Unfortunately, no number of lines of code was reported. Behavior Synthesizer Cyber [18], in addition to the abovementioned, provides various knobs for fine tuning, such as multiple clocks, gated clocks, synchronous/asynchronous resert, and synchronous/asynchronous/pipelined memory. The C code is extended to describe hardware by adding support for bit-length and in-out declarations; synchronization, clocking and concurrency; various data transfers (last two often not required). Such description is called behavioral C or BDL. Therefore, as seen in [23], the existing C code needs to be modified for in/out declaration, fifo requests, and so forth. In addition to control over loop unrolling and pipelining, Cynthesizer [19] also provides control over operator balancing, array flattening, chaining, mapping of arrays or array indexes to memory, and so forth. The designer may also select a part of design to be implemented in gate level design in a given number of cycles. Some of examples of implemented algorithms include multimedia applications [24]: H.264, video scaling, DCT, IDCT, motion estimation, NTSC encoder, VC1; security algorithms [25]: AES, DES, SHA, and MD5 encryption/decryption standards; digital media and security applications for wireless devices [26]: Viterbi encoders and decoders and proprietary noise rejection algorithms.

In case of all of the tools, the data path is built "on the fly" and heavily codependent on controller generation. Moreover, the resulting controller is usually in FSM style. The use of the FSM imposes size constraints for the design. Some of the tools, like Behaviour Synthesizer Cyber [18], and Cynthesizer [19], do provide FSM partitioning or hierarchical FSMs in order to expand beyond these constraints. To overcome capacity issues Catapult then uses its hierarchical

engine to synthesize each function to concurrent hierarchical blocks with autonomous FSMs, control logic, and datapaths [27]. The fundamental difference is in the separation of a data path generation from a controller generation: this allows us to analyze code and perform profiling before the data path generation. Moreover, the separation of data path and controller generation reduces the problem size, therefore, reducing the size and quantity of the data structures that a tool needs to maintain and manipulate on. Besides, while all the above-mentioned tools do allow that a designer gives guideline to a tool, there is no mechanism by which a designer may influence a choice of particular components (other than inferring via code change in case of Catapult). Therefore, after the design has been made, designer may not make any modifications in the datapath. Contrary, the proposed technique separates creation of the data path and the controller, which automatically overcomes size constraint. Also, the designer may specify a subset of components and have the remaining of the data path automatically generated. Finally, the data path can be modified as little or as much after the automatic generation. Therefore, we find that providing designer with ability to control the automated design process and the ability to handle any size of C code are valuable assets in data path generation.

Many traditional HLS algorithms, such as [28, 29], create data path while performing scheduling and binding. The work in [28] uses ILP formulation with emphasis on efficient use of library components, which makes it applicable to fairly small input code. The work in [29] tries to balance distribution of operations over the allowed time in order to minimize resource requirement hence the algorithm makes decisions considering only local application requirements. The work in [30] takes into account global application requirements to perform allocation and scheduling simultaneously using simulated annealing. In contrast with the previous approaches, we separate data path creation from the scheduling and/or binding, that is, controller creation. This separation allows potential reuse of created data path by reprogramming, controllability over the design process, and use prelayout information for data path architecture creation.

The work in [31–35] separate allocation from binding and scheduling. The work in [31] uses "hill climbing" algorithm to optimize number and type of functional unit allocated, while the work in [32] applies clique partitioning in order to minimize storage elements, units, and interconnect. The work in [33] uses the schedule to determine the minimum required number of functional units, buses, register files, and ROMs. Then, the interconnect of the resulting data path is optimized by exploring different binding options for data types, variables, and operations. In [34], the expert system breaks down the global goals into local constraints (resource, control units, clock period) while iteratively moves toward satisfying the designer's specification. It creates and evaluates several intermediate designs using the schedule and estimated timing. However, all of the afore-mentioned traditional HLS techniques use FSM-style controller. Creation and synthesis of such state machine that correspond to thousands of lines of C code which, to the best of our knowledge, is not practically possible. In contrast to this, having programmable controller allows us to apply our technique to (for all practical purposes) any size of C code, as it was shown in Section 4.

Similarly to our approach, the work in [35] does not have limitations on the input size, since it uses horizontally microcoded control unit. On the other hand, it requires specification in language other than C and it produces only nonpipelined designs, none of which is the restriction of the proposed technique.

9. Conclusions and Future Work

In this paper, we presented a novel solution to constructing a processor core from a given application C code. We first create an initial data path design by matching code properties to hardware elements. Then, we iteratively refine it under given user constraints. The proposed technique allows handling of any size of C code, controllability of the design process, and independent optimization of data path and controller. We presented results for wide range of benchmarks, including industrial size applications like the MP3 decoder. Each data path architecture was generated in less than a minute allowing the designer to explore several different configurations in much less time than required for manual design. We define a total cost of a design in terms of total number of slices and RAMs for all selected components, and performance in terms of number of the execution cycles. Our experiments showed that up to 82.13% of slices and 87.46% of RAMs were saved. The number of execution cycles was 18.84% more in case of a single benchmark and for the remaining benchmarks, the maximum increase in the number of cycles was 0.68%. We measured design refinement quality on an example of *dct32* for which we synthesized all the designs on an FPGA board. We also showed that the best generated data path architecture is only 23% slower and had 2.29 times more BRAMs and 0.64 times slices utilized compared to the manual design. In the future, we plan of optimizing the generated core area, performance, and power by automatically determining the best control and data path pipeline configuration.

References

[1] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[2] D. Gajski, "Nisc: the ultimate reconfigurable component," Tech. Rep. TR 03-28, University of California-Irvine, 2003.

[3] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis CODES+ISSS*, pp. 21–26, September 2005.

[4] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 69–74, October 2005.

[5] D. Gajski and M. Reshadi, "Nisc application and advantages," Tech. Rep. TR 04-12, University of California-Irvine, 2004.

[6] J. Trajkovic and D. D. Gajski, "Generation of custom co-processor structure from C-code," Tech. Rep. CECS-TR-08-05, Center for Embedded Computer Systems, University of California-Irvine, 2008.

[7] B. Gorjiara, M. Reshadi, P. Chandraiah, and D. Gajski, "Generic netlist representation for system and pe level design exploration," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 282–287, ACM, New York, NY, USA, 2006.

[8] B. Gorjiara, M. Reshadi, and D. Gajski, "Generic architecture description for retargetable compilation and synthesis of application-specific pipelined ips," in *Proceedings of the Proceedings of International Conference on Computer Design (ICCD '06)*, 2006.

[9] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "An interactive design environment for C-based high-level synthesis," in *IESS*, A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F.-J. Rammig, Eds., vol. 231 of *IFIP*, pp. 135–144, Springer, 2007.

[10] Xilinx: MicroBlaze Soft Processor Core, 2008, http://www.xilinx.com/tools/microblaze.htm.

[11] R. Ang, http://www.cecs.uci.edu/presentation_slides/ESE-Back-End2.0-notes.pdf.

[12] Tensilica: Xtensa LX, 2005, http://www.tensilica.com/products/xtensa_LX.htm.

[13] Automated Configurable Processor Design Flow, White Paper, Tensilica, 2005, http://www.tensilica.com/pdf/Tools_white_paper_final-1.pdf.

[14] Diamond Standard Processor Core Family Architecture, White Paper, Tensilica, 2006, http://www.tensilica.com/pdf/DiamondWP.pdf.

[15] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[16] Stretch: S5000 Software-Configurable Processors, 2008, http://www.stretchinc.com/products/devices.php.

[17] Mentor Graphics Catapult Synthesis, 2008, http://www.mentor.com/esl/catapult/overview//index.cfm.

[18] NEC CyberWorkBench, 2008, http://www.necst.co.jp/product/cwb/english/index.html.

[19] Forte Design System Cynthesizer, 2008, http://www.forteds.com/products/cynthesizer.asp.

[20] Mentor Graphics Catapult Synthesis—Ericsson Success Story, 2011, http://www.mentor.com/esl/success/ericsson-success.

[21] Mentor Graphics Technical Publications: Designing High Performance DSP Hardware using Catapult C Synthesis and the Altera Accelerated Libraries, 2008, http://www.mentor.com/techpapers/fulfillment/upload/ mentorpaper_36558.pdf.

[22] Mentor Graphics Technical Publications: Alcatel Conquers the Next Frontier of Design Space Exploration using Cat-apult C Synthesis, 2008, http://www.mentor.com/techpapers/fulfillment/upload/mentorpaper_22739.pdf.

[23] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, Cyber," in *Proceedings of the Proceedings of the Conference on Design, Automation and Test in Europe (DATE '99)*, p. 83, 1999.

[24] Forte Design System Cynthesizer—Applications: Digital Media, 2008, http://www.forteds.com/applications/digitalmedia.asp.

[25] Forte Design System Cynthesizer—Applications: Security, 2008, http://www.forteds.com/applications/security.asp.

[26] Forte Design System Cynthesizer—Applications: Wireless, 2008, http://www.forteds.com/applications/wireless.asp.

[27] Mentor Graphics Catapult Datasheet, 2010, http://www.mentor.com/esl/catapult/upload/Catapult_DS.pdf.

[28] B. Landwehr, P. Marwedel, and R. Dömer, "OSCAR: optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proceedings of the European Design Automation Conference*, pp. 90–95, IEEE Computer Society Press, Grenoble, France, 1994.

[29] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[30] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 768–781, 1989.

[31] P. Gutberlet, J. Müller, H. Krämer, and W. Rosenstiel, "Automatic module allocation in high level synthesis," in *Proceedings of the Conference on European Design Automation (EURO-DAC '92)*, pp. 328–333, 1992.

[32] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[33] F.-S. Tsai and Y.-C. Hsu, "STAR: an automatic data path allocator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 9, pp. 1053–1064, 1992.

[34] F. Brewer and D. D. Gajski, "Chippe: a system for constraint driven behavioral synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 7, pp. 681–695, 1990.

[35] P. Marwedel, "The MIMOLA system: detailed description of the system software," in *Proceedings of the Design Automation Conference*, ACM/IEEE, 1993.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 484962, 11 pages
doi:10.1155/2012/484962

Research Article

Hardware and Software Synthesis of Heterogeneous Systems from Dataflow Programs

Ghislain Roquier, Endri Bezati, and Marco Mattavelli

Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland

Correspondence should be addressed to Ghislain Roquier, ghislain.roquier@epfl.ch

Received 15 July 2011; Revised 27 October 2011; Accepted 6 December 2011

Academic Editor: Deming Chen

The new generation of multicore processors and reconfigurable hardware platforms provides a dramatic increase of the available parallelism and processing capabilities. However, one obstacle for exploiting all the promises of such platforms is deeply rooted in sequential thinking. The sequential programming model does not naturally expose potential parallelism that effectively permits to build parallel applications that can be efficiently mapped on different kind of platforms. A shift of paradigm is necessary at all levels of application development to yield portable and scalable implementations on the widest range of heterogeneous platforms. This paper presents a design flow for the hardware and software synthesis of heterogeneous systems allowing to automatically generate hardware and software components as well as appropriate interfaces, from a unique high-level description of the application, based on the dataflow paradigm, running onto heterogeneous architectures composed by reconfigurable hardware units and multicore processors. Experimental results based on the implementation of several video coding algorithms onto heterogeneous platforms are also provided to show the effectiveness of the approach both in terms of portability and scalability.

1. Introduction

Parallelism is becoming more and more a necessary property for implementations running on nowadays computing platforms including multicore processors and FPGA units. However, one of the main obstacles that may prevent the efficient usage of heterogeneous platforms is the fact that the traditional sequential specification formalisms and all existing software and IPs, legacy of several years of the continuous successes of the sequential processor architectures, are not the most appropriate starting point to program such parallel platforms [1]. Moreover, such specifications are no more appropriate as unified specifications when targeting both processors and reconfigurable hardware components. Another problem is that portability of applications on different platforms becomes a crucial issue and such property is not appropriately supported by the traditional sequential specification model and associated methodologies. The work presented in this paper focuses in particular on a methodology for the generation of scalable parallel applications that provide a high degree of portability onto a wide range of heterogeneous platforms. We argue that to achieve such objectives is necessary to move away from the traditional programming paradigms and adopt a dataflow programming paradigm. Indeed, dataflow programming explicitly exposes the parallelism of applications, which can then be used to distribute computations according to the available parallelism of the target platforms. Moreover, the methodology described here has also the objective of raising the level of abstraction at all levels of the design stages involving human intervention for facilitating the overall design of complex applications onto heterogeneous systems, composed of multicore processors and FPGAs.

A key requirement in our design approach is that applications have to be portable and scalable. Portability ensures fast deployment of applications with minimal assumption on the underlying architecture, which drastically shortens the path from specification to implementation. The application should be able to run on any processing component architecture of a heterogeneous system from a single description, without code rewriting. Another important feature is that applications should also be scalable. It means that the

performance of the running application should scale with the available parallelism of the target architecture.

The following sections present the main stages of a dataflow-based approach that present the described features in the design of applications on heterogeneous platforms.

2. Related Works

Hardware-Software (HW-SW) codesign concept and fundamental ideas, that are also at the base of our work, have been introduced in the nineties [2, 3]. A formal definition of the term codesign is not unique. In the rest of the document codesign stands for the joint design of SW and HW components from a single-application description.

Our application model is in the line with model-based design. Model-based design was proposed to raise the level of abstraction when designing digital processing systems. High-level models provide useful abstractions that hide low-level details, such as platform independency, in order to ease analysis tasks. Prior research related to model-based design using data- and control-dominated models and a combination of both is the subject of a wide literature. Essentially the various methods proposed mainly differ by the model used and by the so-called model of computation (MoC).

Without claiming to be exhaustive, we can mention the POLIS [4] framework based on Codesign Finite State Machine (CFSM) that relaxes FSMs to communicate asynchronously. Such model has limited expressive power which is a key feature when targeting the design of critical reactive systems and results rather difficult to be used outside the scope of control-dominated systems.

By contrast to control-dominated models, data-dominated models such as communicating sequential processes (CSPs), dataflow graphs or Kahn process networks (KPNs) are preferred when dealing with stream processing algorithms.

SynDEx from INRIA [5] is one of such design approaches based on a restricted dataflow model. In this model, a vertex of the dataflow graph may be factorized (n-times repetitions during the graph execution) and conditioned (conditional statements by using the hierarchy in vertices and control values). The SynDEx model is deliberately restricted to ensure real-time constraints and consequently is not always adapted to model more general class of applications. Moreover, the high-level synthesis (HLS), that turns the model to HDL, is no more maintained.

Compaan/Laura from Leiden University [6] is based on KPN approach by using a subset of MATLAB code to model applications. The HW back-end (Laura) then turns the KPN model expressed by MATLAB code to VHDL. KPN-based models are much more expressive than more restricted MoC and can cover a much broader class of applications. However, since analyzability is, roughly speaking, inversely related to the expressiveness, it is somehow difficult to figure out the ability to generate the corresponding KPN models of more complex applications written in MATLAB.

PeaCE from the Seoul National University is an approach that lays at midway between dataflow (synchronous dataflow—SDF) and FSM [7]. This model raises the level of expressiveness, by enabling the usage of more control structures using FSM inside SDF vertices and vice versa. However, while PeaCE generates the code for composing blocks of the model both is SW and HW, it lacks code generation support for the blocks themselves and thus requires the definition of HW-SW blocks in later stage, which is time consuming when targeting several kinds of platforms.

Another interesting approach is SystemCoDesigner from the University of Erlangen-Nuremberg [8]. SystemCoDesigner is an actor-oriented approach using a high-level language named SysteMoC, built on the top of SystemC. It intends to generate HW-SW SoC implementations with automatic design space exploration techniques. The model is translated into behavioral SystemC model as a starting point for HW and SW synthesis. However, the HW synthesis is delegated to a commercial tool, namely, Forte's Cynthesizer, to generate RTL code from their SystemC intermediate model.

Several Simulink-based approaches have been also proposed to address HW-SW codesign [9]. Simulink, which was initially intensively used for simulation purposes, is becoming a good candidate for model-based design, particularly after the recent development of HW-SW synthesizers. Tools such as Real-Time Workshop for SW, or HW generators such as Synopsys Synphony HLS, Xilinx System Generator or Mathworks HDL coder are examples of these approaches. However, such methods are not always "off the shelf" and require the deep knowledge of a set of commercial tools and their appropriate usage.

Most of the approaches presented in the literature delegates the HW synthesis to commercial HLS tools. Mentor's Catapult, Synopsys's Synphony C Compiler, or Forte's Cynthesizer to name but a few are used with that purpose. Our approach shows such capabilities using free and open source tools. We believe that it is more flexible starting point, since those synthesizers can be easily tuned to target particular requirements.

3. Proposed Methodology

The paper presents a codesign environment that intends to address some of the limitations present in the state of the art particularly supporting SW and HW synthesis of the same source code with the synthesis of SW that scales on multicore platforms. It is expressly thought for the design of streaming signal processing systems. The essentials of the design flow are illustrated in Figure 1 and consist of the following stages.

(i) Dataflow-Based Modelling. We use an extension of the dataflow process network model (DPN), which is closely related to KPN, that enables to express a large class of applications, where processes, named actors, are written using a formal domain-specific language with the useful property of preserving a high degree of analyzability [10].

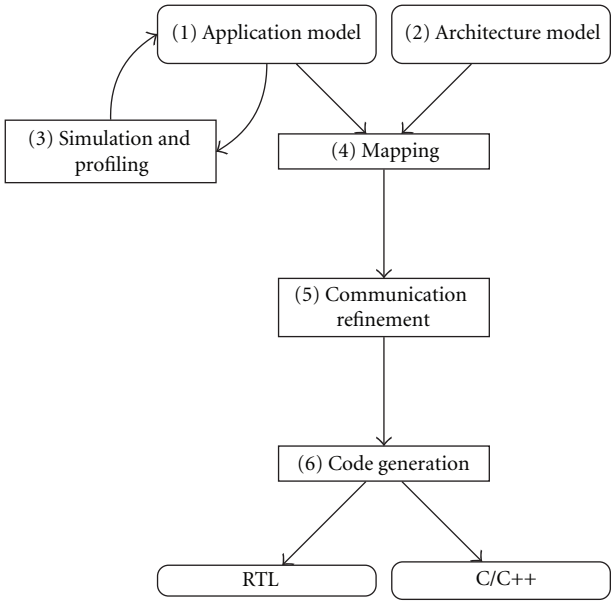


Figure 1: Overview of the design flow.

(ii) Architecture Model. We also employ an architecture model based on [5, 11] that enables to specify any architecture model for heterogeneous platforms composed by multicore processors and FPGAs at a high-level of abstraction.

(iii) Simulation and Profiling. We provide tools for functional validation, performance, and bandwidth estimations.

(iv) Mapping. HW-SW mapping can be both based on designer experience, or based on extracted metrics from the high level profiling or by more accurate profiling metrics if available from the platforms. Scheduling of SW partitions issues and available approaches are also discussed.

(v) Automatic Communication Interface Synthesis. Communication scheduling for interpartition communication as well as interfaces are automatically inserted in the design to be taken into account at synthesis stage.

(vi) Automatic Code Generation. HW and SW are automatically generated from CAL using ORCC and OpenForge synthesizers, including multicore support for SW components.

The codesign environment is implemented as an Eclipse plug-in built on the top of ORCC and OpenForge, open source tools that provide simulation and HW-SW synthesis capabilities. More details related to those capabilities are provided in the following sections. The complete tool chain is illustrated in Figure 2. The inputs of the tool chain are application (XDF) and architecture (IP-XACT) descriptions. The application is made by instantiating and connecting actors taken from an actor database (CAL).

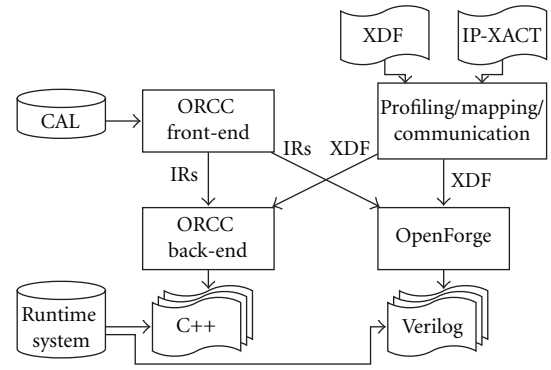


Figure 2: The cosynthesis tool chain.

4. Application Model: Dataflow with Firing

The dataflow paradigm for parallel computing has a long history from the early 1970s. Important milestones may be found in the works of Dennis [12] and Kahn [13]. A dataflow program is conceptually represented as a directed graph where vertices (named actors in the rest of the document) represent computational units, while edges represent streams of data. Figure 3 depicts a possible dataflow program. Formal dataflow models have been introduced in the literature, from Kahn process network (KPN) to synchronous dataflow (SDF) just to name a few. They differ by their models of computation (MoC) that define the behavior of the execution of the dataflow programs. There exists a variety of MoCs which results into different tradeoffs between expressiveness and efficiency.

In the paper, we use a model based on an extension of the dataflow process network MoC (DPN) [14]. Following the DPN MoC, actors execute by performing a number of discrete computational steps, also referred to as firings or firing functions. During a firing, an actor may consume data from input streams, produce data on output streams, and modify its internal state. An important guarantee is that internal states are completely encapsulated and cannot be shared with other actors, that is, actors communicate with each other exclusively through passing data along streams. This makes dataflow programs more robust and safe, regardless of the interleaving of actors. A firing rule is associated to each firing function, which corresponds to a particular pattern matching on the input streams and the current state. A firing occurs when its firing rule is satisfied, atomically and regardless of the status of all other actors.

In [10], authors presented a formal language for writing actors. The language, called CAL, is designed to express actors that belong to the DPN MoC. Of course, the language supports implementations of actors that can belong to more restricted MoCs, for example, CSDF, SDF, and so forth. The CAL language is a domain-specific language, it makes it possible to analyze actors easily and then determine their associated MoCs. It is an important property for many efficient optimizations that can be applied during code generation. An example of such optimizations is the static scheduling of actors (a correct-by-construction sequence of

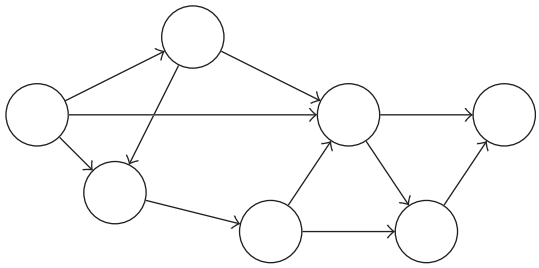


Figure 3: A simple dataflow program as a graph.

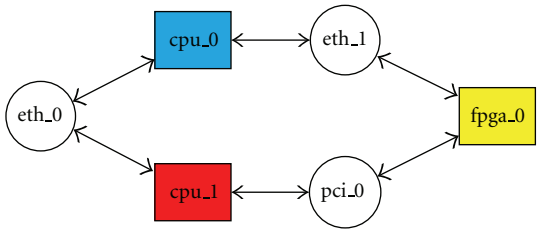


Figure 4: An example of a ring architecture composed of 3 operators and 3 media.

firings that can be executed without testing their firing rules) for some network partitions [15, 16].

CAL can also be directly synthesized to software and hardware [17–19]. Recently a subset of CAL, named RVC-CAL, has been standardized by ISO/IEC MPEG [20]. It is used as reference software language for the specification of MPEG video coding technology under the form of a library of components (actors) that are configured (instantiations and connections of actors) into networks to generate video decoders.

5. Architecture Model

The architecture model used in the design flow presented here, is based on the model proposed in [5, 11]. The architecture is modeled by an undirected graph where each vertex represents an operator (a processing element like a CPU or an FPGA in the terms of [11]) or a medium of communication (bus, memories, etc.), and edges represent interconnections viewed as a transfer of data from/to operators to/from media of communication. The model supports point-to-point or multipoint connections between operators.

The architecture model is serialized into an IP-XACT description, an XML format for the definition, and the description of electronic components, an IEEE standard originated from the SPIRIT Consortium. The architecture description is hierarchical and permits to describe architectures with different levels of granularity. For instance, a multicore processor can be represented as an atomic vertex or hierarchically exposing lower level details, where cores and memories become in turn atomic vertices. Figure 4 depicts a possible architecture with 3 operators connected with 3 point-to-point media.

6. Simulation and Profiling

RVC-CAL is supported by an interpreter that can simulate a dataflow graph of CAL actors. This interpreter is part of the Open RVC-CAL Compiler (ORCC). ORCC is a compiler infrastructure dedicated to RVC-CAL language [21]. More details on the ORCC infrastructure may be found in [18]. Essentially, the front end of ORCC transforms each actor into a corresponding intermediate representation (IR). The IR is then interpreted by the simulator.

A profiler, built on top of the interpreter, allows the user to extract high-level metrics when the execution of the dataflow program is simulated. The goal of the instrumentation is to determine the complexity of the actors. This complexity is extracted by counting instructions (assignment, load and store from/to the memory, loops, and if-statements) and operators in expressions (add, sub, mod, div, lsh, etc.). The instrumentation enables to extract metrics at an early stage of the system design without any information on the target architecture.

The relative bandwidth of FIFOs is also extracted in the profiling stage. The profiler can extract FIFO-related metrics by counting the number and size of data that are exchanged during execution.

7. Algorithm-Architecture Mapping

7.1. Partitioning. The partitioning consists of assigning each actor to a processing element (PE). A partition is defined as a subset of the application graph associated to each PE. In the proposed discussion of the design flow and associated results, partitions are statically defined (there is no actor migration from a partition to another at runtime). The static partitioning can be assigned manually according to the designer experience and/or requirements or automatically by using any optimization method aiming at optimizing appropriate objective function, using the metrics extracted during the profiling stage. However, design space exploration techniques that end up to automatic partitioning are not discussed in the rest of the paper. For more details on objective functions and associated heuristics, readers may refer to [22].

7.2. Scheduling. Once the partitioning is determined, the scheduling of actors assigned on a given PE consists of ordering their executions. In fact, the definition of appropriate scheduling strategies is not necessary in the context of reconfigurable hardware, since all the actors can run in parallel, regardless of the status of other actors. However, when the number of actors bound to a single PE is larger than one, a scheduler needs to be defined to execute actors sequentially. Defining a scheduling consists of determining an ordered list of actors for execution on a given partition. Such list can be constructed at compile time (static scheduling) or at runtime (dynamic scheduling). In our case, the scheduling of actors is deferred to runtime, since all of them are assumed to belong to the DPN MoC. In other words, the scheduler always needs to check if a given actor is enabled before execution, based on its firing rules. A simple scheduling strategy has

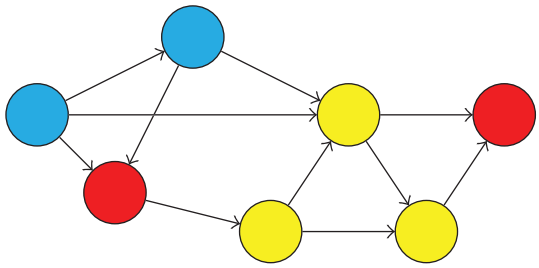


Figure 5: Mapping of the initial application graph.

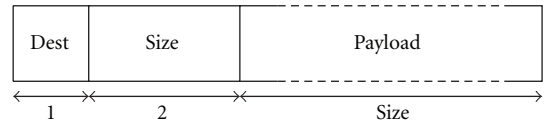


Figure 6: Header and the payload of the stored data in the serialization FIFO.

been selected that consists of using a round-robin scheduling algorithm, where each actor is checked one after the other by the scheduler for execution. Once an actor is selected, the scheduler tries to run it as long as it can while it matches one of its firing rules.

The runtime checks result in a significant runtime overhead [23] due to a large number of (sometimes unnecessary) conditional statements. However, scheduling statically (a subset of) those actors are sometimes possible when they belong to more restricted MoCs, namely, SDF and CSDF MoCs, that can help to reduce the overhead by removing some of those unnecessary checks. Several studies are devoted to solve this problem using other approaches and some interesting results showing relevant improvements are discussed in [23, 24].

The mapping stage is illustrated in Figure 5, where vertices of the dataflow graph are associated to PEs (the color of vertices comes from Figure 4).

8. Communication Scheduling

The HW-SW mapping results in a transformed dataflow graph. Transformations still need to be applied on the dataflow graph in order to exhibit the communications across partitions. The process mainly consists of transforming the initial dataflow graph by inserting additional vertices that represent communications between partitions, using the appropriate media between PEs from the architecture. Such transformation introduces special vertices in the application graph, which will encapsulate at a later stage the (de)serialization of data and the inclusion of the corresponding interfaces between partitions. This step is illustrated in Figure 1 where (de)serialization (resp., Ser. and Des.) and interface vertices are inserted.

The underlying DPN application model prevents from being able of scheduling communications statically. The serialization has the objective of scheduling the communications between actors that are allocated on different partitions at runtime. The fact is that when several edges from the dataflow graph are associated to a single medium of the architecture, data need to be interlaced in order to be able to share the same underlying medium.

In the case of serialization, on the sender side, "virtual" FIFOs are used to connect the serializer to incoming actors. By contrast with conventional FIFOs that store data and maintain the state (read/write counters), "virtual" FIFOs just maintain the state while data are directly stored into a single FIFO, shared by incoming actors. The idea behind such procedure is to emulate the history of FIFOs (emptiness, fullness) in order to fairly schedule data in the serialization FIFO. Data are scheduled by actors themselves, without using any scheduling strategy in the serializer. In order to retrieve data on the receiver side, a header is placed at the beginning of each data that defines the destination FIFO and the size (in byte) of the payload. This simple header is illustrated on Figure 6. On the receiver side, conventional FIFOs are used to connect the deserializer to outgoing actors. The deserializer is responsible to decode the header and put the payload to the appropriate destination FIFO.

For instance, in Figure 5, the blue partition has two outgoing FIFOs connected to the red partition. Thus, a serializer vertex is inserted in Figure 7.

Reconfigurable hardware and multicore processors can invoke various interprocess communication methods (shared memory, sockets, etc.) through various physical/logical interconnections (PCI Express, Ethernet, etc.) to implement the interaction between PEs. Interfaces communicate with other components via appropriate I/O. Interfaces are introduced during the synthesis stage and must be supported by libraries according to the nature of the PE presents on the platform. On the sender side, data from the serialization FIFO are packed (for instance, we use the maximum transmission unit in case of ethernet) and sent using the appropriate interface. On the receiver side, data are just unpacked and passed to the deserializer. In Figure 7 a PCIe interface is inserted and connected to the serializer previously cited.

9. Hardware and Software Code Generation

9.1. Software Synthesis. The software synthesis generates code for the different partitions mapped on SW PEs. ORCC compiler infrastructure is used to generate source code. The front end transforms each actor into a equivalent intermediate representation (IR). Then, a back end that translates the IR into C++ code has been developed. A naive implementation of a dataflow program would be to create one thread per actor of the application. However, in general, from the efficiency point of view it is not a good idea. In fact, distributing the application among too many threads results into too much overhead due to many context switches. A more appropriate solution that avoids too many threads is presented in [14]. It consists of executing actors in a single thread and using a user-defined scheduler, that selects the sequence of actor execution. The extension to multicore

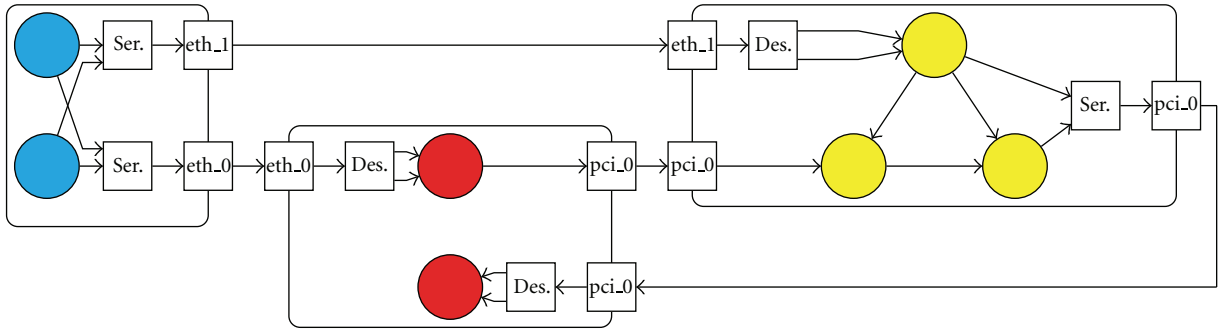


Figure 7: Communication refinement of the input application graph.

creates as many threads as existing cores. Since each core executes a single thread, threads are executed in parallel.

9.2. Hardware Synthesis. For the generation of the executable on programmable HW units, a synthesizable HDL code is automatically generated from the CAL dataflow program assigned to FPGAs. OpenForge, the HW synthesizer, is used to generate RTL code based on behavioral synthesis method translating the IR into HW descriptions expressed in Verilog [25] that can then be synthesized into implementations on FPGAs. More details about Openforge synthesis capabilities can be found in [17].

9.3. Runtime System. Runtime system libraries are needed for both SW and HW. Those libraries implement the FIFOs, serializers, deserializers, and the instantiation of the appropriate interfaces. On the software side, the dedicated runtime system provides the supports for the multicore requirements (threads, thread-safe queues, etc.). Note that the runtime is cross-platform and has been successfully tested on x86, PowerPC, and C66x DSP.

10. Experiments

10.1. Experiments on Multicore. CAL and its ISO standard subset RVC-CAL have been used with success for implementing video decoding algorithms in HW as reported and demonstrated in [17], and the dataflow model is clearly a good approach for exploiting massive parallel platforms such as FPGAs. Targeting less massively parallel systems, such as multicore platforms, it requires in addition appropriate methodologies for yielding efficient partitioning and scheduling.

The goal of the investigations presented here is to show how scalable parallelism can be achieved, in other terms that applications can be written at high level and their implementations can run faster when more parallelism is available in the implementation platform.

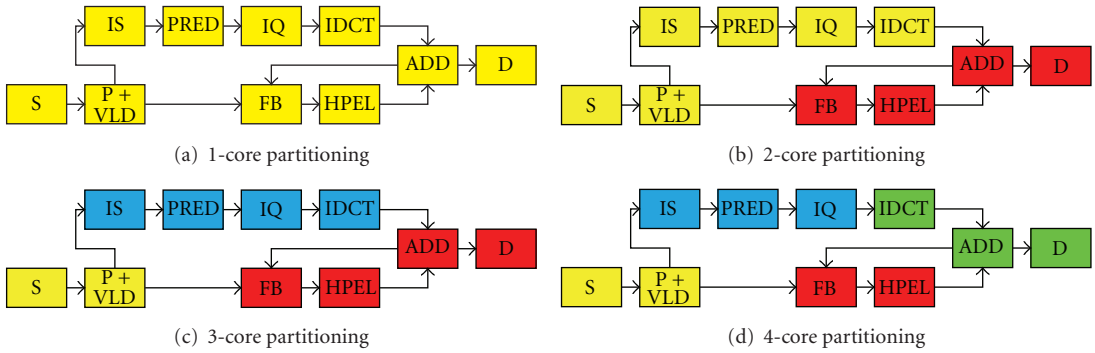
Prior researches have already reported implementation results of CAL programs on multicore. In [19], an implementation of an MPEG-4 SP decoder running on 2-core processor is reported. The ACTORS project [26] has reported the implementation of an MPEG-4 SP decoder with rather good speedup on 4-core processors [27]. The experimental results reported here show the evolution of these works in terms of improved scalability, portability on different platforms and increased throughput for the same application example. In this case study we have implemented 2 versions of an MPEG-4 Simple Profile decoder onto 2 different multicore platforms.

(i) Serial MPEG-4 Simple Profile. It is illustrated in Figure 8. It contains 13 actors that correspond to the entropy decoding (syntax parser and the variable length decoder), the residual decoding (AC-DC predictions, inverse scan, inverse quantization, and IDCT), and the motion compensation (framebuffer, interpolation, and residual error addition). The source (S) reads the bitstream, while the sink (D) displays the decoded video.

(ii) YUV-Parallel MPEG-4 Simple Profile. It is illustrated in Figure 9. The so-called parallelism is due to the fact that the color space components Y, U, and V can be decoded separately. It is composed by 33 instances of actors. Each branch includes two subnetworks "TX" and "MX", where "X" is to be replaced by the appropriate component, that, respectively, corresponds to the residual decoding and the motion compensation. The "P" subnetwork (with 7 actors) corresponds to the entropy decoder and finally the "M" actor merges the decoded components and sends data to the display (D).

Two implementation platforms have been used: a desktop computer with an Intel i7-870 processor, with 4 cores at 2.93 GHz, and a Freescale P4080 platform, using a PowerPC e500 processor with 8 cores at 1.2 GHz. The result of the SW synthesis from the dataflow program is a single executable file that is configured using the parameters that defines the partitions. A list of actors is extracted from each partition. Each list of actors is then used to create a thread where actors are instantiated and scheduled. Note that in both cases, it is a single executable file that is running on 1, 2, 3, or 4 cores. Three sequences at different resolutions have been encoded at different bitrates: foreman (QCIF, 300 frames, 200 kbps), crew (4CIF, 300 frames, 1 Mbps), and stockholm (720p60, 604 frames, 20 Mbps).

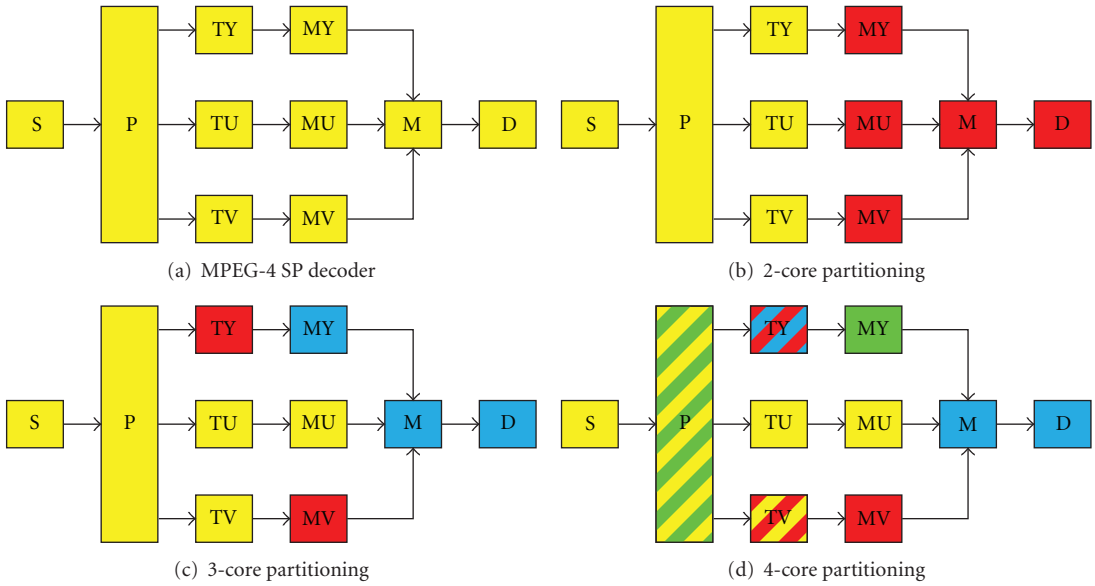
Figures 8 and 9 describe the different partitions used on the serial and the parallel versions of the decoders



(a) 1-core partitioning (b) 2-core partitioning

(c) 3-core partitioning (d) 4-core partitioning

Figure 8: Partitions of the serial version of the MPEG-4 SP decoder for 1, 2, 3, and 4 cores configurations.



(a) MPEG-4 SP decoder (b) 2-core partitioning

(c) 3-core partitioning (d) 4-core partitioning

Figure 9: Partitions of the parallel version of the MPEG-4 SP decoder for the 1, 2, 3, and 4 cores configurations.

Table 1: Framerate of the serial MPEG-4 SP decoder at QCIF, SD and HD resolutions.

Platform	Resolution	Framerate (no. of cores)			
		1	2	3	4
Intel i7-870	176 × 144	1788	3426	4416	5260
	704 × 576	126	203	219	307
	1280 × 720	37	63	71	84
Freescale P4080	176 × 144	275	520	676	913
	704 × 576	18	30	43	51
	1280 × 720	6	10	12	16

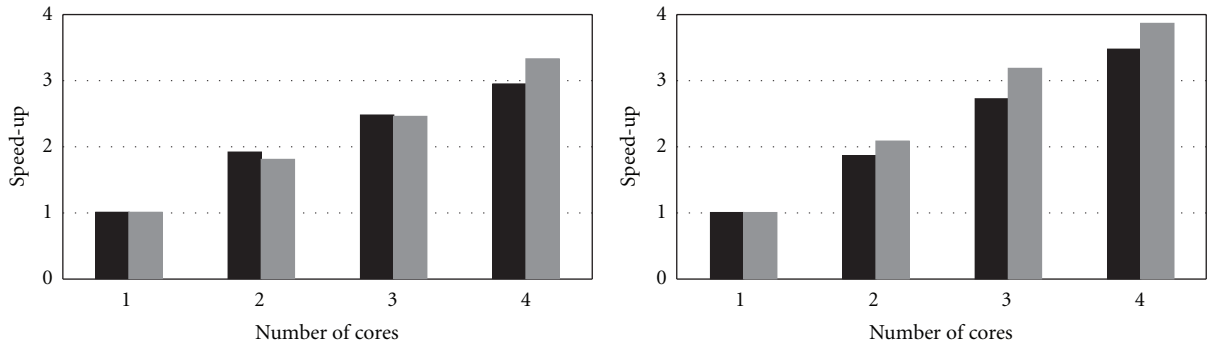
Table 2: Framerate of the YUV-parallel MPEG-4 SP decoder at QCIF, SD and HD resolutions.

Platform	Resolution	Framerate (no. of cores)			
		1	2	3	4
Intel i7-870	176 × 144	1580	2940	4303	5494
	704 × 576	104	178	267	340
	1280 × 720	34	62	75	89
Freescale P4080	176 × 144	223	465	711	853
	1280 × 720	15	30	43	52
	1280 × 720	5	9	13	18

respectively. The blocks represented by a stripe background are distributed over different partitions.

Tables 1 and 2 report the framerate in frame per second (fps) of the serial and the parallel decoders, respectively, on the 4 cores.

The resulting speedup is illustrated in Figure 10. It shows that it is possible to achieve significant speedups when adding more cores. It is also observed in the experiment that a near to linear speedup in most of the cases is obtained, which tends to indicate that the dataflow applications scale well on multicore processors. Particularly, on the P4080 at QCIF resolution, it is observed a more than linear scale. This anomaly is due to the reduction of the scheduling overhead when the number of the partitions increases.



(a) Speedup for the serial decoder at QCIF resolution

(b) Speedup for the parallel decoder at QCIF resolution

Figure 10: Speedup of the MPEG-4 SP decoder running on an X86 quad-core (in black) and on a PowerPC e500 8-core (in gray) processors.

Table 3: RVC-CAL JPEG encoder throughput for one image in ms.

Type	Resolution	FPGA Frequency	
		50 Mhz	80 Mhz
RVC-CAL HDL	512×512	48 ms	28.9 ms
Generated Code	1920×1080	373 ms	223 ms
Handwritten VHDL	512×512	31.2 ms	18.7 ms
	1920×1080	317 ms	190 ms

Figure 11 reports results related to the scalability that is also preserved when changing the resolution of the video format. The results show that both decoders produce real-time performances for HD resolution (720p60) starting from the 2-core configurations. Results that are remarkable considering the high-level starting point and the lack of any low-level and platform-specific optimizations. In terms of speedup factor versus the single-core performance, the results are slightly better than the ones presented in [27].

In term of absolute throughput, the experiment shows a very significant improvement. The normalized throughput results in macroblock (A macroblock corresponds to 16×16 pixels in MPEG-4 SP, which is equivalent to $6 \times 8 \times 8$ bytes in 4 : 2 : 0 format.) per second divided by the frequence—$\text{MB} \cdot \text{s}^{-1} \cdot \text{Hz}^{-1}$ are 5.94×10^{-6} $\text{MB} \cdot \text{s}^{-1} \cdot \text{Hz}^{-1}$ for the ARM11 in [27], 22.68×10^{-6} for the PowerPC and 60.41×10^{-6} for the Intel i7-870 (for the serial decoder at QCIF resolution).

Figure 11 shows that the parallel version of the decoder scales much better than the serial one. Moreover, the higher the resolution, the lower the speedup factor we obtain. This result is due to the fact that the application is constituted by a lower number of actors, a fact that reduces the number of possible partitions, thus reducing the possibility to balance equitably the processing load. By contrast, the parallel version is less sensitive to the resolution. Those results indicate that the parallel decoder seems to be a better starting point when targeting implementations on more cores.

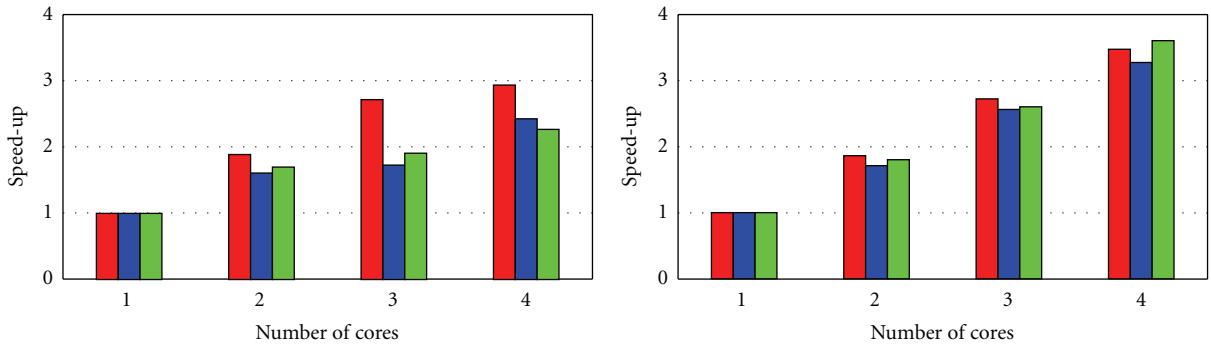
10.2. Experiments on Reconfigurable Hardware. The purpose of this experiment is to compare the HDL synthesis from a dataflow application with a handwritten one. To this end, a baseline profile JPEG encoder was developed and a VHDL JPEG encoder was taken from the OpenCores project [28]. Figure 12(a) represents the dataflow JPEG encoder where computation blocks are at the encoding DCT, quantization (Q), zigzag scan (ZZ), and variable-length encoding (VLC). As for the VHDL encoder which is represented in Figure 12(b) the DCT, Q, and ZZ are processed in parallel for the luma (Y) and chroma (UV) blocks. This VHDL encoder design was chosen for its dataflow resemblance with the RVC-CAL JPEG encoder.

The ML509 Virtex 5 board was used for both encoders. Table 3 indicates the throughput of both encoders for encoding two images with different resolutions and the throughput of this images for two different clock frequencies. The result of this experiment shows that the handwritten VHDL JPEG encoder is only 1.5 times faster when compared to the automatically generated HDL from the initial version of the dataflow JPEG encoder. One of the reasons why the VHDL encoder is faster is that it uses the FDCT IP core accelerator from Xilinx and that the luma and chroma blocks are processed in parallel. The splitting of the Y, U, and V components is a possible optimization for the RVC-CAL JPEG encoder.

Table 4 compares the resource usage of both encoders on the FPGA. The generated JPEG encoder uses 5% less registers, 4 times less LUT-FF pairs, and 2% more of DSP48E1 (which represents only 3% of the DSP block on a Virtex 5 FPGA, those DSP blocks are mainly used in the FDCT and in the quantization actors) than the handwritten VHDL JPEG encoder. Thus the generated HDL from the high-level version of the RVC-CAL JPEG encoder requires less FPGA logic resources than the handwritten one.

We can also notice that the RVC-CAL JPEG encoder can encode 4 Full HD images (1920×1080) in less than a second at 80 MHz and it can encode in real-time 512×512 images.

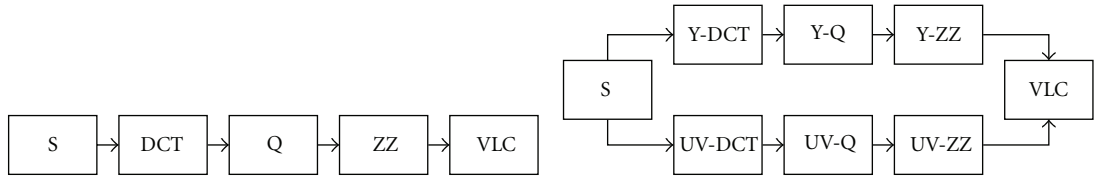
10.3. Synthesis of Both HW and SW from the Same Dataflow Description. The goal of the experiment is to validate the portability for the proposed design approach. To this end, additional to the JPEG encoder, we developed a JPEG decoder in RVC-CAL for creating a baseline profile JPEG



(a) Speedup for the serial decoder according to the resolution

(b) Speedup for the parallel decoder according to the resolution

Figure 11: Speedup of the MPEG-4 SP decoder on an Intel i7-870 processor at QCIF (red), SD (blue) and HD (green) resolutions.



(a) RVC-CAL JPEG Encoder

(b) Handwritten VHDL JPEG Encoder

Figure 12: Description of the RVC-CAL and handwritten VHDL JPEG encoders.

codec. We have implemented this codec onto three different platforms made of FPGAs and embedded processors. Figure 13 represents the JPEG codec. The host used in all platforms is the one that was used in the first experiment. The two first platforms are FPGA-based platforms. The first one is a proprietary board with a Xilinx Spartan3 FPGA and the second one is a ML509 board with a Virtex-5 FPGA as used in the previous experiment. The last platform is composed by a Freescale P2020 board, with 2 cores at 1 GHz. Ethernet, PCI express (PCIe) and RS232 are used to communicate between the host and the specialized PEs.

It has to be noticed that several mappings have been successfully tested. In fact, a single actor can seamlessly be synthesized to general-purpose processors, embedded processors, and FPGAs. A partition of the dataflow application can be swapped from SW to HW and vice versa, and all yield functionally equivalent implementations. For the sake of clarity, results are given only for a meaningful partitioning, separating the encoding and the decoding processes. More precisely, the partitioning of the application consists of assigning the whole encoding process on the specialized PE and the decoding process is done by the host. Note that on the P2020 the encoding is balanced between the 2 cores.

Results of the experiment are summarized in Table 5. Three different media of communication have been tested ("—" indicates that the corresponding communication link has not been tested in the executed experiments). The results using the serial link present low performances in term of fps. At least, it makes explicit that several interfaces can be used in the design flow.

The results using the Virtex5 and the PCI express interface are competitive with the ones presented in [6, 8].

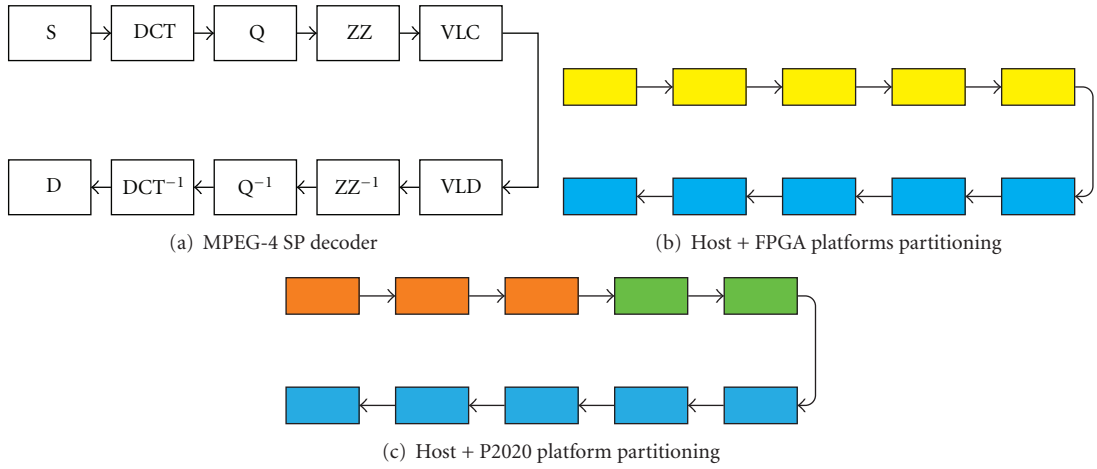
Table 4: FPGA occupation of the handwritten VHDL JPEG encoder versus the RVC-CAL JPEG Encoder.

Logic utilization	FPGA occupation			
	Handwritten		Generated	
	Usage	%	Usage	%
Registers	17869	11	10965	6
Slice LUTs	16439	19	14413	18
LUT-FF Pairs	11817	64	3504	16
Block RAM	35	13	43	14
DSP48E1s	2	1	18	3

Table 5: Framerate of the JPEG codec on 3 platforms with 3 interfaces and a 512×512 video resolution.

	Serial link	Ethernet	PCIe
Spartan3	0.2	3.7	N.A.
ML509 with Virtex5	0.2	—	8.5
P2020 with PowerPC	—	3.8	—

On the one hand, the encoder only implemented on the Virtex-5 can encode around 4 full HD frames per second at 80 MHz. While on the other hand, the decoder only, on the host, is able to decode at 135 fps 512×512 frames. This result indicates that either the interface bandwidth or the communication scheduling is the limit for the design performance.



(a) MPEG-4 SP decoder

(b) Host + FPGA platforms partitioning

(c) Host + P2020 platform partitioning

Figure 13: Description of the JPEG codec and the partitioning for the platforms.

11. Conclusions

The presented approach provides a unified design flow for the automatic synthesis of SW and HW components. The synthesis is directly obtained from high-level descriptions of both application programs and platform architectures. The high degree of abstraction enables the program to undergo several design iterations that enable rapid prototyping of applications onto architectures, validation, and testing of performances for different mappings by relying on automatic cosynthesis tools. Indeed, it consumes much less resource to refactor the dataflow program. In general, it is not possible to map imperative programs arbitrarily onto platforms without code rewriting. Rewriting the code, to fit a given platform, is time consuming and error prone, and usually most of the design time is wasted in the phase of debugging to reobtain design that works correctly. In our design flow, the design can go through many more design iterations, with less effort since just the mapping needs to be changed, that can shorten the path to implementation.

The paper has demonstrated, by experiments, both the scalability and the portability of real-world applications. Several implementations have been validated onto different platforms composed of both FPGAs and multicore processors. The results reported in the paper have only addressed and demonstrated the portability and scalability features provided by the approach. However, the potential of the approach can progress much further in terms of design space exploration capabilities. Future investigations will focus on the development of tools for automatic design space exploration, driven by objective functions that will use the metrics extracted during the profiling stage.

References

[1] G. De Micheli, "Hardware synthesis from C/C++ models," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 382–383, 1999.

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

[3] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for dsp applications," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16–28, 1993.

[4] F. Balarin, M. Chiodo, P. Giusto et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers, Norwell, Mass, USA, 1997.

[5] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proceedings of the 7th International Conference on Hardware/Software Codesign (CODES'99)*, pp. 74–78, Rome, Italy, May 1999.

[6] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using Khan process networks: the Compaan/Laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 340–345, February 2004.

[7] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. P. Joo, "PeaCE: a hardware-software codesign environment for multimedia embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, Article ID 1255461, 2007.

[8] J. Keinert, M. Streubuhr, T. Schlichter et al., "SystemCoDesigneran automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, article no. 1, 2009.

[9] S. A. Butt, P. Sayyah, and L. Lavagno, "Model-based hardware/software synthesis for wireless sensor network applications," in *Proceedings of the Electronics, Communications and Photonics Conference (SIECPC '11)*, pp. 1–6, April 2011.

[10] J. Eker and J. Janneck, "CAL Language Report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003.

[11] M. Pelcat, J.F. Nezan, J. Piat, J. Croizer, and S. Aridhi, "A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP '09)*, 2009.

[12] J. B. Dennis, "First version of a dataflow procedure language," in *Proceedings of the Symposium on Programming*, pp. 362–376, 1974.

[13] G. Kahn, "The semantics of simple language for parallel programming," in *Proceedings of the IFIP Congress*, pp. 471–475, 1974.

[14] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.
[15] J. Eker and J. W. Janneck, "A structured description of dataflow actors and its applications," Tech. Rep. UCB/ERL M03/13, EECS Department, University of California, Berkeley, 2003.
[16] M. Wipliez and M. Raulet, "Classification and transformation of dynamic dataflow programs," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP '10)*, pp. 303–310, 2010.
[17] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2011.
[18] M. Wipliez, G. Roquier, and J. F. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2009.
[19] I. Amer, C. Lucarz, G. Roquier et al., "Reconfigurable video coding on multicore: an overview of its main objectives," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 113–123, 2009.
[20] ISO/IEC 23001-4:2009, "Information technology—MPEG systems technologies—Part 4: Codec configuration representation," 2009.
[21] "Open RVC-CAL Compiler," http://orcc.sourceforge.net/.
[22] M. Mattavelli, C. Lucarz, and J.W. Janneck, "Optimization of portable parallel signal processing applications by design space exploration of dataflow programs," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2011.
[23] J. Ersfolk, G. Roquier, F. Jokhio, J. Lilius, and M. Mattavelli, "Scheduling of dynamic dataflow programs with model checking," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2011.
[24] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '09)*, pp. 565–568, IEEE Computer Society, Washington, DC, USA, 2009.
[25] "OpenForge," https://sourceforge.net/projects/openforge/.
[26] "ACTORS Project," http://www.actors-project.eu/.
[27] A. Carlsson, J. Eker, T. Olsson, and C. von Platen, "Scalable parallelism using dataflow programming," in *Ericson Review*, On-Line Publishing, 2011, http://www.ericsson.com/.
[28] "OpenCores," http://opencores.org/.

Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 649057, 14 pages
doi:10.1155/2012/649057

Research Article

High-Level Synthesis: Productivity, Performance, and Software Constraints

Yun Liang,[1] Kyle Rupnow,[1] Yinan Li,[1] Dongbo Min,[1] Minh N. Do,[2] and Deming Chen[2]

[1] *Advanced Digital Sciences Center, 1 Fusionopolis Way, No. 08-10 Connexis North Tower, Singapore 138632*
[2] *Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

Correspondence should be addressed to Yun Liang, eric.liang@adsc.com.sg

Received 20 July 2011; Accepted 25 October 2011

Academic Editor: Kiyoung Choi

FPGAs are an attractive platform for applications with high computation demand and low energy consumption requirements. However, design effort for FPGA implementations remains high—often an order of magnitude larger than design effort using high-level languages. Instead of this time-consuming process, high-level synthesis (HLS) tools generate hardware implementations from algorithm descriptions in languages such as C/C++ and SystemC. Such tools reduce design effort: high-level descriptions are more compact and less error prone. HLS tools promise hardware development abstracted from software designer knowledge of the implementation platform. In this paper, we present an unbiased study of the performance, usability and productivity of HLS using AutoPilot (a state-of-the-art HLS tool). In particular, we first evaluate AutoPilot using the popular embedded benchmark kernels. Then, to evaluate the suitability of HLS on real-world applications, we perform a case study of stereo matching, an active area of computer vision research that uses techniques also common for image denoising, image retrieval, feature matching, and face recognition. Based on our study, we provide insights on current limitations of mapping general-purpose software to hardware using HLS and some future directions for HLS tool development. We also offer several guidelines for hardware-friendly software design. For popular embedded benchmark kernels, the designs produced by HLS achieve 4X to 126X speedup over the software version. The stereo matching algorithms achieve between 3.5X and 67.9X speedup over software (but still less than manual RTL design) with a fivefold reduction in design effort versus manual RTL design.

1. Introduction

Field programmable gate array (FPGA) devices have long been an attractive option for energy-efficient acceleration of applications with high computation demand. However, hardware development targeting FPGAs remains challenging and time consuming—often requiring hardware design expertise and a register transfer level (RTL) algorithm description for efficient implementation. Manual design of RTL hardware often takes many months—an order of magnitude longer than software implementations even when using available hardware IP [1–3].

High-level synthesis (HLS) targets this problem: HLS tools synthesize algorithm descriptions written in a high level language (HLL) such as C/C++/SystemC. A HLL description can typically be implemented faster and more concisely, reducing design effort and susceptibility to programmer error. Thus, HLS provides an important bridging technology—enabling the speed and energy efficiency of hardware designs with significantly reduced design time. In recent years, HLS has made significant advances in both the breadth of HLS compatible input source code and quality of the output hardware designs. Ongoing development of HLS has led to numerous industry and academia-initiated HLS tools [4–25] that can generate device-specific RTL descriptions from popular HLLs such as C, C++, SystemC, CUDA, OpenCL, MATLAB, Haskell, and specialized languages or language subsets.

The advancements in language support for HLS mean that many implementations can be synthesized to hardware, but the original software design may not be suitable for hardware implementation. High level languages such as C have been shown as an effective means for capturing FPGA circuits when the C description is written specifically for HLS and hardware (e.g., C implementation is derived from manual FPGA implementation) [24]. However, although software

specifically written for HLS is sometimes available, the vast majority of software is not designed for synthesis.

There are many success stories of using HLS tools, but there is little systematic study of using HLS tools for hardware design, particularly when the original software is not written specifically for HLS. The code refinement process for HLS [25] and the benchmarks proposed by Hara et al. [26] bear similarity to our work. In [25], various coding guidelines are introduced to make code more HLS friendly and better performing. In [26], CHStone, a suite of benchmarks for HLS, is proposed, and the benchmarks are analysed in terms of source level characteristics, resource utilization, and so forth. In contrast, in this paper we present a systematic study of HLS including its productivity, performance, and software constraints. Without such a systematic study, the crucial insights into how to use current state of the art HLS tools are lacking, including the following.

(i) Performance of HLS-produced hardware on typical software.

(ii) Advantages and disadvantages of common code transformations.

(iii) Challenges and limitations of transforming code for HLS.

(iv) Coding style for hardware-friendly software design.

These limitations motivate us to investigate the abilities, limitations and techniques to use AutoPilot [9], one of the state of the art HLS tools. In this paper, we evaluate the achievable speedup over software design, the performance gap compared to manual design, coding constraints, required code optimizations, and development time to convert and optimize software for HLS. First, we evaluate AutoPilot using embedded benchmark kernels. Through evaluation of these kernels, we test the suitability of HLS on a wide range of applications as these kernels are widely used for various applications. However, these kernels are relatively small—real applications will contain multiple computation kernels that communicate, and more complex coding styles and data structures. Whereas a single small kernel can be easily transformed to correspond to a coding style well-suited for HLS, such transformations are more difficult for real applications. Thus, we select stereo matching [27] as a case study of HLS on complex real-world applications. Stereo matching is an important underlying technology for 3D video; the depth maps generated by stereo matching are used for interpolated video views and 3D video streams. Techniques employed for stereo matching algorithms include global energy minimization, filtering, and cost aggregation, which are used throughout image and video processing applications. In this study, stereo matching presents the additional challenge that software implementations are created by computer vision researchers unfamiliar with hardware design constraints. Thus, our stereo matching case study must optimize software not originally designed for HLS implementation.

HLS implementations of the embedded benchmark kernels achieve 4X to 126X speedup over the software implementation. In our stereo matching case study, we examine a variety of stereo matching algorithms, evaluate suitability of the software for AutoPilot compatibility, and convert four suitable software implementations not originally intended for HLS. Our experiments demonstrate that HLS achieves 3.5X to 67.9X speedup with 5X reduction in design effort compared to manual RTL design, where manual RTL design is still faster than the HLS-produced RTL.

This paper contributes to the study of HLS with the following.

(i) Evaluation of common barriers to HLS compatibility.

(ii) An effective HLS optimization process.

(iii) Evaluation of HLS on popular embedded benchmark kernels.

(iv) A case study of stereo matching algorithms for HLS suitability.

(v) Guidelines for mapping general-purpose SW to HW using HLS.

(vi) Directions for future study and enhancements of HLS tools.

The rest of this paper is organized as follows. Section 2 discusses the AutoPilot HLS tool and its supported features. Section 3 discusses the embedded benchmarks kernels, the stereo matching problem, and various stereo matching algorithms for HLS. Section 4 presents our HLS optimization process. Section 5 presents the experiments and results, and finally Section 6 presents our observations and insights on the productivity, usability, and software constraints to use HLS.

2. AutoPilot High Level Synthesis

AutoPilot is a commercial HLS tool developed by AutoESL (AutoESL was acquired by Xilinx in January 2011) [9] that supports input languages of C, C++, and SystemC, which can be annotated with directives to guide the high level synthesis with respect to the hardware implementation. AutoPilot supports a subset of C/C++; the main unsupported features are dynamic memory allocation and arbitrary indirection (pointers that are not static arrays). AutoPilot supports integer and floating point data types, as well as arbitrary precision fixed-point integer types. AutoPilot employs a wide range of standard compiler optimizations such as dead-code elimination, strength reduction, and function inlining. After these code optimizations, synthesis is performed at the function level—producing RTL modules for each function. Each module has private datapath and FSM-based control logic. By default all data arrays are mapped to local BRAMs; scalar variables are mapped to registers.

AutoPilot can apply optimizations to five groups of software source code: communication interfaces, function calls, for loops, data arrays, and labeled regions (a named code section enclosed by curly brackets). AutoPilot performs some optimizations automatically including expression balancing, loop unrolling, loop flattening, and simple array partitioning. However, AutoPilot is conservative in applying these optimizations to allow the user flexibility in optimizing

Table 1: Communication interface directives.

Directive	Description
`protocol`	Region is a protocol—do not reschedule operations
`interface`	For a communication interface, use a specified protocol (among predefined list)

Table 2: Function Call Directives.

Directive	Description
`dataflow`	Dataflow optimization to overlap computation between multiple function calls (or loop, or regions)—used with ping-pong or FIFO buffers
`instantiate`	Create a separate implementation of this function call—allow separate optimization of each "instantiated" function call
`inline`	Inline this function call (do not create separate level of RTL hierarchy)—allow resource sharing and optimization across hierarchy levels

the design for area, clock speed, throughput, or some combination of them. All of AutoPilot's optimizations are available as `#pragma` annotations and synthesis script directives.

After code optimizations, AutoPilot uses information about the implementation platform to further specialize the code to the particular platform. The hardware synthesis process then maps the optimized code to hardware, performing computation scheduling, resource binding, and pipelining. Finally, AutoPilot generates the interface code so that the synthesized code transparently maintains the same communication interface as the original implementation.

2.1. Communication Interfaces. Directives can specify that data accesses use particular communication interface protocols such as ACK, Valid, memory, or FIFO (among others). Additionally, users can define their own protocol and define a code region as a protocol so that code in that region is not rescheduled. Table 1 shows the details. For this work, we do not develop or use specialized communication protocols.

2.2. Function Calls. By default, AutoPilot generates RTL for each functional call as a separate module, and function execution is not overlapped. The directives (Table 2) can specify that functions can use fine-grained communication and overlap computation of multiple functions. In addition, directives can inline functions to prevent extra levels of RTL hierarchy and guide AutoPilot's optimization.

2.3. For Loops. For loops are kept rolled by default to maintain the maximum opportunity for resource sharing. AutoPilot directives can specify full or partial unrolling of the loop body, combination of multiple loops, and combination of nested loops. When accessing data in arrays, loop unrolling is commonly performed together with data array partitioning (next subsection) to allow multiple parallel independent array accesses, and thus creating parallelism opportunity along with pipelining opportunity. In addition, the loop directives as shown in Table 3 can specify expression balancing for improved fine-grained parallelism, and pipelining of computation within a code section.

Table 3: For loop directives.

Directive	Description
`loop_flatten`	Combine multiple levels of perfectly nested loops to form a single loop with larger loop bounds
`loop_merge`	Combine two separate loops at the same hierarchy level into a single loop
`loop_unroll`	Duplicate computation inside the loop—increase computation resources, decrease number of iterations
`pipeline`	Pipeline computation within the loop (or region) scope—increase throughput and computation resources
`occurrence`	Specify that one operation occurs at a slower (integer divisor) rate than the outer loop—improve pipeline scheduling, resource use
`expression_balance`	Typically automatic-code in the loop (or region) is optimized via associative and commutative properties to create a balanced tree of computation

Table 4: Data Array Directives.

Directive	Description
`array_map`	Map an array into a larger array-allow multiple small arrays to be combined into a larger array that can share a single BRAM resource
`array_partition`	Separate an array into multiple smaller arrays-allow greater effective bandwidth by using multiple BRAMs in parallel
`array_reshape`	First partition an array, then map the sub-arrays together back into a single array-creates an array with same total storage, but with fewer, wider entries for more efficient use of resources
`array_stream`	If array access is in FIFO order, convert array from BRAM storage to a streaming buffer

2.4. Data Arrays. Data arrays may be transformed to improve parallelism, resource scheduling, and resource sharing. Arrays may be combined to form larger arrays (that fit in memory more efficiently) and/or divided into smaller arrays (that provide more bandwidth to the data). In addition, when the data is accessed in FIFO order, an array may be specified as streaming, which converts the array to an FIFO or ping-pong buffer, reducing total storage requirements. Table 4 lists the array directives.

2.5. Labeled Regions. Some of the directives (as denoted in Table 3) can also be applied to arbitrary sections of code labeled and enclosed by curly brackets. This allows the programmer to guide AutoPilot's pipelining and expression balancing optimizations to reduce the optimization space.

Table 5: Kernel characteristics.

Kernel Name	Description
`Matrix Multiply (MM)`	Computes multiplication of two arrays (used in many applications). Array sizes are 1024 × 1024 or 512 × 512. Data type is 32-bit integer.
`Blowfish encrypt/decrypt`	Blowfish is a symmetric block cipher with a variable length key. It is widely used for domestic and exportable encryption.
`Adpcm encode/decode`	Adaptive Differential Pulse Code Modulation (ADPCM). It is a variation of the standard Pulse Code Modulation (PCM).
`AES`	Advanced Encryption Standard (AES) is a block cipher with option of 128-, 192-, and 256-bit keys and blocks.
`TDES`	TDES applies the Data Encryption Standard (DES) cipher algorithm three times to each data block.
`SHA`	SHA is the secure hash algorithm. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.

2.6. Other HLS Tools and Their Evaluation. CatapultC [13] and ImpulseC [17] are two widely used industry HLS tools. CatapultC and ImpulseC use transformations similar to AutoPilot but with fewer total features. CatapultC supports C++/SystemC, with data array and loop pragmas, but no function or dataflow transformations. Available pragmas include loop merging, unrolling and pipelining, and data array mapping, resource merging, and width resizing. ImpulseC uses a highly customized subset of the C language, with coding style restrictions to make the input more similar to HDL. As a result, ImpulseC supports a wide range of loop and data array transformations, again without function or dataflow pragmas (dataflow hardware is described explicitly). ImpulseC supports simultaneous loop optimization with automatic memory partitioning (using scalarization). Other ImpulseC pragmas are specifically related to the coding style, which requires explicit identification of certain variable types used for interfunction communication. LegUp is an academic-initiated open source HLS tool [23]. Given a C program, LegUp can automatically perform hardware software codesign, where some program segments are mapped to custom hardware (synthesized from the C-code) and the remaining code is mapped onto an FPGA-based soft processor. LegUp leverages the low-level virtual machine (LLVM) [28] infrastructure, which provides a variety of allocation, scheduling, and binding algorithms that can be applied to HLS.

Berkeley Design Technology, Inc. (BDTI) [29] offers HLS tool certification program which evaluates the capabilities of HLS tools in terms of quality of results and usability. However, the evaluation workload focuses exclusively on the digital signal processing applications. The participated HLS vendors perform a self-evaluation first and then BDTI certifies the results. In contrast, we evaluate AutoPilot using popular embedded benchmark kernels, and real-world stereo matching software. More importantly, we evaluate HLS using software not designed for HLS, and optimize ourselves (rather than allowing the tool vendor to find the best optimization), tracking design effort.

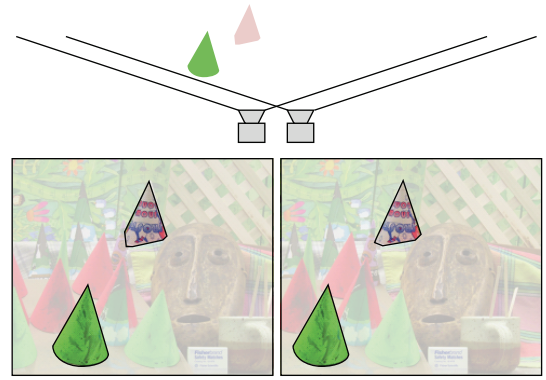


Figure 1: Example image capture for stereo matching. Two cameras physically offset capture an image of the same scene. The disparity between the objects in the left and right images infers information about object depth. One foreground and one background object are highlighted for clarity.

3. Benchmarking Applications

We evaluate HLS with AutoPilot using two categories of applications. We first use popular embedded benchmark kernels, which are widely used in various applications. However, these kernels are relatively small in code size and simple in data structure. Therefore, we also would like to evaluate HLS using real-world applications with complex code styles and data structures. We picked stereo matching for 3D image construction, which uses techniques also common for image de-noising, image retrieval, feature matching, and face recognition. In the following, we present the kernels and stereo matching algorithms in details.

3.1. Embedded Benchmark Kernel. To meet the stringent performance or energy constraints, embedded system designers often identify one or more computational intensive kernels (e.g., for signal processing, image processing, compression and decompression, cryptographic algorithms, etc.) for hardware acceleration. Although these computational-intensive kernels are often small in terms of code size, they play an important role for the overall performance and energy consumption as they account for significant execution time. Thus, efficient hardware implementation of these kernels could have significant impact on the overall system performance.

We choose a subset of applications from MiBench [30] including three common and representative cryptographic algorithms: AES, TDES, and SHA. The kernel description is shown in Table 5. Most of these kernels are implemented with a computational-intensive loop without dynamic memory allocation or complex pointer access.

Table 6: Common resolutions, frame rates, and disparity ranges (disparity range is computed with the same image sensor size (17.3 mm^2), focal length (2.5 mm), distance between cameras (12 mm) and image depth (0.5 m) for each image resolution. Parameters are based on the commercial cameras [31]. Computation scaling normalized to standard definition video).

Standard	Resolution	Maximum frame rate	Disparity range	Computation scaling
Middlebury Test Image	450 × 375	N/A	60	N/A
SD Video	640 × 480	30 fps	85	1
1080 p HD	1920 × 1080	120 fps	256	81
UHDTV	7680 × 4320	240 fps	1024	10000

3.2. Stereo Matching. Stereo matching is an extremely active area of research in computer vision, and an important underlying technology for 3D video. The depth maps generated by stereo matching are used for interpolated video views and 3D video streams. It measures the disparity between corresponding points in an object between two or more time-synchronized but spatially separated images, captured by a multiple camera system [31]. Input images are rectified to make the problem easy and accurate, so corresponding pixels are assumed to be on the same horizontal line in the left and right images. Disparity measures distance in pixels between an object in one image and the same object in another image, which is inversely proportional to object depth, as depicted in Figure 1. The depth map is subsequently used to generate interpolated view angles and 3D video streams.

Techniques employed for stereo matching algorithms include global energy minimization, filtering, and cost aggregation, which are used throughout image and video processing applications. In particular, these techniques are also employed for image de-noising, image retrieval, feature matching, and face recognition. The computational com-plexity of computer vision applications in general and stereo matching applications specifically demands hardware acceleration to meet frame rate goals, and its rapid evolution demands a shorter development cycle. For these reasons, stereo matching is representative of many computer vision applications that may demand acceleration; it requires a fast development cycle, and the available software is representative of algorithms developed for CPU implementations (but not designed or optimized for HLS). The depth map is used together with input color image(s) to produce synthesized views for 3D video applications. Computation complexity to measure pixel disparity has multiple scaling factors when we attempt to generate depth maps on successively higher resolution video. For a dense depth map, each pixel must be assigned a disparity, high-definition video requires high frame rate, and increased image resolution also increases disparity range. Table 6 shows the resolution, typical maximum frame rates, and disparity range in pixels for standard, high-definition, and next-generation high-definition video standards.

The large computation complexity of stereo matching requires hardware acceleration to meet performance goals, but stereo matching is also rapidly evolving, which demands reduced development time for hardware implementations. Thus, stereo matching is an attractive application for HLS-based hardware design.

We examined twelve sets of freely available stereo matching source code including an internally developed stereo matching code. The available source includes both published research work [32–40] as well as unpublished stereo matching source code. As stated previously, these codes are developed for stereo matching research, not suitability for HLS. Thus, despite this seeming wealth of available source code, many of the source packages use common programming techniques that are only efficient (and wise) to use in software, but unsuitable for HLS support. These include the following.

(i) Libraries for data structures (e.g., Standard Template Library).

(ii) OpenCV computer vision library for efficient implementations of common, basic vision algorithms.

(iii) Use of dynamic memory reallocation.

For example, as an effort to compare and evaluate many stereo matching algorithms, Scharstein et al. [32] developed a framework that implements many algorithms within a single software infrastructure. However, the implementation employs heavy use of memory re-allocation to instantiate the correct combinations of computation blocks and resize storage elements properly.

Stereo matching algorithms can be classified into global and local approaches. Global approaches use a complex optimization technique to simultaneously optimize the disparity matching costs for all pixels in the image. In contrast, local approaches compute the pixel matching costs individually for each pixel and concentrate on effective cost aggregation methods that use local image data as a likely source of semantic information. From the above set of algorithms, five of the twelve sets of source code can be transformed for high level synthesis compatibility. For each of the five algorithms, we perform transformations to improve suitability for HLS, but we do not redesign the algorithm implementation with HLS in mind.

We test two global approaches, Zitnick and Kanade [33], and constant-space belief propagation [34] and three local approaches, our internally developed stereo matching code, a scanline-optimized dynamic programming method, and cross-based local stereo matching [40]. Each of these algorithms uses differing underlying techniques to generate depth maps. We do not aim to judge the relative merits of different stereo matching approaches in terms of depth map accuracy. Rather, we discuss algorithm and implementation

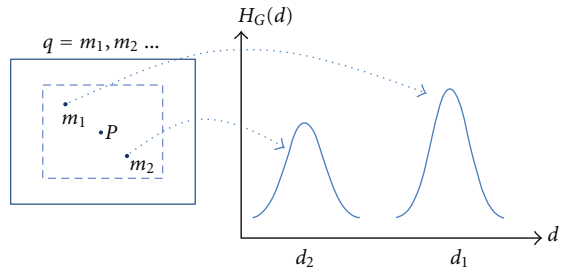


Figure 2: Calculation of the adaptive support function with Gaussian filters. Horizontal axis d represents disparity, $H_G(d)$ is the Gaussian filter amplitude. Point m_1 above has a larger $H_G(d)$ value because of closer color-space value.

details that make the algorithms more or less suitable for HLS.

3.2.1. Zitnick and Kanade (ZK). The algorithm proposed by Zitnick and Kanade [33] generates dense depth maps under two global constraints: (1) uniqueness—each pixel in the depth map corresponds to one and only one depth (and thus disparity value) and (2) smoothness—in most portions of the depth map, the depth of adjacent pixels is continuous. The implementation of the ZK algorithm is based on a 3D array, with one entry for each possible combination of pixel and disparity. The ZK algorithm uses a large, dense, 3D array of data for computation and storage; although the code is compatible with AutoPilot's language restrictions, the access order is not suitable for streaming (to reduce storage needs), and bitwidth reductions are insufficient to reduce storage needs. Therefore, due to the infeasible storage requirements, we omit ZK from detailed synthesis results.

3.2.2. Constant Space Belief Propagation (CSBP). The constant space belief propagation algorithm [34] also generates dense depth maps based on a global energy minimization solution. In original belief propagation [35], data cost is computed per pixel and disparity value. Then, each pixel iteratively updates messages with its 4 neighbors based on the smoothness constraint, and the final disparity is estimated as the minimum cost. CSBP refines BP by reorganizing computation so memory use is independent of the maximum disparity (but scales with image size). Hierarchically, pixel messages are computed on down sampled versions of the image and successively refined as the image is scaled towards the original resolution. Thus, CSBP scales the computation hierarchy in order to limit the maximum memory consumption.

3.2.3. Bilateral Filtering with Adaptive Support Function (BFAS). The BFAS algorithm is a new local stereo method developed by a team of our computer vision researchers as a driver algorithm to study HLS capabilities. It consists of an initial estimation using absolute difference between pixel values, multiscale image downsampling [41] and the fast bilateral filtering method [42] for initial cost aggregation, and refinement using an adaptive support function. The depth map is computed using winner-takes-all voting and occlusion via cross-checking left and right disparity maps.

Following the depth map computation, we can optionally refine the depth map quality in a postprocessing step that uses an adaptive support function. In this aggregation step, each pixel's contribution to the aggregated choice is scaled based on the distance (within the support window) from the center of the window, and color-space distance as shown in Figure 2.

3.2.4. Scanline Optimization (SO). Scanline optimization [32] (Software is an internally developed version of the algorithm in [32].) is a simple 1D-optimization variant of a dynamic programming stereo matching formulation. Each individual row of the image is independent—pairwise matching costs are computed for each pixel and disparity, then the minimum cost path through the matrix of pairwise matching costs simultaneously optimizes for the matching cost function and the smoothness constraint (small disparity change between adjacent pixels). Optionally, SO can generate a single depth map, or generate left and right depth maps that can be used with a cross-checking technique to compute pixel occlusion within the depth maps.

3.2.5. Cross-Based Local Matching (CLM). Cross-based local matching [40] is, like the BFAS algorithm, a local matching algorithm. However, whereas the BFAS algorithm uses a fixed window size and Gaussian filtering for aggregation, CLM uses an adaptive window shape and size to determine the pixels to aggregate. The support region consists of a region of contiguous pixels where the pixel's luminance value is within an empirically selected delta. Each pixel's support region is defined based on four parameters, pixel distance for ±horizontal and ±vertical, which forms a cross in the center of the region, and defines the bounds of the irregularly shaped region. Support regions are computed independently for the left and right images, and pixels that are in both the left and right support regions are used for cost aggregation.

4. Optimization Process

In total, we will evaluate the AutoPilot high level synthesis output using eight embedded benchmark kernels (matrix multiply, blowfish encrypt/decrypt, adpcm encode/decode, AES, TDES, SHA) and five stereo matching algorithms (BFAS, CSBP, SO without occlusion, SO with occlusion, and CLM). For all these benchmarks, we perform a five-step optimization process where applicable to convert for HLS compatibility and optimize the hardware implementation. The five optimization steps are baseline implementation, code restructuring, bitwidth reduction and ROM conversion, pipelining, and parallelization via resource duplication. These optimizations are intended to reduce resource use, improve resource utilization, expose software features that can be pipelined and/or parallelized, and take full advantage of available FPGA resources.

4.1. Baseline—Minimum Modifications. For each benchmark, we generate a baseline implementation—the minimum code modifications so that the algorithm can be properly synthesized using AutoPilot. For all the embedded kernels, they are AutoPilot synthesizable without code modifications. However, for stereo matching algorithms, we have to perform some modifications including conversion of dynamic memory to static declarations, conversion of memcpy and memset calls to for loops, and conversion of arbitrary (run-time changing) pointers to static pointers to memory locations.

All the baseline implementations are AutoPilot compatible, and produce hardware designs that produce correct output. However, some of the benchmarks use significantly more BRAM resources than available in the FPGA chip we use (Xilinx Virtex-6 LX240T). In addition, the BFAS software versions also have complex computation, which causes overconstrained use of LUT, FF and DSP resources as well. For the kernels, all of them can fit in the FPGA except matrix multiplication. As for the stereo matching algorithms, only SO without occlusion can fit in the FPGA after minimum modifications.

In addition to area inefficiency, these designs are always slower than the original software, sometimes by an order of magnitude. These results are expected, as AutoPilot's default settings do not unroll loops, pipeline computation, or transform data storage elements. In addition, the slowdown is exacerbated by several factors including reduced efficiency of for loops versus memset/memcpy calls, limited pipeline and parallelism, inefficient datapath width, and the difference between the CPU clock period and the achievable FPGA clock period. All of these reasons will be eliminated or mitigated by the following optimization steps.

4.2. Code Restructuring. For the embedded benchmark kernels we perform various code restructuring transformations including data tiling and block merging. For matrix multiplication, we perform data tiling to the source code since the array size is too big to fit into the BRAM of FPGA. In other words, a tile of array A and a tile of array B are copied from off-chip memory to on-chip memory for multiplication and then the computed data are written back to the off-chip memory array C. For example, with an array size of 1024×1024 and a tile size of 64×64 elements the array is divided into 256 tiles. In our experiments, we explore different tile sizes (8×8, 16×16, 32×32, 64×64, and 128×128). For matrix multiplication, one of the stringent resources is on-chip BRAM. Before tiling, the estimated number of required BRAMs is 5595 for 1024×1024 arrays, which is too big to fit into the FPGA. After tiling, the designs of one tile can fit into FPGA with less BRAMs (e.g., only 48 BRAMs are required for tile size 64×64). More importantly, tiling enable us to run multiple tiles in parallel (Section 4.5).

To efficiently use memory bandwidth, we transform arrays to increase element size and match the system's bound on memory access bit-width. However, the computation is still performed on the original, smaller element. Finally, to enable a powerful AutoPilot optimization—array streaming (Section 4.3), we change parameters passed in function calls from arrays of data to pointers.

For the stereo matching algorithms, the most important code restructuring task is to partition the image into subimages that can independently compute disparity and depth information. In all of the algorithms, the largest required data storage element(s) are directly proportional to the size of image processed. For the SO algorithm, this conversion is relatively simple: disparity computation is dependent within one row of pixels but independent between rows. However, the CSBP, BFAS, and CLM algorithms use windowed computation and a support window that spans both rows and columns of the image. These algorithms employ averaging, interpolation and filtering. Thus, we must partition the image into overlapping windows so that the computation result is correct.

In addition, we also perform function merging, interchange nested loops to improve parallelism opportunity, and share internal memory buffers to reduce resource use if it is possible. At this stage in optimization, we perform these code transformations manually. Although AutoPilot has synthesis directives that can merge loops or eliminate memory buffers that are used in FIFO order, these directives are relatively limited compared to transformations we can perform manually.

4.3. Reduced Bitwidth and BRAMs. The bit-width optimization is mainly effective for stereo matching algorithms; their computation involves a lot of arrays and the array element bitwidth can be reduced for some of these arrays. The bitwidth reduction can improve both the latency and resource usage. As for the embedded kernels, we observe fewer opportunities to perform these optimizations.

Throughout all of the stereo algorithms, full-width integer data types are commonly used for convenience. However, based on operand range analysis, we can reduce the operand bit-width. Similarly, floating point computation is used for convenience, but in these applications all values are still in the range of 0–255 (8-bit RGB components), with constants accurate to within 0.001. This relatively low accuracy still requires 10 fractional binary digits, but the combination of bitwidth reduction and smaller, simpler functional units can offer significant gain in terms of both latency and resource.

Using constant propagation, AutoPilot can sometimes determine that a set of variables have a fixed range and automatically reduce the bit-width of those variables to reduce storage requirements. However, this automatic bit-width reduction is not compatible with AutoPilot's array directives; when we wish to use array_map to share a BRAM between multiple data arrays, AutoPilot will not automatically reduce the variable size of the arrays. Therefore, this optimization step is a multistep process. First, for each data array in the design, we determine the operand range and redefine the array using AutoPilot's fixed-point integer (ap_fixed) variables to reduce the per-element storage requirement. Then, we use the array size in number of elements and access order to determine what resources should be used.

BRAM optimizations are effective for the embedded benchmark kernels and stereo matching algorithms. For

```
ap_fixed<20,3>  m_bI_F[101]  = {…};
ap_fixed<20,3> m_bI1_F[101] = {…};
ap_fixed<20,3> m_bI2_F[101] = {…};
ap_fixed<20,3> m_bI3_F[101] = {…};

#pragma       AP       array_map       instance=m_BI
variable=m_BI_F,m_bI1_F,m_bI2_F,m_bI3_F vertical

RecursiveGaussian_3D(…, m_BI_F[NumOfI-1],
m_bI1_F[NumOfI-1], m_bI2_F[NumOfI-1],
m_bI3_F[NumOfI-1]);
```

Figure 3: Code Example 1—array map directive to save BRAM use.

```
VoteDpr = 0;
count   = pDprCount[0];
for(d = 1; d < DprRange; d++){
#pragma AP unroll complete
#pragma AP expression_balance
   if(pDprCount[d] > count){
      count   =  pDprCount[d];
      VoteDpr =  d;
   }
}
```

Figure 4: Code example 2—loop unroll and expression balancing for fine-grained parallelism.

arrays with few total elements (in our case, less than 100) and statically determined access order, we use complete array partitioning which directs AutoPilot to use registers instead of BRAMs. For the other arrays, we search for combinations of arrays where the access order is synchronized or array access is entirely disjoint. For these access patterns, sharing the same physical BRAM does not result in additional read or write latency. Therefore, for such arrays, we use array_map to combine the arrays and share physical BRAMs. For example, in Figure 3, we show code extracted from BFAS; there are 4 parameter arrays with 101 entries each that are used in only one function in synchronized order. The default AutoPilot implementation uses 1 BRAM each for the arrays although the total storage bits used is much less than an 18 K BRAM. Therefore, we use the array_map pragma to map the arrays together into a single array that is stored in a single BRAM.

It is important to note that AutoPilot provides one powerful BRAM optimization—array streaming. The array stream pragma converts data arrays that are accessed in FIFO order into smaller, fixed-size buffers (significantly reducing BRAM use), and also allows dataflow optimizations which overlap computation of multiple RTL blocks in the same manner as pipelining does on smaller computation units. This optimization not only reduces BRAM usage but can have significant impact on performance improvement. However, it is not always feasible to apply this optimization as the data has to be written and read in FIFO order. Of all our benchmark applications, we only can apply the array stream optimization for the three cryptographic algorithms, AES, SHA, and TDES. These algorithms process streams of data in a sequential manner (e.g., block by block). More importantly, the data is written and read in FIFO order. Because of the requirement for this FIFO order, the array stream optimization is not available for the other embedded benchmark kernels or the stereo matching algorithms due to the complex data access order.

4.4. Pipelining and Loop Optimization. Through the previous three steps, we have reduced the amount of computation and memory resources. In this step, we examine the computation loops in the program and apply loop pipelining, loop merging, loop unrolling, loop flattening, and expression balancing to optimize performance. Because of the manual transformations in the code restructuring step, there are relatively few opportunities for loop merging, but it is used in a few cases to combine initialization loops with different loop bounds. When possible, we convert imperfectly nested loops to perfectly nested loops to allow loop flattening, which saves 1 cycle of latency for each traversal between loop levels.

For inner loops, we normally use pipelining to improve the throughput of computation. Using the pipeline directive, we set an initiation interval (II) of 1 as the target for all pipelined code. In most cases, AutoPilot can achieve this initiation interval. However, in some cases the computation on the inner loop requires multiple reads and/or writes from/to different addresses in the same BRAM. Thus, for these loops the initiation interval is longer to account for the latency of multiple independent BRAM reads/writes.

In some cases, when the inner loop has a small loop bound and loop content is performing a computation or search (rather than memory writes), we use complete unrolling and expression balancing instead of pipelining. For example, in Figure 4, we show a code section from CLM; instead of pipelining the inner loop computation, we fully unroll the inner loop (with DprRange = 60), and then use expression balancing to perform the search for a maximum pDprCount value in parallel instead of sequentially.

For this step, the best benefit is available when computation loops use static loop bounds—a static loop bound allows AutoPilot to perform complete unrolling on the inner loop to increase pipeline efficiency, or partially unroll by a known factor of the upper bound.

In general, performance can be improved via loop optimizations at the expense of extra resource usage. For example, both loop unrolling and pipelining enable multiple loop iterations to be executed in parallel with more resource usage (e.g., registers or functional units), but different optimizations improve performance and use resources in different ways. More importantly, it is possible to apply multiple optimizations together, that is, loop pipelining unrolled loops. Furthermore, if the computation can be divided into different independent small parts, we can instantiate multiple computation pipelines to parallelize the computation by duplicating the logic (Section 4.5). Hence, for loop optimizations, it is important to consider resource use; loop optimizations may improve performance, but can limit flexibility of implementing multiple computation pipelines in the parallelization and resource duplication step.

```
#define FS_UNROLL 2
ap_fixed<22,9> DispCost[FS_UNROLL][W*H];
ap_fixed<22,9> CostL[FS_UNROLL][W*H];
#pragma AP array partition complete dim=1
variable=DispCost,CostL

for(int k=Min; k<=Max; k += FS_UNROLL){
   FL_00:for(int l=0; l< FS_UNROLL; l++){
   #pragma AP unroll complete
      SubSampling_Merge(…,DispCost[l],…, k+l);
      CostAgg(…,DispCost[l],CostL[l],…);
      Cost_WTA(…,CostL[l],…, k+l);
   }
}
```

Figure 5: Code Example 3—array partition and complete loop unroll to expose functional parallelism.

4.5. Parallelization and Resource Duplication. At this step, we examine the synthesis result of the previous step and further parallelize the computation by duplicating logic within the computation pipeline to instantiate multiple, parallel computation pipelines and fit in the Virtex-6 LX240T. In AutoPilot, function parallelism is easily explored through a combination of array partitioning and loop unrolling. However, AutoPilot does not contain a directive or pragma to explicitly denote parallelism; all parallelism is created implicitly when AutoPilot detects that computations are independent. As a result, introducing parallelism can be sensitive to AutoPilot correctly detecting independence between computations.

To create a position where we can explore functional parallelism, we define an extra array dimension on data arrays used within the inner loop. Then, we create a new inner loop and specify complete unrolling of the inner loop. Note that this seems logically identical to a partial partitioning of the data array and partial unrolling of the computation inner loop; however, this method more clearly signifies functional independence to expose the parallelism opportunity to AutoPilot. For example, in Figure 5 we show code extracted from BFAS that demonstrates a section where we can explore the amount of parallelism in the disparity computation by changing the FS_UNROLL parameter, where the [FS_UNROLL] dimension of the data arrays is the added dimension that will be unrolled completely.

5. Experimental Results

In this section, we present the results of our proposed HLS optimization process on both embedded benchmark kernels and various stereo matching algorithms. For each benchmark, we use autocc (AutoPilot's C compiler) and autosim to verify correctness of the modified code. Then, we perform HLS using AutoPilot [9] version 2010.A.4 targeting a Xilinx Virtex-6 LX240T. Then, if the AutoPilot-produced RTL can fit in the FPGA, we synthesize the RTL using Xilinx ISE 12.1. Area and clock period data are obtained from ISE after placement and routing reports. After synthesis of AutoPilot's RTL, we measure the latency in clock cycles using ModelSim simulation. Then, hardware latency in seconds is computed by multiplying the clock period by the measured clock cycles; speedup is the ratio of hardware latency to original (unmodified) software latency. The software execution is performed on an Intel i5 2.67 GHz CPU with 3 GB of RAM.

5.1. Embedded Benchmark Kernels. Recall that we perform data tiling to matrix multiplication. In our experiments, we explore different tile sizes (8×8, 16×16, 32×32, 64×64, and 128×128). More importantly, tiling allows us to perform the subsequent optimizations (Sections 4.4 and 4.5). First, tiling enables us to do a complete loop unrolling for the inner most loop computation within one tile as the loop bounds are smaller compared to the original version. Second, tiling reduces the BRAM usage for one tile, thus we can parallelize the computations of multiple tiles by instantiating multiple pipelines. The exploration results of various tile sizes are shown in Figure 6. As shown, the runtime is very sensitive to the tile size and tile size 64×64 returns the best results. This is because larger tile size limits the degree of parallelization while smaller tile size limits the unrolling factor (e.g., loop iterations) within one tile. Hence, the best tile size is somewhere in between.

For embedded benchmark kernels, most of the arrays can be completely partitioned for better memory bandwidth because the arrays used by the kernels are relatively small and there are relatively few arrays in total. We observe that there are few opportunities to perform reduced bit-width optimization for the embedded benchmark kernels because the bit-width has typically been optimized already. The performance improvement is mainly from the last two optimization steps as multiple RTL blocks are allowed to execute in parallel through either loop pipelining or parallelization.

With a tile size of 64×64, there are 256 total tiles for an array size of 1024×1024. For the hardware design, we can fit 16 computation pipelines in the FPGA, therefore, we can execute 16 tiles in parallel. For each computation tile, we completely unroll the innermost loop. This solution gives us the best performance in our experiments. Blowfish and ADPCM cannot be parallelized with the resource duplication optimization due to data dependency. Blowfish and adpcm process data block by block, with data processing serialized between blocks. Hence, the improvement is from the loop optimization step only—loop unrolling and pipelining. Similarly, TDES, AES, and SHA process streams of data with required sequential dependence between neighbouring elements in the data stream. Thus, it precludes parallelization via duplication of computation pipelines. Fortunately, for these algorithms we can enable the array stream optimization which converts the arrays into FIFOs and then allows efficient fine-grained pipelining of the algorithm computation in the loop optimization step.

Figure 7 shows the speedup over original software for each kernel. As shown, AutoPilot produces high quality designs for all the kernels; 4X to 126X speedup is achieved. For matrix multiplication, we try two different input sizes, 1024×1024 and 512×512. For matrix multiplication, additional speedup is achieved at the parallelization step as multiple tiles computation can be run in parallel. However, for the rest of the kernels, the speedup is from loop optimization step

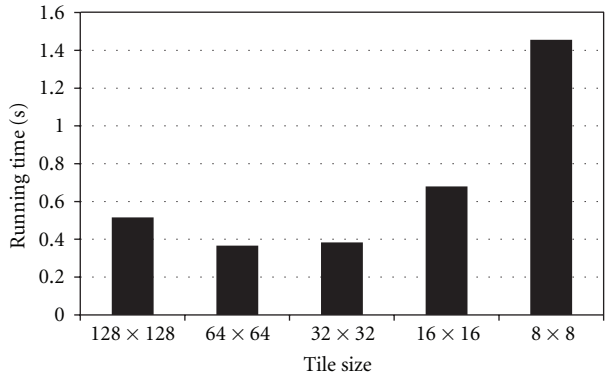


Figure 6: Comparison of different tile sizes for MM.

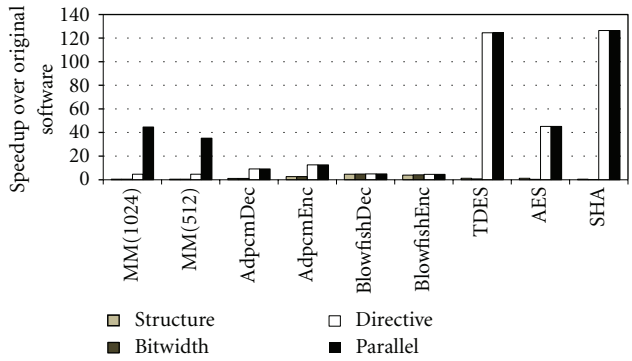


Figure 7: Speedup over original software for each kernel. Directive denotes the pipelining and loop optimization step.

only due to the sequential data dependency. We observe that TDES, AES, and SHA achieve significantly more speedup than the other kernels. Thanks to the powerful array stream optimization applied at the Reduced Bitwidth & BRAM step, we are able to do fine-grained loop and function pipelining in the loop optimization step (directive). For TDES and AES, we successfully achieve initiation interval (II) of 1. The resource utilization of the kernels are low due to their small size. The utilization for LUT is from 1% to 28%. FF utilization ranges from 1% to 20%. For all the kernels except MM, the required BRAM usage is very small because the array sizes are small and are successfully completely partitioned. MM requires 192 BRAMs for 1024×1024 size array and 160 BRAMS for 512×512 size array.

In summary, AutoPilot offers high-quality designs across a wide range of embedded benchmark kernels. Furthermore, HLS allows us to easily explore various optimizations at high level (C for AutoPilot). It only took a hardware engineer two weeks to complete the experiments for all the kernels.

5.2. Stereo Matching. Figure 9 shows the speedup over original software for each stereo matching algorithm and optimization step. Figure 8 shows the usage of various resources in combination with speedup for all the optimization steps. As shown, incremental improvement is achieved at each step.

First, for the code restructuring step, BFAS results in a 50% reduction in AutoPilot estimated LUT use, 70% fewer flip-flops, 60% fewer DSP blocks, and 95% fewer BRAMs. The other stereo matching algorithms do not employ computation resources as heavily, so there was less benefit for LUT, FF, or DSP resources, but all of them received at least 90% reduction in memory resources.

Second, for the reduced bitwidth and BRAMs step, all of the algorithms except CSBP reduce BRAM consumption by 50% to 75%. CSBP primarily uses one large array of data, so there is no gain via the use of array directives, and the data element size was already chosen well for the storage. However, for all of the stereo matching algorithms, the bitwidth optimization also reduced LUT, FF, and DSP use due to reduced size datapath computation units. Only BFAS made use of functions that were suitable for conversion into ROM tables; after ROM conversion, BFAS had an additional 8% reduction in LUT use, 5% fewer FFs, and 7% fewer DSP units than the bitwidth reduced software. The BFAS algorithm uses exponential, square root, and logarithm floating point functions, but with small input ranges; the cost of a ROM is small compared to the cost of implementing floating point or integer functional units that perform these functions.

Third, for the pipelining and loop optimization step, directive insertion is performed iteratively together with parallelization to find the best tradeoff of directives and parallelization. For the code eventually used with the next optimization step, BFAS achieved 1.5x speedup over the bitwidth and ROM step, CSBP achieved 2.5x improvement, CLM achieved 2.9x speedup, and the SO versions achieved 7.2x and 5.3x with and without occlusion, respectively.

Finally, for the parallelization step, algorithms with larger resource use have relatively little flexibility to employ resource duplication—BFAS can duplicate 4 disparity computation pipelines; CSBP can also use 4 disparity pipelines. In contrast, the SO algorithm is significantly simpler—it can allow 20 parallel pipelines with occlusion and 34 without.

Overall, all of the stereo matching algorithms achieve speedup after parallelization and resource duplication: from 3.5x to 67.9x improvement over the original software. In general, each optimization step provides some incremental improvement, but the final step shows the greatest benefit. However, this is not to mean that the other steps are not important; rather, this emphasizes the importance of minimizing resource consumption in order to allow maximum flexibility in the parallelization step.

5.3. Discussion. We have demonstrated that HLS can produce high quality designs for embedded benchmark kernels: 4X to 126X speedup. In general, parallelization and loop pipelining are two effective optimizations. For the kernels without data dependency (e.g., MM), significant speedup is achieved via parallelization and resource duplication. For the benchmarks (e,g. AES, TDES, and SHA) available for array stream optimizations, significant speedup is achieved via fine grained pipelining. The benchmarked kernels are widely used in various applications, which indicate that HLS is suitable for a wide range of applications. The high speedup

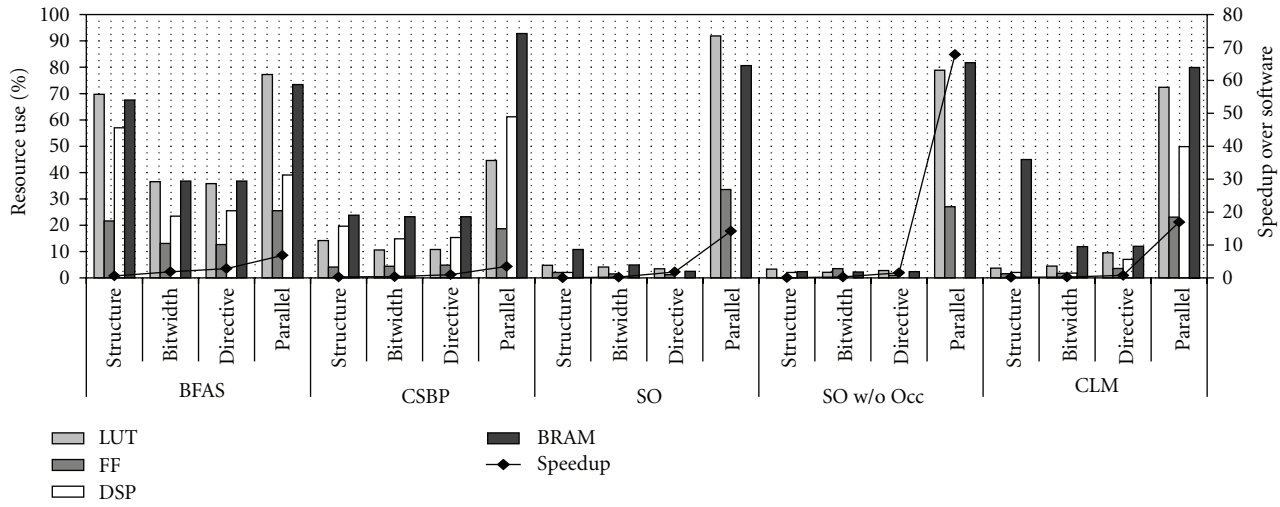


Figure 8: Resource use (left y-axis) and speedup (right y-axis) for each ISE synthesizable software version. Directive denotes the pipelining and loop optimization step.

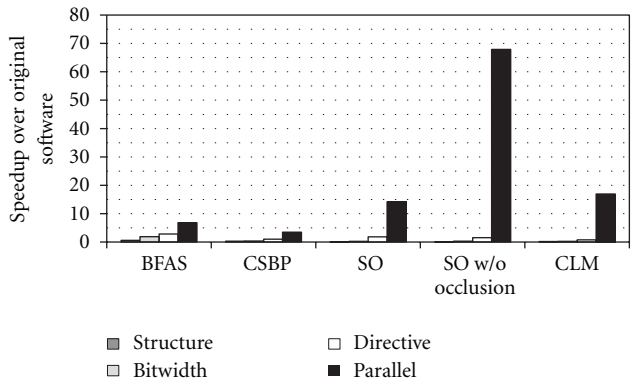


Figure 9: Speedup over original software for each algorithm for each ISE synthesizable optimization step.

also suggests that HLS can be used together with hardware software co-design tools for choosing the potential kernels for hardware acceleration.

We also perform a case study on stereo matching using HLS. Stereo matching contains multiple kernels related through data communication. Ideally, we want to convert the data arrays that are communicated through kernels into FIFOs and allow pipelining between kernels as mentioned earlier. However, this array stream optimization is only applicable to data written and read in FIFO order, which is not the case for any of the stereo matching algorithms. This certainly limits the speedup we achieve. Fortunately, for stereo matching, we can partition the image into sub-images that can independently compute disparity and depth information. The image partition allows us to process multiple sub-images in parallel. Thus, for the final solution, speedup of 3.5X to 67.9X is still achieved.

6. Observations and Insights

Through our experiments, we have demonstrated that HLS can achieve significant speedup for both embedded benchmark kernels and complex stereo matching algorithms. Also, because of the dual requirements of high-performance and fast develop cycle, HLS is an attractive platform for acceleration of these applications. More importantly, HLS synthesizes the kernels written at high level (C/C++ for AutoPilot). This allows us to easily explore various optimizations at high level. Now, we evaluate HLS in terms of the productivity, software constraints, usability, and performance of the tools in order to reach the performance we have demonstrated.

6.1. Productivity. It is important to evaluate the design effort required to achieve this level of speedup. Table 7 shows the development effort spent on embedded benchmark kernels and stereo matching algorithms, normalized to development time of a single hardware designer. All the experiments for embedded kernels are completed within 2 weeks as the kernels are relatively small. BFAS required longer than the others, as it was the first stereo matching algorithm implemented and some time was spent on learning the stereo matching problem. The manual stereo matching design presented in [3] required 4-5 months of design effort for an experienced hardware designer to implement the design, plus additional time for a team to design the algorithm. Manual implementations for other application domains quote similar development efforts [1, 2]. Thus, compared to manual design, HLS achieves a fivefold reduction in design effort.

6.2. Software Constraints. As discussed earlier, typical software constraints of HLS tools require statically declared memory, which also precludes the use of many standard libraries such as STL or OpenCV. Furthermore, there are

additional guidelines on *efficient* software for HLS. These include the following.

(i) Convert loops using `break` or data-dependent loop bounds to static loop bounds to improve pipelining/parallelization.

(ii) Use FIFO data read & write order for dataflow optimizations.

(iii) Reduce operand size to minimize storage needs.

(iv) Use array_map to reduce storage by combining arrays.

(v) Use complete array partitioning to convert arrays to scalars.

(vi) Structure and parameterize likely parallelization locations to simplify parallelism search space.

(vii) Perfectly nest loops when possible—when not possible, consider parallelism on the innermost loop.

The "best" loop to be the innermost loop is a tradeoff between multiple factors including the number of transitions between loop levels (which requires 1 cycle of latency per transition), data access order for computation, ability to unroll/parallelize, and ability to pipeline computation. These factors are sometimes conflicting (e.g., complete unrolling a small to medium size inner loop may be best, pipelining the largest loop may be best, etc.).

In many cases, these software constraints are easily achieved by software engineers familiar with optimization techniques. Although the goals of optimization are somewhat different because the code will correspond to hardware, the techniques are similar to typical optimization. However, these constraints sometimes conflict with "good" software engineering practices. Examples of "good" software practices include maximizing code reuse with heavily parameterized code; using variable loop bounds and early exit conditions to reduce worst-case paths in comparison to FIFO ordered fine-grained interleaving of computation. These constraints suggest that HLS tools may also need to improve in ability to efficiently handle some such codes. For example, AutoPilot contains a loop_tripcount directive that is used for performance analysis, but not in the synthesis process. If also used during the synthesis process to specify bounds and typical iterations on variable loops, this could allow easier optimization of such code.

6.3. Usability. AutoPilot's optimizations are very powerful—array map and array partition can have significant impact on storage requirements, and together with loop unrolling, it is possible to explore possible parallelism points quite easily. However, automatic transformations sometimes make this task more difficult; by default AutoPilot will attempt to completely unroll an inner loop to expose parallelism when pipelining, but when the loop has variable bounds, this can result in significant overhead.

AutoPilot is conservative in applying optimizations, which prevents generation of incorrect hardware. However, this also can make exposing information about code independence (for parallelism) difficult. For example, the parallelism code shown in Section 4.5 is required because anything except complete array partitioning does not guarantee that AutoPilot will assume partitioned arrays are independently accessed. Furthermore, because AutoPilot does not have a directive to explicitly denote parallelism, creating parallel hardware is sensitive to AutoPilot's ability to detect independence. This can be challenging in cases where by necessity code shares data resources, but the user knows (and could denote) that parallel function calls would not interfere.

Finally, although AutoPilot has powerful optimizations available, it is sometimes difficult to apply it to code that was not designed in advance to use the optimization. As demonstrated with the AES, TDES, and SHA benchmarks, array streaming to allow dataflow optimization is an extremely powerful optimization for data marshaling and computation overlapping, but it is only available if data is generated and used in FIFO order, which would require significant restructuring in 2D-windowed computation such as computer vision applications.

Table 7: Development effort.

Algorithm	Development Effort
BFAS	5 weeks
CSBP	3 weeks
CLM	4 weeks
SO	2.5 weeks
Embedded kernels	2 weeks

6.4. Performance. We have demonstrated that AutoPilot can achieve high speedup over the software implementation: 4X to 126X speedup for embedded benchmark kernels and 3.5X to 67.9X speedup for stereo matching algorithms. It is important to consider the performance difference between HLS and manual hardware implementations. A manual implementation of CLM achieved speedup of ~400X [3], similar in magnitude to other FPGA stereo matching solutions [43]. A GPU implementation of CLM [44] achieved 20X speedup over the original software, which is similar to the 17X speedup achieved in this work.

It is important to emphasize that this performance gap is a gap between HLS hardware produced from software *not designed for HLS* and manually produced RTL. This is not to suggest that HLS-produced hardware cannot achieve performance near manual designs, but to point out that the current abilities of HLS cannot be easily used in general software not designed for HLS. We have achieved significant speedup on some of these algorithms without significant restructuring of the original software, but a significant portion of the remaining gap is due to memory-level dependence and data marshalling. When we examine the hardware design in [3], a major portion of the performance is attained through fine-grained overlapping of computation throughout the design pipeline. Although AutoPilot has synthesis directives that *can* create this sort of hardware, the code must be designed in advance to use the correct data access order and code

structure, and that software code structure is different from typical software structure that is used in CPU source.

6.5. Future Directions. Together, this study leads to two groups of future directions for HLS tools: one to improve the usability and accessibility of currently available HLS features, and the second to improve performance gap between HLS and manual design by adding new features. Some of these observations are specific to AutoPilot's optimization and code generation flow; however, the challenges of supporting a wider range of input source code are applicable to all of the state of the art HLS tools.

6.5.1. Usability

(i) Improved loop unrolling/pipelining for complicated loops that require temporary register and/or port duplication.

(ii) Support for port duplication directives to add extra read and/or write ports to BRAM-based storage through directives rather than manual data array duplication.

(iii) Automatic tradeoff analysis of loop pipelining and unrolling.

(iv) Automatic detection and conversion for common computation structures such as tree-based reductions.

(v) Improved robustness of dataflow transformations, streaming computation for 2D access patterns.

6.5.2. Performance Gap

(i) Detection of memory level dependence across multiple, independent loops and functions, automatic interleaving of computation between the loops and functions.

(ii) Automatic memory access reordering to allow partitioning, streaming, or improved pipelining.

(iii) Automatic temporary buffers for memory access reuse.

(iv) Iteration between array optimizations, pipelining, and parallelization for efficient search of design space.

7. Conclusions

High level synthesis tools offer an important bridging technology between the performance of manual RTL hardware implementations and the development time of software. This study uses popular embedded benchmark kernels and several modern stereo matching software codes for HLS, optimizes them, and compares the performance of synthesized output as well as design effort. We present an unbiased study of the progress of HLS in usability, productivity, performance of produced design, software constraints, and commonly required code optimizations. Based on this study, we present both guidelines for algorithm implementation that will allow HLS compatibility and an effective optimization process for the algorithms. We demonstrate that with short development time, HLS-based design can achieve 4X to 126X speedup on the embedded benchmark kernels and 3.5X to 67.9X speedup on the stereo matching applications, but more in-depth, manual optimization of memory level dependence, data marshaling, and algorithmic transformations are required to achieve the larger speedups common in hand-designed RTL.

Acknowledgments

This paper is supported by the Advanced Digital Sciences Center (ADSC) under a grant from the Agency for Science, Technology, and Research of Singapore.

References

[1] J. Bodily, B. Nelson, Z. Wei, D.-J. Lee, and J. Chase, "Comparison study on implementing optical flow and digital communications on FPGAs and GPUs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 2, p. 6, 2010.

[2] C. He, A. Papakonstantinou, and D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 412–418, October 2009.

[3] L. Zhang, K. Zhang, T. S. Chang, G. Lafruit, G. K. Kuzmanov, and D. Verkest, "Real-time high-definition stereo matching on FPGA," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, 2011.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, vol. 3, pp. 174–184, September 1998.

[5] B. Bond, K. Hammil, L. Litchev, and S. Singh, "FPGA circuit synthesis of accelerator data-parallel programs," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 167–170, May 2010.

[6] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: object-oriented programming across the hardware/software boundary," in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 76–103, 2008.

[7] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: low-power high-performance computing with reconfigurable devices," *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 458–463, 2010.

[8] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: enabling efficient compilation of CUDA kernels onto FPGAs," in *Proceedings of the 7th IEEE Symposium on Application Specific Processors*, pp. 35–42, July 2009.

[9] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: a platform-based ESL synthesis system," in *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds., Springer, New York, NY, USA, 2008.

[10] Nallatech, "DIME-C," 2011, http://www.nallatech.com/Development-Tools/dime-c.html.

[11] Y Explorations, "eXCite," 2011, http://www.yxi.com/.

[12] Altium, Limited, "C-to-Hardware Compiler User Manual".

[13] "CatapultC Synthesis Datasheet".

[14] Synopsys, "Synthesizing Algorithms from MATLAB and Model-based Descriptions. Introduction to Synphony HLS White pape".

[15] Altera, "Nios II C-to-Hardware (C2H) Compiler".

[16] G. Sandberg, "The Mitrion-C Development Environment for FPGAs".

[17] Impulse Accelerated Technologies, "ImpulseC Datasheet".

[18] P. Coussy, G. Corre, P. Bomel, E. Senn, and E. Martin, "High-level synthesis under I/O timing and memory constraints," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 680–683, May 2005.

[19] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-lev l synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the International Conference on VLSI Design*, p. 461, Los Alamitos, Calif, USA, 2003.

[20] R. Domer, A. Gerstlauer, and D. Gajski, "SpecC methodology for high-level modeling," in *Proceedings of the DATC Electronic Design Processes Workshop*, 2002.

[21] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[22] D. Gajski, *NISC: The Ultimate Reconfigurable Component*, Center for Embedded Computer Systems, TR 03-28, 2003.

[23] A. Canis et al., "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 33, 2011.

[24] S. Sirowy, G. Stitt, and F. Vahid, "C is for circuits: capturing FPGA circuits as sequential code for portability," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 117–126, February 2008.

[25] G. Stitt, F. Vahid, and W. Najjar, "A code refinement methodology for performance- improved synthesis from C," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 716–723, November 2006.

[26] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[27] D. Scharstein and R. Szeliski, "Middlebury Stereo Vision Website," http://vision.middlebury.edu/stereo/.

[28] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86, March 2004.

[29] "BDTI High-Level Synthesis Tool Certification Program Results," http://www.bdti.com/Resources/BenchmarkResults/HLSTCP.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, pp. 3–14, 2001.

[31] Point Grey Research, "Point Grey Stereo Vision Cameras," http://www.ptgrey.com/products/stereo.asp.

[32] D. Scharstein, R. Szeliski, and R. Zabih, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," in *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision*, p. 0131, 2001.

[33] C. Lawrence Zitnick and T. Kanade, "A cooperative algorithm for stereo matching and occlusion detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 7, pp. 675–684, 2000.

[34] Q. Yang, L. Wang, and N. Ahuja, "A constant-space belief propagation algorithm for stereo matching," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1458–1465, June 2010.

[35] T. Meltzer, C. Yanover, and Y. Weiss, "Globally optimal solutions for energy minimization in stereo vision using reweighted belief propagation," in *Proceedings of the 10th IEEE International Conference on Computer Vision*, vol. 1, pp. 428–435, October 2005.

[36] B. M. Smith, L. Zhang, and H. Jin, "Stereo matching with non-parametric smoothness priors in feature space," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 485–492, June 2009.

[37] E. Tola, V. Lepetit, and P. Fua, "DAISY: An efficient dense descriptor applied to wide-baseline stereo," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 5, Article ID 4815264, pp. 815–830, 2010.

[38] A. S. Ogale and Y. Aloimonos, "Shape and the stereo correspondence problem," *International Journal of Computer Vision*, vol. 65, no. 3, pp. 147–162, 2005.

[39] K. J. Yoon and I. S. Kweon, "Adaptive support-weight approach for correspondence search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 650–656, 2006.

[40] K. Zhang, J. Lu, and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 7, Article ID 4811952, pp. 1073–1079, 2009.

[41] D. Min and K. Sohn, "Cost aggregation and occlusion handling with WLS in stereo matching," *IEEE Transactions on Image Processing*, vol. 17, no. 8, pp. 1431–1442, 2008.

[42] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," in *Proeedings of the 9th European Conference on Computer Vision*, 2006.

[43] S. Jin, J. Cho, X. D. Pham et al., "FPGA design and implementation of a real-time stereo vision system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 1, pp. 15–26, 2010.

[44] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool, "Real-time accurate stereo with bitwise fast voting on CUDA," in *Proceedings of the 12th International Conference on Computer Vision Workshops*, pp. 794–800, October 2009.