HARDWARE-SOFTWARE CODESIGN FOR RUN-TIME RECONFIGURABLE FPGA-BASED SYSTEMS

SIM, JOON EDWARD

NATIONAL UNIVERSITY OF SINGAPORE

JANUARY 2010

HARDWARE-SOFTWARE CODESIGN FOR RUN-TIME RECONFIGURABLE FPGA-BASED SYSTEMS

by

SIM, JOON EDWARD

(M.Eng. (Hons.), Imperial College of Science, Technology And Medicine)

(S.M. Singapore-MIT Alliance)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

SINGAPORE

JANUARY 2010

List of Publications

Publications related to this thesis:

- *Compile-time Design Space Exploration for Dynamically Reconfigurable System-on-a-Chip*. Joon Edward Sim, Tulika Mitra and Weng-Fai Wong. Invited presentation at the Optimizing Compiler Assisted SoC Assembly Workshop (**OCASA**), September 2005.

- *Defining Neighborhood Relations For Fast Spatial-Temporal Partitioning of Applications on Reconfigurable Architectures*. Joon Edward Sim, Tulika Mitra and Weng-Fai Wong. IEEE International Conference on Field-Programmable Technology (**FPT**), December 2008.

- *Optimal Placement-aware Trace-based Scheduling of Hardware Reconfigurations for FPGA Accelerators*. Joon Edward Sim, Weng-Fai Wong and Jürgen Teich. 17th IEEE Symposium on Field-Programmable Custom Computing Machines (**FCCM**), April 2009.

- *Interprocedural Placement-Aware Configuration Prefetching for FPGA-based Systems*. Joon Edward Sim, Weng-Fai Wong, Gregor Walla, Tobias Ziermann and Jürgen Teich. 18th IEEE Symposium on Field-Programmable Custom Computing Machines (**FCCM**), May 2010.

Other publications:

- *DEP: Detailed Execution Profile*. Qin Zhao, Joon Edward Sim and Weng-Fai Wong. 15th International Conference on Parallel Architectures and Compilation Techniques (**PACT**), September 2006.

- *An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization*. Huynh Phung Huynh, Joon Edward Sim and Tulika Mitra. 7th ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (**CASES**), October 2007.

- *Fast and Accurate Simulation of Biomonitoring Applications on a Wireless Body Area Network*. Kathy D. Nguyen, Ioanna Cutcutache, Saravan Sinnadurai, Shanshan Liu, Cihat Basol,

Joon Edward Sim, Linh Thi Xuan Phan, Teck Bok Tok, Lin Xu, Francis Eng Hock Tay, Tulika Mitra and Weng-Fai Wong. Proceedings of the 6th International Workshop on Wearable and Implantable Body Sensor Networks (**BSN**), June 2009

- *BSN Simulator: Optimizing Application Using System Level Simulation*. Ioanna Cutcutache, Thi Thanh Nga Dang, Wai Kay Leong, Shanshan Liu, Kathy Dang Nguyenm, Linh This Xuan Phan, Joon Edward Sim, Zhenxin Sun, Teck Bok Tok, Lin Xu, Francis Eng Hock Tay and Weng-Fai Wong. Proceedings of the 5th International Workshop on Wearable and Implantable Body Sensor Networks (**BSN**), June 2008

- *An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization*. Huynh Phung Huynh, Joon Edward Sim and Tulika Mitra. Springer Journal of Design Automation for Embedded Systems, 2009.

Acknowledgements

I would like to thank my advisor Professor Wong Weng-Fai for this support and guidance throughout the entire process of producing this thesis, including the years that were spent in preparing the research behind this document. His enthusiastic and hands-on approach towards scientific research has been and always an inspiration for my future undertakings. Especially, I appreciate his patience in the way he has guided and taught me over the years. I wish to thank the members of my thesis committee, Professor P.S. Thiagarajan and Abhik Roychoudhury for their invaluable input and discussions right from the early stage of this work. This thesis is not possible without their help.

I wish to thank Professor Tulika Mitra and Professor Jürgen Teich for giving me the opportunity to work with them in the past years. Their creative and yet careful approach to research will always be an example to me. I have derived great personal benefit from working together with them.

I would like to thank all the colleagues, both past and present, whose time in the embedded systems lab has overlapped with mine. I would like to especially thank Pan Yu, Kathy Nguyen Dang, Phan Thi Xuan Linh, Ge Zhiguo, Unmesh Dutta Bordoloi, Ankit Goel, Liang Yun, Sun Zhenxin, Wang Chundong, Qi Dawei, Chen Jie, Sivakuma Achudan, Liu Shanshan, Andrei Hagiescu and Huynh Bach Khoa. Space does not permit me to list everyone. I cannot imagine going through the PhD candidature without your company. Thanks for the foosball games, the meals and the conversations that ranged from technical discussion, politics to personal life sharing. I would especially like to thank Zhao Qin and Huynh Phung Huynh, with whom I have co-wrote some publications, for giving me the joy and opportunity to work with them.

I would like to thank my Christian friends, especially those from the community of Covenant Evangelical Free Church. Your prayers, encouragements and concerns are greatly

appreciated. I would like to especially thank Alan Cheng, Tan Gek Woo, Tan Huai Tze, Gabriel Koh, Wang Yi Tian and Maureen Ng for just being there when I need them.

My wife Liu Shuyi has been a tremendous support for me throughout the years, especially during the last stages of the thesis research. The way that she has sacrificially supported me and loved me has been my greatest motivation.

Last but not least, I thank my God Jesus, who died for me on a shameful cross even when I do not deserve it.

Contents

List of Figures

List of Tables

Abstract

Run-time reconfigurable FPGA-based systems create both opportunities and challenges for hardware-software codesign. On the one hand, it has been shown that significant speedups could be obtained for computations when performed on the reconfigurable hardware fabric and this potential speedup can be achieved without re-fabrication costs. On the other hand, the virtualization of the hardware resources comes at a price. Hardware computation modules have to be pre-loaded onto the FPGA prior to execution and the time taken to preload these modules can be significant. In order to obtain quality solutions for implementing applications on these platforms, we need to navigate the trade-off between the speedup achievable for individual components and the reconfiguration costs required to load them.

Envisioning that run-time reconfigurable computing will be a major part of mainstream computing, this thesis studies and proposes methodologies that can be incorporated into the design process of single, sequential programs written in high-level programming language (e.g. Java, C etc) for reconfigurable computing platforms. This thesis makes the following contributions.

First, we propose a novel design-space search framework for hardware-software partitioning of a single, sequential program. A key feature of this framework is that it facilitates the efficient computation of reconfiguration costs. Our definition of neighboring relationships between design points, when coupled with execution traces encoded in `SEQUITUR` grammar, speeds up the process of reconfiguration cost estimation when the search moves between neighboring design points. To our knowledge, this is the first work that examined the problem of implementing neighborhood searches of both the temporal and spatial partitioning space. Our experiments show that searches can be speeded up by up to 2 orders of magnitude when all the key features of our framework are employed.

Second, although the design-space search framework allows efficient computation of reconfiguration costs, it has been assumed that the reconfiguration costs cannot be hidden through techniques such as configuration prefetching that can occur in parallel with computation. In the second part of the thesis, we propose a novel, polynomial time algorithm that examines an execution trace and schedules placement-aware configurations to minimize overall execution time. This algorithm is provably optimal and our experiments show a speedup of up to 40% when compared with schedules done by online scheduling algorithms that relies on hardware predictors.

Finally, we visit the problem of inserting configuration prefetching calls into an Interprocedural Control-Flow Graph statically. While the algorithm described above yields optimal schedules, the schedules that it produces are very specific for particular input execution traces. Through the usage of profiled execution frequencies of control-flow edges, our proposed algorithm estimates placement-aware probabilities of reaching hardware execution for each basic block. The prefetches are then generated based on these probabilities. Experiments show that our proposed algorithm makes significant improvements over the state-of-the-art prefetching strategies that do not consider placement conflicts.

Chapter 1

Introduction

Reconfigurable computing is an alternative computing paradigm to the traditional Von Neumann model of computation. Typically, a general purpose processor (GPP) is coupled with a reconfigurable hardware for the purpose of application acceleration. By the word 'reconfigurable', it is intended to refer to the feature that the hardware may be configured multiple times (e.g., either during application run-time or prior to execution) to perform different computation in a way that is analogous to different programs being loaded into memory for execution. These 'softer' hardware – sometimes termed *configware* and usually a Field-Programmable Gate Array (FPGA)– carves a middle ground between the flexibility of general processors and the high performance of traditional hardware. The reconfigurability of the hardware allows rapid modifications of the platform, decreasing the time-to-market delay and prototyping costs. Although the performance of carefully designed custom hardware (e.g., Application Specific Integrated Circuits (ASICs)) still surpass that of FPGAs, studies have shown that applications are sped up by orders of magnitude compared to running the same on a general purpose machine. Due to these advantages, reconfigurable computing is now considered a viable option, especially in embedded systems, due to increasing complexity and requirements imposed by applications.

Although reconfigurable computing has been the subject of intensive research and development during the past decade, the success of reconfigurable computing remained beset by a dearth of automatic design tools for efficient implementation of applications written in high-level languages (e.g., C, C++ etc.). In particular, most current state-of-art design tools still assume that the developers have deep understanding of both hardware and software designs and it is their responsibility to fully exploit the benefits of this approach. On the other hand, traditional hardware-software co-design techniques cannot be easily extended for such architectures. The main issue is that traditional hardware-software co-design techniques are applied to small, embedded systems where the number of applications running in the system are few and limited. Furthermore, these techniques do not consider run-time reconfiguration in general because their target platform is ASICs instead of FPGAs. Thus, the overhead of run-time reconfiguration are not considered in traditional approaches.

In this thesis, we focus on proposing novel methodologies that could be adapted into design tools for reconfigurable computing platform. These innovations in hardware-software partitioning and reconfiguration scheduling seek to exploit the run-time reconfiguration features of reconfigurable computing platforms for designs that are written in high-level languages such as C.

In this chapter, we outline the motivation and problem overview of the thesis within the context of FPGA *Run-Time Reconfiguration* (RTR) in Section 1.1. In Section 1.2, we outline the major contributions and the organization of this thesis.

1.1 Motivation and Problem Overview

In traditional hardware-software codesign, it is a usual practice to transfer a proportion of the computation of an application to hardware for the purpose of application acceleration.

Assuming that a proportion P of the computation can be transfered to the custom hardware (e.g., ASICs) and that this proportion P can be improved by a speedup of Q, Amdahl's law states that the maximum speedup S achievable is

$$S = \frac{1}{(1-P) + \frac{P}{Q}}$$

.

However, in the case of reconfigurable computing platforms, the situation is different. The support for run-time reconfiguration of the hardware implies that the resources available in that hardware can be time-multiplexed and shared by different computations. Thus, for the same silicon area, having a run-time reconfigurable unit implies that the proportion P that can be transferred onto the hardware (e.g., FPGA) can be larger than before. While this may increase the potential for greater speedup, the speedup achievable is offset by the overhead needed during run-time to configure the FPGA. By denoting this overhead as R, the maximum speedup S achievable is

$$S = \frac{1}{(1-P+R) + \frac{P}{Q}}$$

Let us consider a single, sequential program `429.mcf` that is to be executed on machine based on the abstract reconfigurable computing architectural model shown in Figure 1.1. This is an application taken from SPEC2006[58], In the architectural model that we are considering, the CPU and the FPGA co-processor share the same memory address space through a shared bus connection. The *reconfiguration manager* is responsible for configuring the FPGA with hardware modules by loading bitstreams stored in memory onto the FPGA. It should be noted that the reconfiguration manager enables loading of bitstreams to be done in parallel with CPU execution. Although details may differ in actual implementations of reconfigurable architectures (see Chapter 2 for an overview of reconfigurable

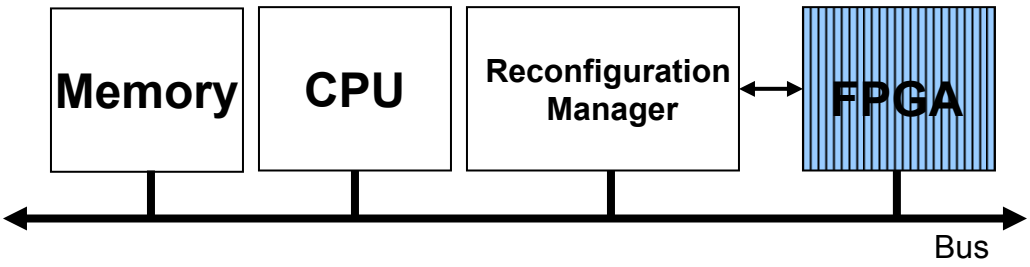


Figure 1.1: Abstract model of reconfigurable architectures.

architectures), this abstract model is fairly representative of the architectures that we are interested in for application acceleration.

Table 1.1 shows the top most 8 computationally-intensive loops for `429.mcf` and the respective proportion of the total computation time taken up by them[1] . The loops are indexed alphabetically and the loop names are taken from the function names of the C code and the loop ids suggested by the compiler[2]. We shall use this benchmark information to illustrate the complexity of implementing an optimized, efficient program on a reconfigurable architecture.

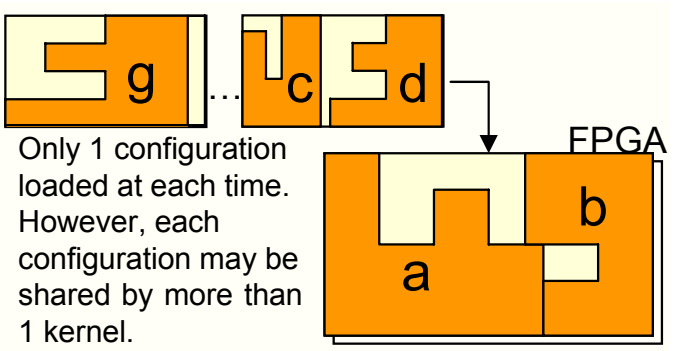
Figure 1.2(a) and 1.2(b) shows us two FPGA models: a run-time reconfigurable FPGA and a partially run-time reconfigurable FPGA. For the sake of convenience, we refer to the former as rFPGA and the latter as pFPGA for the rest of the thesis. Any references to FPGA should be considered generic and applicable to both models of FPGA. We shall now proceed to outline the factors involved when considering the implementation of (`429.mcf`) for a rFPGA and then for a pFPGA.

[1]The profile was taken and estimated by running `429.mcf` on a PowerPC machine and scaling it the number of cycles to the typical clockspeed of PowerPC embedded processors.

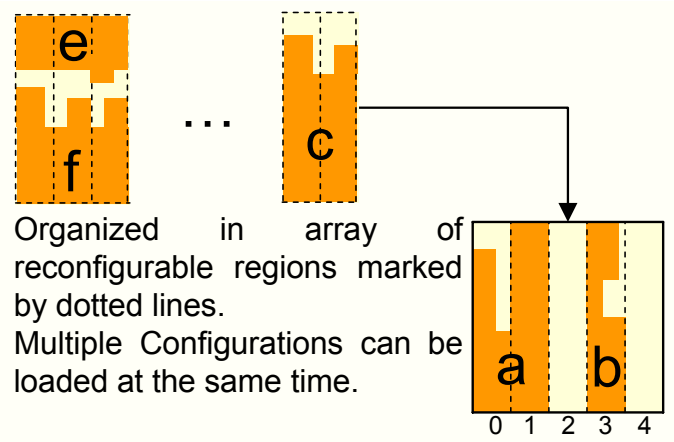
[2]It should be noted that while we have identified these loops as the computationally intensive portions that's to be implemented in hardware, computationally intensive basic blocks or even functions could be suitable candidates as well.

Index	**Loop Name (func_name-loop_id)**	**No. of Cycles (Nearest '000000 cycles)**	**Proportion of Computation (%)**
A	`primal_bea_mpp-2`	24206	0.36
B	`price_out_impl-2`	8995	0.13
C	`refresh_potential-2`	2682	0.04
D	`sort_basket-2`	2632	0.04
E	`primal_iminus-1`	1394	0.02
F	`primal_bea_mpp-4`	1337	0.02
G	`sort_basket-3`	830	0.01
H	`update_tree-1`	60	0.01
total			0.62

Table 1.1: The top 8 most computationally-intensive loops for benchmark `429.mcf`.



(a) rFPGA



(b) pFPGA

Figure 1.2: Two different FPGAs supporting runtime reconfiguration.

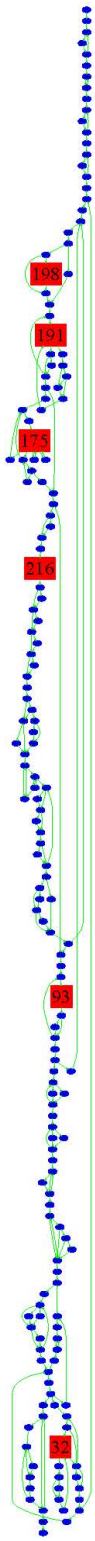


Figure 1.3: `429.mcf` An example control flow graph. Basic blocks are shown in blue. Hardware regions are shown in red.

1.1.1 Run-Time Reconfigurable FPGA

For the rFPGA, only one configuration may be loaded at any one time instance. However, the configurations may be shared by various hardware modules. As shown in Figure 1.2(a), hardware module `g` (or the hardware instance of loop `g`) occupies a single configuration while hardware instances of loops `a` and `b` share a configuration.

On a FPGA, there are both routing resources (e.g., I/O pins, interconnect switches etc), computation resources (e.g., lookup-tables, hardware multipliers) and storage resources (e.g., Block RAMs, flip-flops etc). The amount and type of resources used by a hardware module depends upon the requirements of the hardware module. For example, modules that read video input may need to use the I/O pins that are connected to the VGA inputs. Other modules may require specific hardware resources such as hardware multipliers to optimize the computation.

In the case where there are sufficient resources on the rFPGA for all the hardware modules to be loaded, the solution becomes trivial. All the computationally intensive loops shall be selected for hardware implementation. However, given the increasing complexity of applications and the practical constraints imposed by cost and size considerations, this luxury cannot be realistically enjoyed. In the light of this, the following are the factors that affect the quality of the solution:

- **The selection of a subset of the candidate kernels for hardware implementation**
  If the rFPGA resources are insufficient for all hardware modules to be loaded at the same time, we can only choose a subset of the hardware kernels for implementation. We can still choose to have all the candidate kernels to be implemented in hardware, but this entails that we need to reconfigure the rFPGA during run-time so that the FPGA resources can be *virtualized* and *time-shared*.

- **The selection of suitable candidate kernels to share a configuration** Although the rFPGA resources may be insufficient for all hardware modules to be loaded at the same time, it may have sufficient resources to hold a subset of the hardware instances of the candidate kernels. Consequently, a good solution needs to determine which candidate kernels should share a configuration.

- **The selection of one of the alternative implementations of the candidate kernels** The search for a good solution is being made more complex when one considers that each of these candidate kernels may have alternative implementations. In general, the more parallelism that is exploited, the more resources that would be needed to be employed. Thus, a good solution would need to strike a balance between resources used on the rFPGA and the speedup that may be obtained.

Therefore, a quality solution needs to consider both *temporal* and *spatial* partitioning. Temporal partitioning is the selection of configurations that would be loaded into the rFPGA during run-time (hence time-sharing or time-partitioning the rFPGA). Spatial partitioning is the selection of candidate kernels to share the rFPGA resources in one configuration (hence spatially sharing the rFPGA). When considering both spatial and temporal partitioning, a design point may fall into one of the following categories, as shown in Figure 1.4

- Static single kernel(SS): A single kernel is implemented in hardware without dynamic reconfiguration. Neither spatial nor temporal partitioning is required. In Figure 1.4, loop `a` is selected to be realized in hardware. Loop `b`, `c` and `d` are executed in software.

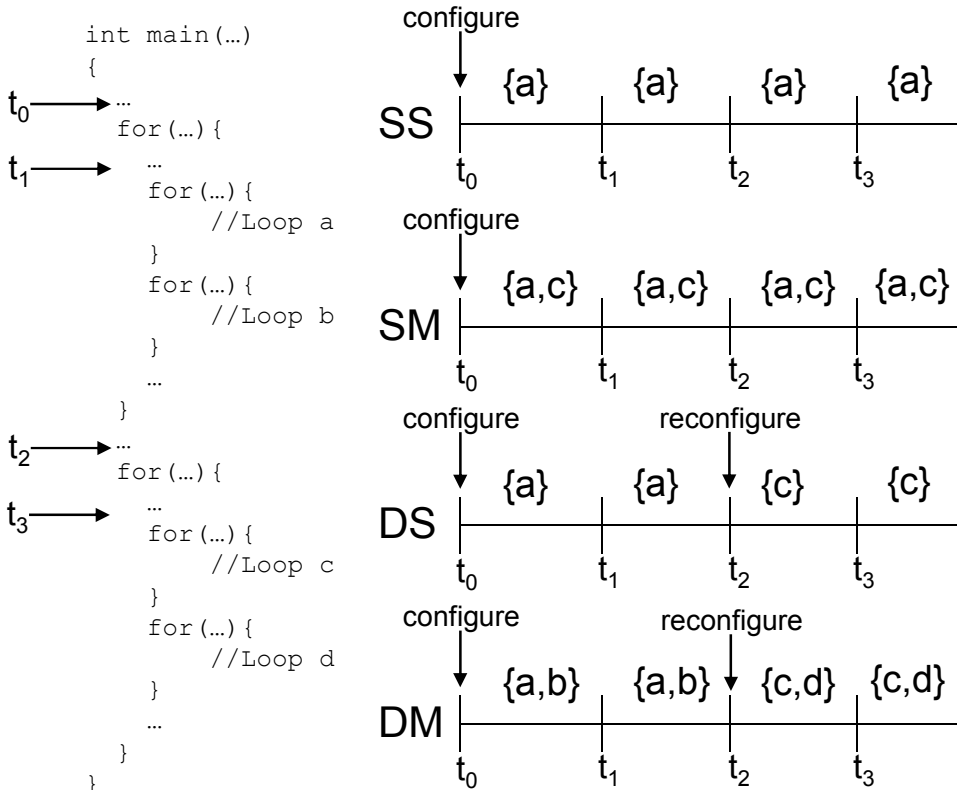


Figure 1.4: Four partitioning strategies for hardware software codesign.

- Static multi kernel(SM): Hardware is spatially partitioned among multiple kernels. However, there is no dynamic reconfiguration, i.e., no temporal partitioning. In Figure 1.4, loops `a` and `c` share the hardware. Loop `b` and `d` are executed in software.

- Dynamic single kernel (DS): Hardware is temporally partitioned such that at any point exactly one kernel occupies the entire hardware. There is no spatial partitioning in this case. In Figure 1.4 loop `a` and loop `c` occupy the hardware at time t_0 and t_2, respectively. Loop `b` and `d` are executed in software.

- Dynamic multi-kernel(DM): Hardware is both spatially and temporally partitioned. In Figure 1.4, loop `a`, `b` share the hardware between time t_0 and t_2 and they are swapped out by loops `c`, `d` at time t_2.

1.1.2 Partially Run-Time Reconfigurable FPGA

The configuration architecture of the pFPGA is organized as an single-dimension array of minimum columnar reconfigurable regions marked out by dotted lines in Figure 1.2(b). In the example shown in the diagram, there are 5 such reconfigurable regions numbered 0 to 4.

Instead of having to occupy the entire pFPGA, a configuration only needs to span across a multiple of the minimum reconfiguration regions. One particular feature of such pFPGAs is their ability to perform computation in parallel with reconfiguration under the condition that the regions being reconfigured are not the same as those performing the computations. Thus if hardware module `b` is to be replaced by loading in hardware module `c`, hardware module `a` can continue computation without interruption. This highlights one of the major differences between this and the previous configuration architecture. Namely, while

rFPGAs allow reconfiguration to be in parallel with software execution, pFPGAs allow reconfiguration to be in parallel with both software and hardware execution.

However, this added advantage comes with increased complexity to the problem. In addition to the three factors mentioned above in Section 1.1.1, a good quality solution that targets the partially RTR FOGA needs to consider the following factors as well.

- **Selection of placements of configurations** We refer to the exact location that hardware modules occupy on the pFPGA as placements. In Figure 1.2(b), module `a` occupies reconfiguration regions 0 and 1. Hence the placement of module `a` begins at 0. Placements of hardware modules (i.e., the reconfigurable region they occupy) are usually decided during design-time for 3 reasons. Firstly, if the hardware modules require specific I/O, the positions of the I/O pins constrains the region the hardware module can occupy. Secondly, since the placement of hardware modules are usually embedded within the configuration bitstream information, relocation of hardware modules during run-time could be a costly operation. Thirdly, due to the above mentioned fact that hardware modules may require specific resources, the placement of such modules are constrained to be in reconfigurable regions that contain such resources.

  Hardware modules that share the same reconfigurable regions are said to be in 'conflict' with one another(i.e., if `a` and `b` are in conflict, they cannot be loaded on the pFPGA at the same time. The rFPGA could be considered as a pathological case of having only one reconfigurable region, thus making every distinct configuration in conflict with one another. In general, a configuration for pFPGA only conflicts with a subset of other configurations. Thus, the placements of configurations affects the number of conflicts and since configurations in conflicts cannot be loaded on the pFPGA at the same time, the number of conflicts in turn influence the number

of reconfigurations that occur during run-time and hence the overall reconfiguration overhead as well.

- **Reonfiguration scheduling** As noted above, pFPGAs make it possible for reconfiguration to occur in parallel with both hardware and software execution. It is not trivial to schedule reconfiguration for a single, sequential program that is presented in a control flow graph as shown in 1.3 because the control flow of the execution changes dynamically during runtime.

In summary, in order to implement an efficient, accelerated single, sequential program for a machine based on the model shown in Figure 1.1, the factors mentioned above, especially with regards to run-time reconfiguration overhead, need to be considered. It should be noted that these factors are inter-dependent and inter-related.

1.2 Contributions and Thesis Organization

The main aim of this thesis is to study the effects of run-time reconfiguration overhead and propose new novel methodologies that address run-time reconfiguration issues that can be incorporated into the design process of applications for reconfigurable computing. Run-time reconfiguration overhead can be adversely increased by a wrong choice of candidate kernels and mis-prefetches during configuration scheduling. Consequently, a design framework of applications for reconfigurable computing platform needs to factor in these issues for application acceleration.

The main contributions of this thesis are the development of methodologies that can be incorporated into such frameworks. Specifically, this thesis makes the following contributions to the state of the art:

1. For a design space that spans both temporal and spatial partitioning, we have implemented a framework in which efficient neighborhood searches can be implemented. Through defining the neighborhood relationship between the design points carefully, the run-time reconfiguration cost can be efficiently computed when moving from one design point to its neighbor during the search. Our experiments show that the employment of this framework speeds up neighborhood searches (specifically, hill climbing and Tabu search) by up to *two orders of magnitude.*, compared to implementations of these neighborhood searches that either a) do not use the framework at all or b) use the framework in a partial manner.

2. We present a novel, polynomial time algorithm for scheduling reconfiguration given an execution trace of hardware modules that is both provably optimal and placement-aware. The algorithm includes a dependence analysis to determine whether for each instance of hardware module execution, a reconfiguration task is needed prior to its execution in hardware. A formal proof that our scheduling algorithm is optimal with respect to the application's overall execution time is also given. Our experiments demonstrate how previously proposed online scheduling algorithms fare in comparison with the optimal algorithm.

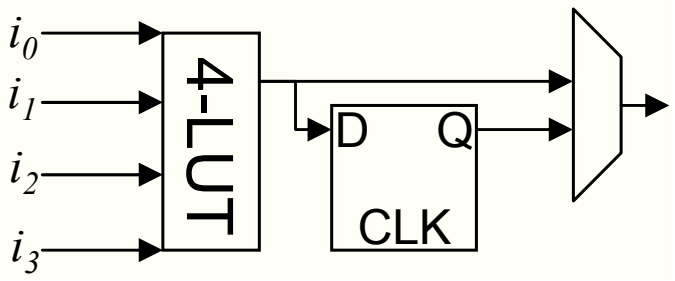
3. We present a novel static reconfiguration scheduling algorithm for programs defined in control-flow graphs. Using profiling information, we first perform an interprocedural, path-sensitive reachability analysis of the control flow graph. The analysis estimates for each basic block, the probability of reaching a hardware module without encountering conflicting configurations on the way. Our experiments show that the proposed novel algorithm performs better than current state-of-the-art algorithms across varying sets of conflicing hardware modules.

This thesis is organized as follows: In Chapter 2, we give an overview of FPGA and discuss previous and related work in reconfigurable computing, especially work in reconfigurable architectures, hardware-software partitioning and configuration prefetching; in Chapter 3, we present the neighborhood search framework for the efficient implementation of neighborhood searches of the hardware-software partitioning design space; in Chapter 4, we present MLS, a provably optimal reconfiguration scheduling algorithm for an execution trace of hardware modules; in Chapter 5, we present a static scheduling algorithm for a control-flow graph specification. Finally, we conclude the thesis by summarizing the contributions of this thesis before discussing future research directions in Chapter 6.

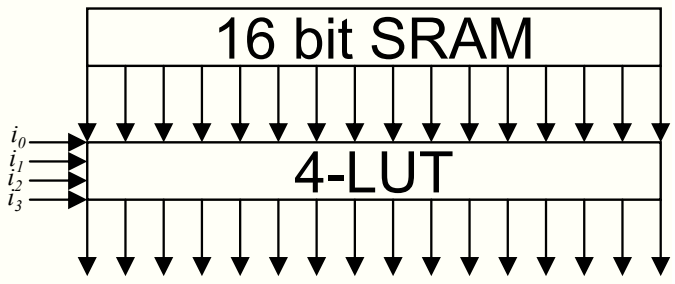
Chapter 2

Run-Time Reconfigurable Computing and Hardware-Software CoDesign

Reconfigurable computing has been the subject of intensive research for the past decade. However, research for implementing applications written control-flow specifications (or programs written in high-level languages) have often focused on synthesizing the specification for hardware implementation. In comparison, there has been fewer works that focused on hardware-software partitioning and configuration scheduling for the such applications. In this chapter, we review the background and related works of this thesis. In Section 2.1, We begin with an overview of FPGAs, especially with regards to the reconfiguration technology available in commercial chips such as Xilinx Virtex II Pro and Xilinx IV. In Section 2.2, we present a classification of reconfigurable architectures and describe some of their distinguishing features. In Section 2.3, we review some of the key works in hardware-software partitioning for run-time reconfigurable computing. In Section 2.4, we focus on the current state of the art in configuration scheduling.



(a) A simple logic block



(b) A 4-LUT can be programmed with a 16 bit SRAM

Figure 2.1: Basic computation units in modern FPGAs.

2.1 FPGA Overview

Field-programmable gate arrays(FPGA) are integrated circuits that are re-programmable after fabrication (hence *field-programmable*). In contrast with ASICs that are set in stone once fabricated, FPGAs are reprogrammable(i.e., more flexible) and therefore have a low non-recurring engineering cost. Traditionally, it has been used as a prototyping platforms for chip design, but due to increasing complexity of the requirements of applications there have been increasing attempts to use FPGAs as co-processors for program acceleration.

The main reason why FPGAs are 'programmable' is the insight that any computation can be represented as a Boolean equation. In turn, any Boolean equation can be expressed as a truth table. Lookup tables (LUTs) could be used to implement any truth tables. Using these truth tables as the basic unit of expression of a computation, complex and even

conditional statements (e.g., classic statements such as if-then-else) can be expressed by combining these LUTs to form a complex computational structure. LUTs give FPGAs the generality to implement arbitrary digital logic.

LUTs alone are insufficient if you consider the need to hold state information, especially for recursive/iterative computations that depends on the results from previous states. However, if we add a flip-flop register and a multiplexer to the LUT as shown in 2.1(a), the new circuit is enabled to hold previous state information. Figure 2.1(a) shows a typical example of a *logic block* that forms the basic unit of computation for an FPGA. Depending on the multiplexer input bit, this piece of logic will either output the previous value held in the flip-flop register or the output from the LUT directly.

This logic block can be programmed through the use of SRAMs. For example, by initializing the 16-bit SRAM in Figure 2.1(b) with appropriate values, we can implement any 4-input boolean logic. The flip-flop register and multiplexer can also be initialized with single-bit SRAMs. The size of the LUT was the subject of considerable research[56]. 4-LUTs is the usual size for modern FPGAs although the new Virtex-5 SRAM-based FPGA from Xilinx has a 6-LUT architecture. Figure 2.2 shows a typical island-style FPGA where the individual logic block 'islands' are organized in a mesh-like matrix in a 'sea' of interconnects.

2.1.1 Run-time Partial Reconfiguration of FPGAs

Xilinx has supported partial run-time reconfiguration since the Virtex II Chip[71]. Instead of reconfiguring the entire FPGA, synthesis tools provided by Xilinx are able to generate partial bitstreams that configure on parts of the chip. The smallest reconfigurable unit of the Xilinx FPGA series is called a 'frame'. In Virtex II Chips, the frames are 1-bit wide configuration data for logic cells that spanned the height of the device[73]. All hardware

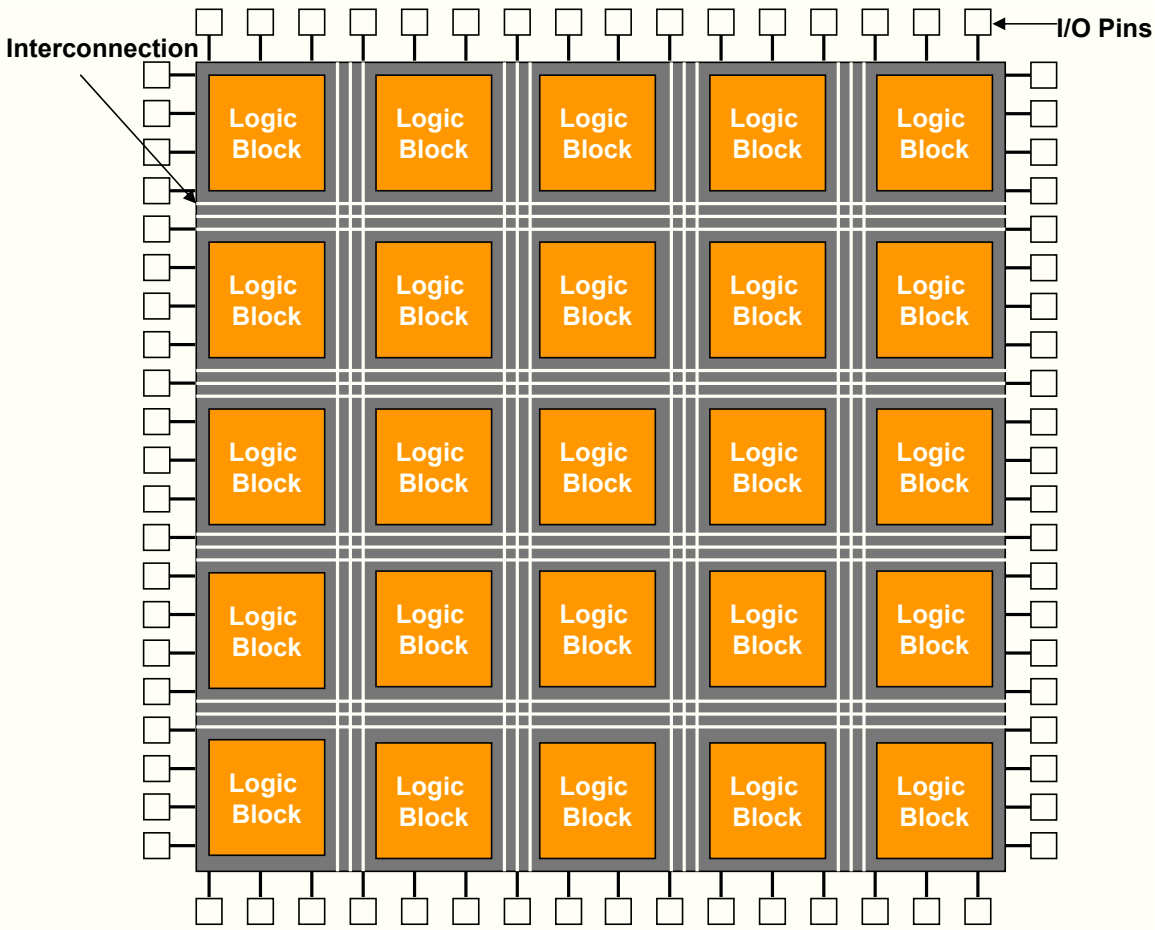
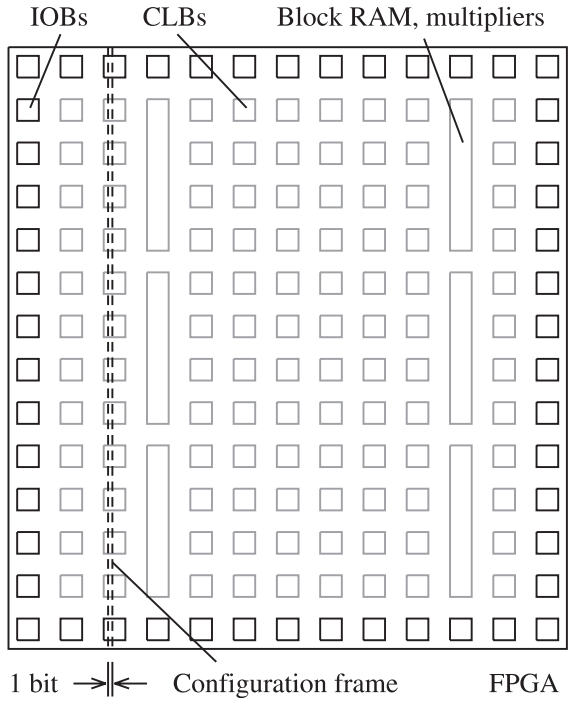


Figure 2.2: A typical island-style FPGA. The interconnect shown is an abstraction and not intended to represent realistic implementations of the FPGA.

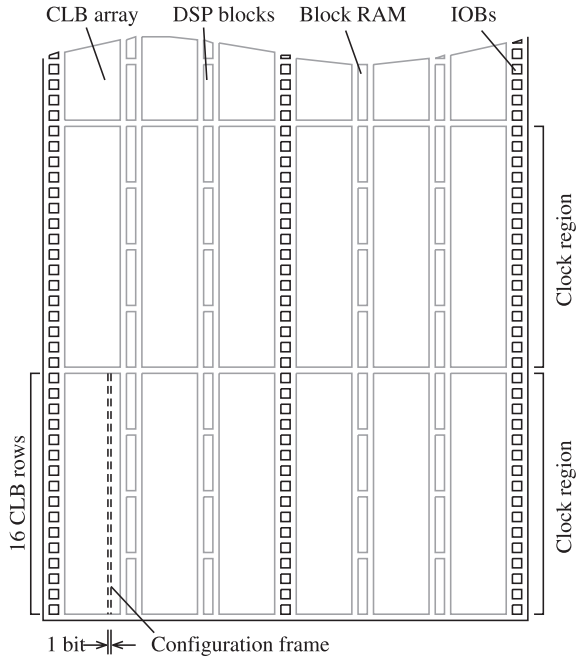
module designs must occupy at least one frame. The fact that each frame spans the height of the FPGA implies a potential inefficiency in terms of resource utilization because the hardware module design may not be placed and routed in such a way as to fill up the entire height of the device. So, even if the hardware module utilizes only half the height of the device, the entire height still needs to be configured. Starting from the Virtex 4[75] chips, each frame has a fixed height that forms the unit of the column height of the device. This means that each column constitutes a multiple of frames. For example, each column of Virtex 4 LX25 chips contains 12 frames and each frame has a fixed size of 328 bits. This configuration architecture improves resource utilization and hence the efficiency of run-time reconfiguration as well. Figure 2.3 shows the difference between the two architectures.

2.1.2 Heterogeneous Processing Elements

Figure 2.2 shows a basic architecture of FPGAs that is a two-dimensional array of homogeneous logic blocks and this is reflective of traditional FPGAs. However, modern FPGAs are increasingly heterogeneous. Besides the traditional LUTs, hardware multipliers, BRAMs(in Virtex II series) and even hardcore PowerPCs (in Virtex II Pro chips [70]) have been embedded into the design of modern FPGAs. The current Virtex 6 [76] chips contain DSP processors within its core. Recent research[28] have studied the advantage of embedded floating point cores within the FPGA. All these show that the trend of having heterogeneous processing elements is here to stay. These developments provide a greater challenge for the placement problem described in chapter 1.



(a) Virtex-II Configuration Architecture



(b) Virtex-4 Configuration Architecture

Figure 2.3: Different configuration architectures of Xilinx FPGAs.

2.2 Reconfigurable Architectures

A reconfigurable system is loosely defined as any architecture in which a general processor is augmented with reconfigurable fabric. Kiran[5] identified a wide spectrum of architectures which may be termed as such. The architectural details will have an impact upon hardware software codesign for the system. For example, loose coupling between the processor and the reconfigurable hardware will imply communication overhead that needs to be taken into consideration when designing applications for the system. The following aspects clarifies the distinctions between the various architectures:

2.2.1 Types of Coupling with Host

In most reconfigurable systems, the processor is known as the host. There is a master-slave relationship between the processor and the attached reconfigurable logic. The host is expected to configure the programmable logic with the appropriate configuration before initiating computation. The host is also responsible for marshaling the proper inputs and reading back outputs produced by the configured computation. The communication between the host and the logic is a potential bottleneck. Roughly speaking, these are the classifications of various type of coupling (following the classifications given in [5]):

- *Loose Coupling through I/O peripherals* A typical example of a system that employs this kind of coupling would be the situation where a FPGA board is attached on a the CPU, communicating with the CPU through the PCI bus interface. The communication overhead between the host and the FPGA is high. Also, data access is expensive for the reconfigurable logic since it does not share the same memory interface as the host processor. As such, the granularity of computation which can be put into the

hardware must be large because the communication overhead would be too costly to cover any speedups that may be gained.

- *Loose Coupling on a Chip Level* An example of this would be PRISM-1[2] that consists of a 10 Mhz M68010 processor directly linked to four Xilinx 3090 FPGAs through a direct 16-bit bus connection. Although this is an improvement upon the previous type of coupling, PRISM[2] reported a communication overhead of 48 to 72 processor cycles to transfer input arguments, which is significant.

- *Tight Coupling on Single-Chip* There has been considerable academic and commercial research efforts that produced various architectures where the host and the reconfigurable fabric are placed on the same chip, effectively reducing the communication overhead to the minimum. This is part of the trend of that moved towards the 'System on a Chip'. A trend that was encouraged by increasing silicon density predicted by Moore's law. GARP[27], Chimaera[77], AEPIC[62], eMIPs[51] and MOLEN[66] are examples of such systems. There is no doubt that the advent of these reconfigurable systems which promise potential performance gain creates the need for the design and implementation of new hardware-software partitioning algorithms.

- *Multi-cores including Reconfigurable Fabric* This is an extension of the previous category where traditionally, a single general processor is tightly coupled with one reconfigurable co-processor on the same chip. However, silicon technology has developed to a stage where it is possible for multiple CPU cores to be placed in the same chip. One of these cores could be a reconfigurable FPGA. The Intel FSB-FPGA[33] with FPGA accelerators pluggeed into Intel Xeon processor sockets is a prime example of this recent development.

2.2.2 Interface with Reconfigurable Logic

The way the hosts interfaces with the reconfigurable logic depends on the way the reconfigurable logic is coupled with the processor. In general, a reconfigurable logic can be placed on the datapath of the general processor and serves as a reconfigurable functional unit (RFU) or it can be a co-processor akin to dual cores architectures. Chimaera[77] and Stretch[1] are examples of the former while GARP[27] and AEPIC[62] are examples of the latter.

In order to create new custom instructions for architectures like Chimaera and Stretch, data-flow graphs are constructed and patterns of subgraph are mapped into custom instructions. The RFUs that execute these instructions usually do not access memory and impose a restriction on the number of inputs and outputs. The execution of the RFUs is usually transparent to the CPU. However, the ISA design for these architectures must accomodate space for opcode extension. As a result, there is a limit to the number of custom instructions that can be supported by such systems.

By contrast, the interface between the CPU and the reconfigurable logic for architectures such as GARP and AEPIC resembles function call semantics. As the reconfigurable logic is a co-processor that shares the same bus as the CPU instead of being placed on the datapath, the communication overhead when executing the reconfigurable logic is higher. However, the advantage is that larger granularity of computation can be accelerated and the system places no limits on the number of hardware accelerators supported.

2.2.3 Reconfiguration Latency Hiding

As mentioned in the previous chapter, reconfiguration latency is a potential performance bottleneck in these systems. Some architectures reduce the reconfiguration latency by sup-

porting multiple configuration contexts in its configurable hardware. AEPIC's[62] reconfigurable hardware is called Multiple-context Reconfigurable Logic Array (MRLA), which consists of a two-dimensional array of programmable logic and interconnection blocks, collectively known as programmable elements. Through a configuration store being associated with each programmable logic, the entire programmable hardware is effectively an array of FPGAs. When presented with a *context_id*, the various programmable elements will employ the associated configuration indexed by *context_id* into the configuration store. Chimaera[77] reports a similar structure, supporting multiple contexts for its Reconfigurable Functional Unit (RFU). These hardware features are further supported by the presence of configuration caches in both cases. That is, when a particular configuration is used and not present in the configuration store, the desired configuration may be swiftly fetched from the configuration cache, avoiding a full reconfiguration of the hardware. This scheme can be further improved by having a design of hierarchical configuration memories analogous to the memory hierarchy's in traditional computer architectures. On the other hand, not all reconfigurable architectures support and supply multiple contexts through the above mentioned method. GARP[7] reduces the reconfiguration overhead by having a wide data path between the GARP's array and memory coupled with the employment of its configuration cache.

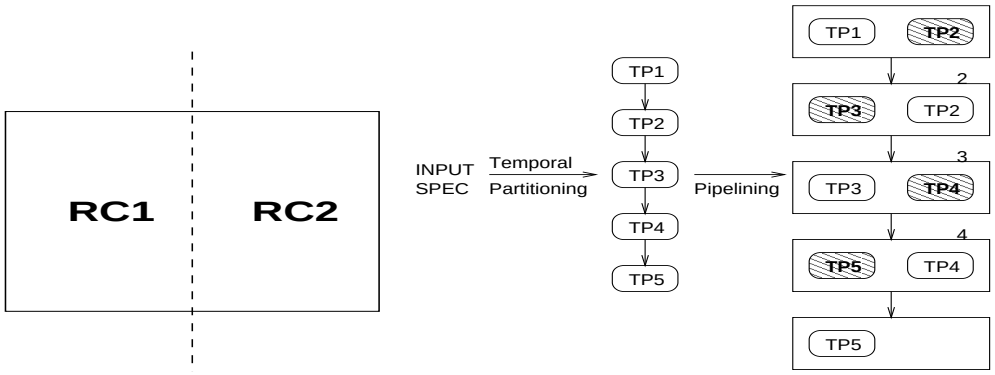
Many of the surveyed reconfigurable architectures are attempts to build better silicon devices for reconfigurable computing. Unfortunately, while many innovations and advances have been proposed, as of now, almost none of these systems are commercially viable. For this reason, our thesis has focused on developing new codesign methodologies that targets systems coupled with FPGAs, especially from the Xilinx family of FPGA chips.

2.3 Hardware Software Partitioning

The subject of hardware-software partitioning[34, 16] has been extensively researched over the years. Stated informally, the problem of hardware software partitioning is to find a suitable designation of the various portions of an application to either hardware or software implementation so that certain performance metrics (e.g., minimize overall execution time, energy savings) can be achieved. In the early 90s, the hardware-software partitioning problem was being solved for traditional architectures (i.e., CPU with ASIC). While it may be possible extend the standard techniques proposed then applied to these traditional architectures, run-time reconfigurable architectures present new challenges to an old problem. As we have indicated so far in this thesis, minimizing reconfiguration overhead is critical for obtaining a quality solution. However, as we have tried to show in the thesis thus far, this overhead is in turn influenced by factors such as latency hiding and the heterogeneity of resources available on the FPGA. There have been several works on hardware-software partitioning [17, 40, 10, 22, 46, 60, 52, 3] over the years. However, most of these works do not consider both temporal and spatial partitioning. In the following, we offer brief descriptions of certain notable and related projects.

Nimble

The Nimble compiler[40] is built on top of the garpcc[6] to address hardware-software partitioning issues not dealt with in garpcc. Given a set of loops within the program code, the Nimble compiler seeks to choose between the loops for implementation in hardware through a cost function estimate. This work did not exploit either partial reconfiguration nor pre-fetching. Rather, it heuristically tried to identify loops which may compete for the hardware through the loop-procedure hierarchy graph. After clustering such loops which may be in contention, the compiler selects loops to be moved into hardware so that overall



(a) Division of the hardware resource (b) Pipelining the application into temporal segments

Figure 2.4: Using pipelining to reduce reconfiguration costs.

performance may be optimized, having estimated the reconfiguration costs. The Nimble compiler is bench-marked against a locally optimal algorithm and an idealized performance upper bound for the benchmarks. Experiments have shown that the Nimble compiler gives near optimal solution when compared with the performance upper bound.

Physically-aware Hardware-Software Partitioning

Banerjee et al. [3] have focused solving a co-partitioning and scheduling problem for task graphs. The proposed algorithm determines for every task, whether it is implemented in hardware and if so, when to configure the task and when to execute it so as to minimize the the entire schedule. The main motivation for their work is the fact that existing scheduling algorithms may suffer from producing so-called 'optimal' schedules that are physically unrealizable due to placement constraints. Figure 2.5 shows a schedule where T_1 and T_3 occupies one column each while T_2 occupies two columns, assuming that the resource contains only 4 columns. Although by the time $t2$, two columns are free to be reconfigured, yet T_4 which requires two contiguous columns may not be configured. The point is that

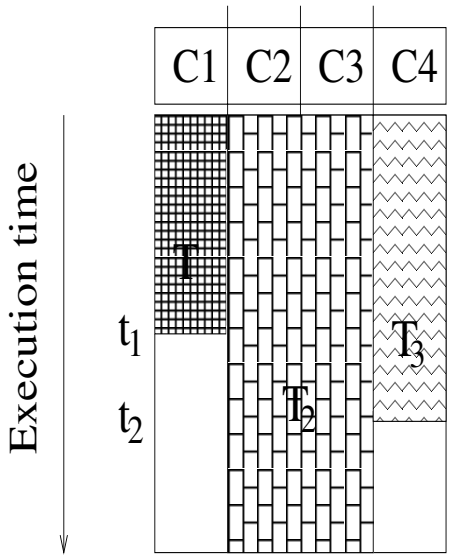


Figure 2.5: An example of infeasible placement.

algorithms that do not consider physical placement of hardware may end up with physically unrealizable reconfiguration schedules. Therefore Banerjee's work has focused on tackling this issue by incorporating hardware placement into the partitioning algorithm, using an extension of the KLFM heuristic proposed by Vahid et al.[65]

Temporal Partitioning with Pipelining

Vemuri et al. [22] integrate partial reconfiguration with temporal partitioning to improve overall performance. The key to their technique lies in dividing the pFPGA into 2 specific reconfigurable regions, as shown in Figure 2.4(a). By pipelining the application into temporal partitions(TP) as shown in Figure 2.4(b), we can overlap computation with reconfiguration time to reduce the reconfiguration overhead. For example, the execution of TP1 overlaps with the configuration of TP2 on the hardware. The reconfiguration overheaed is incurred only when the execution time of the temporal partitions is less than the reconfiguration delay.

Iterative Approach

Huynh et al.[30] have focused on the problem of performing both spatial and temporal partitioning for run-time reconfigurable custom instructions. For a set of candidate custom instructions of size N, they seek to find a suitable subset for hardware implementation while reducing the run-time reconfiguration ovehead at the same time. The proposed iterative algorithm relied on the the insight that the optimal solution could choose to implement n custom instructions, where $0 \leq n \leq N$. This is the only work we know of that performs both spatial and hardware partitioning for control-flow specified applications.

2.4 Configuration Scheduling

Techniques to reduce the reconfiguration overhead include *configuration compression* [41, 35], *configuration caching* [41], and *configuration prefetching* [26, 42, 11, 18]. While configuration compression is supported in commercial FPGAs, the chips proposed that support configuration caching have yet to be fabricated. We shall focus on the configuration prefetching in this section.

Although configurations can be loaded as and when they are requested, a significant overhead is incurred when the entire execution stalls while waiting for the loading to complete. Instead, configurations can be requested to be loaded ahead of time in anticipation of their usage. This process is called *prefetching* and as indicated in chapter 1, is done so that reconfiguration can occur in parallel with application execution. Although this is an effective method for reducing the reconfiguration overhead, one must be careful of misprefetches that may result in evicting hardware modules that are executed later.

Early work[26, 63] has focused on prefetching for reconfigurable fabrics that can only hold one configuration at a time. Later works have targeted platforms with multi-context

FPGAs[54, 46, 47, 23]. More recently, research[59, 29] has begun to focus on scheduling configurations at the level of operating systems scheduling. Prefetching technique is also being employed for High-performance Computing research[15] that attaches a supercomputer with a reconfigurable device. Solutions proposed may be divided into 2 categories - online (or run-time) and offline approaches.

2.4.1 Online Scheduling

Online scheduling requires hardware support and additional storage to keep track of historical information and monitor the dynamic state of execution. The advantage of online approaches is that these solutions are both highly adaptive and do not require access to application's source code for implementation. Noguera[46, 47] technique relies on an event window by which the system anticipates the next hardware event to be configured. Their work targets a special hardware architecture with multiple homogeneous rFPGA devices and the method depended on prefetching for the highest priority task within the event window as new incoming tasks occur. Fu[21] proposed another window-based scheduler that uses a multi-constraint knapsack approach to select configurations with best speedup for the next window of time. Li[42] and Chen[11] proposed a hardware-based predictor based on building a Markov-chain and a least-mean square model during run-time. Banerjee et al.[4] proposed an online heuristic scheduling for a linear sequence of task (a task chain) that takes into consideration the bandwidth required by the tasks as an added constraints. Although this work targets pFPGAs, they do not prefetch tasks out of order because the focus is upon scheduling the tasks to achieve maximum data parallelism. Resano[54] proposed a hybrid scheme that schedules prefetches for multiple dynamically reconfigurable hardwares but their approach does not consider a Xilinx-like pFPGAs. This thesis shows in Chapter 4 that the state-of-the-art online prefetching techniques are still considerably sub-

optimal. We expect online scheduling to be helpful in particular because reconfigurable systems are moving into mainstream computing.

2.4.2 Offline Scheduling

Offline scheduling can be classified in two categories. The first takes in a task graph as a specification while the latter takes in a Control-Flow Graph as a specification for the input application. This difference is not trivial because the task graph is usually a DAG in which the edges are merely precedence constraints (i.e., successive tasks can immediately be fired off once precedence conditions are satisfied) while the control-flow graph is a directed graph that contains cycles and the executed control-flow is conditional upon input and run-time information. Furthermore, while task graphs are usually at most hundreds in terms of size, the number of basic blocks on control-flow graphs can be tens of thousands for small applications and the execution traces obtained could be in the order of millions. Immediately, offline scheduling of task graphs that uses ILP formulation[18, 53] could not be applied to control-flow graph specifications and scalability becomes an intractable issue for such approaches. There have been relatively less research on configuration scheduling for control-flow graphs. The control-flow graph prefetching problem is usually formulated as an instruction scheduling problem[49, 26, 42] (i.e., where to insert the prefetch instruction to load configurations in advance). Offline scheduling are important for either systems that run relatively a small, static set of applications or in environments where the operating system does not support configuration management.

Chapter 3

Design Space Search for Hardware-Software Partitioning

When mapping an application that has multiple kernels onto a rFPGA, it may be necessary to share the hardware among these kernels through *spatial partitioning* [64] so as to accelerate overall execution. A reconfigurable computing architecture allows for the virtualization of hardware through *run-time reconfiguration*. In this case, the kernels can be swapped in and out of hardware at runtime. This is useful if the total area required to realize different kernels in hardware exceeds the available area. Run-time reconfigurability adds another dimension to the already complex design space exploration problem. We need to consider the *temporal partitioning* of the kernels in addition to the spatial partitioning. Moreover, the key to success is to ensure that the benefits derived from hardware acceleration of a kernel are not overwhelmed by the overhead of runtime reconfiguration [14]. Thus, this overhead should be taken into account in the partitioning decision.

For a single, sequential program, the executions of these kernels are mutually exclusive, i.e., only one kernel can execute at any point in time. The spatial partitioning problem looks at the optimal choice and placement of kernels constrained by the amount of available

hardware resource (area). The problem is further complicated by the fact that there often exist multiple instantiations of a candidate kernel. For example, applying different optimizations (e.g., loop unrolling) on a loop kernel results in a number of different hardware implementations with varying area and performance. The design space exploration needs to take all these different instances of the kernels into consideration in order to obtain an optimal solution.

Neighborhood searches such as GRASP[19] and Tabu[24] have been used to solve complex combinatorial problems effectively. Thus one way to traverse the design space is to use some form of neighborhood search. A key insight in speeding up such searches is that all these techniques involve evaluating the neighbors of the current design point. Such evaluations are often time-consuming. In this chapter, we will propose a way of speeding up such neighborhood searches.

The rest of the chapter is organized as follows. Section3.1 states the problem formulation. Section 3.2 describes the various aspects of the framework that we proposed that supports fast evaluation of neighboring points, including the key contribution: a neighborhood relationship among design points, a method for computing reconfiguration cost. After that, we show how these various aspects put together to improve the efficiency of the neighborhood searches. In Section 3.3, we present and analyze the experimental results, where we evalute the results of our framework with Tabu and Hill-Climb Search. This is followed by a conclusion.

3.1 Problem Formulation

In this section, we formally define notions used in the description of our technique in Section 3.

3.1.1 The Design Space

The design space in the context of this chapter is defined in terms of the following parameters.

- $K_1 \ldots K_N$: Candidate kernels (loops)
- $k_{i,1} \ldots k_{i,m_i}$: Different hardware implementation instances of kernel K_i with varying area and performance
- $a(k_{i,j})$: Area required by kernel instance $k_{i,j}$
- $s(k_{i,j})$: Savings in execution time due to hardware implementation instance $k_{i,j}$ of kernel K_i over its software execution
- Loop trace indicating the run-time execution sequence of the candidate kernels
- A: Total hardware area constraint
- ρ: Time to perform one reconfiguration

The loop trace and candidate loops can be obtained through profiling [61, 40]. The savings and area estimates of alternate hardware implementations can be obtained through behavioral synthesis [57] and other methods of estimation. The details of profiling and estimation are beyond the scope of this chapter and are orthogonal to its contribution. For a particular reconfigurable hardware, we assume that ρ and A are constants.

3.1.2 Configurations and Partitions

We define a *configuration* to be a non-empty set of kernels. A *configuration instance* is a particular implementation of a configuration. A configuration instance is obtained from a

configuration by choosing particular instances corresponding to each member kernel. Using the example in Figure 1.4, a configuration can be of the form $\{K_a, K_b\}$ but configuration instance will be of the form $\{k_{a.i}, k_{b.j}\}$, where $k_{a.i}, k_{b.j}$ are hardware instances of loops `a` and `b`, respectively. The total area required by all the kernel instances in a configuration instance must not exceed the hardware area constraint. Given a configuration, selecting an optimal configuration instance is a sub-problem of the entire design-space exploration problem. Switching from one configuration to another incurs a reconfiguration cost.

A set of configurations is called a *partition*. Similarly, a *partition instance* is a particular implementation of a partition. A partition instance is obtained from a partition by choosing particular instances corresponding to each member configuration. A partition consisting of a single configuration corresponds to static configuration. This is SS when the configuration is a singleton, and SM otherwise. A partition with more than one configuration implies dynamic reconfiguration. This is DS when all the configuration are singletons, and DM otherwise. An empty partition implies that no kernel was chosen for hardware implementation. It should be noted that a chosen partition implicitly implies that the other kernels have been designated for software implementation. For a partition P, we refer the set of kernels designated for hardware implementation as $HW(P)$ and the set of the kernels designated for software implementation as $SW(P) = \{K_1, \ldots, K_N\} - HW(P)$.

We have chosen to enforce a constraint that a loop kernel can appear in at most one configuration in a partition. The reason for such a constraint is if a loop is allowed to have multiple hardware versions, then it becomes necessary to dynamically infer the context under which a particular hardware version of the loop should be loaded, which further complicates the problem.

Our approach aims to minimize the total execution time of the application by accelerating candidate loops in hardware. Therefore, we define a set of functions to compute the

savings corresponding to kernel instances, configuration instances, and partitions.

$$\begin{aligned} s(k_{i.j}) &= t_{sw}(K_i) \times n_{sw}(K_i) - o(k_{i.j}) \times n_{hw}(k_{i.j}) \\ &\quad - t_{hw}(k_{i.j}) \times n_{hw}(k_{i.j}) - t_{sw}(K_i) \times n_{sw}(k_{i.j}) \end{aligned} \tag{3.1}$$

Equation 3.1 shows the savings corresponding to a hardware implementation of a kernel. The terms on the right-hand side of the equation represent the software execution time, communication overhead, hardware execution time, and software execution time of the remainder loops, respectively.

We now define the *savings* (in execution time) for configuration and partition instances.

$$s(C) = \sum_{k_{i,j} \in C} s(k_{i,j}) \tag{3.2}$$

$$s(P) = \sum_{C \in P} s(C) - n(P) \times \rho \tag{3.3}$$

Equation 3.2 shows the savings of a configuration instance. The savings of a configuration instance is simply the sum of the savings of its member hardware kernel instances. We compute the total savings of a partition instance in Equation 3.3 by offsetting the total reconfiguration time against the total savings of the member configuration instances. $n(P)$ is the expected number of reconfigurations for partition instance P and ρ is the time to perform one reconfiguration.

PROBLEM Develop a design space framework by defining the neighborhood relationships and an efficient evaluation function to facilitate the implementation of efficient neighborhood searches that solves the above problem i.e., maximize the savings of each configuration as to achieve overall execution time minimization.

We shall now describe how our neighbors of a given current point in the design space can be evaluated over a SEQUITUR-compressed trace of loops.

3.2 Fast Evaluation of Neighboring Design Points

We consider the exploration of the design space described above using some neighborhood search scheme. One of the key components that is common among these search strategies is the evaluation of the design points within a certain neighborhood. We shall now describe a neighborhood relationship between partitions that 1) is complete in coverage of the partitioning space and 2) does not recompute unnecessarily when evaluating the neighbors of an evaluated partition. The necessary components of our techniques are:

- Loop traces encoded using `SEQUITUR` grammar
- Evaluation of a single partition (without any evaluated neighbors)
- The neighborhood relationship proper
- Evaluation of a partition's neighboring points

We shall now describe each of these.

3.2.1 Evaluating Partitions

We evaluate a partition by determining the optimal way to implement the partition. The savings of a partition instance depends on the savings of its member configuration instances and the number of reconfigurations. However, all the partition instances corresponding to a partition requires the same number of reconfigurations for a given loop trace. In the example shown in Figure 1.4, if loops `a` and `b` are put in one configuration, and `c` and `d` are put in another, there will be only one reconfiguration per iteration of the outer while loop, regardless of the instances of the loops chosen to be implemented in hardware. Therefore, *an optimal partition instance can be obtained by simply choosing optimal configuration instances.* Given this insight, we need both a method for choosing an optimal configuration

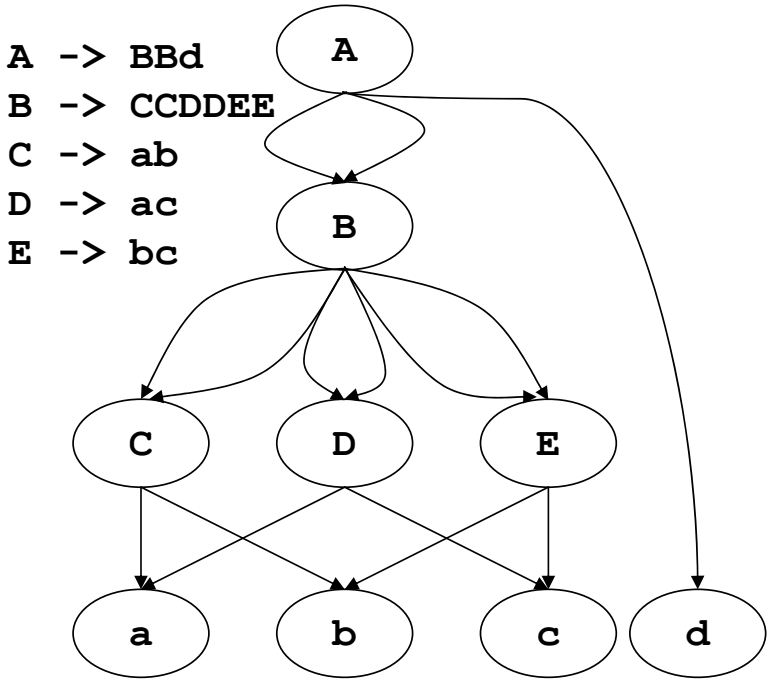


Figure 3.1: DAG representing a SEQUITUR grammar.

instance and a method for calculating the number of reconfigurations. These are described in the following subsections.

3.2.1.1 Computing Optimal Configuration Instance

Each loop kernel is associated with a number of alternative hardware implementations. A naive approach to find the optimal instance corresponding to a configuration would be to enumerate all feasible instances. However, this approach does not scale either with the number of kernels or with the number of instances corresponding to each kernel.

We handle this problem by pruning the number of instances corresponding to a kernel. We only keep the pareto-optimal instances corresponding to each kernel. Intuitively speaking, these instances are more efficient in terms of area utilization, giving better speedups with less area. After this pruning, the optimal configuration instance is found by an exhaustive enumeration of the remaining feasible configuration instances. We do not synthesize

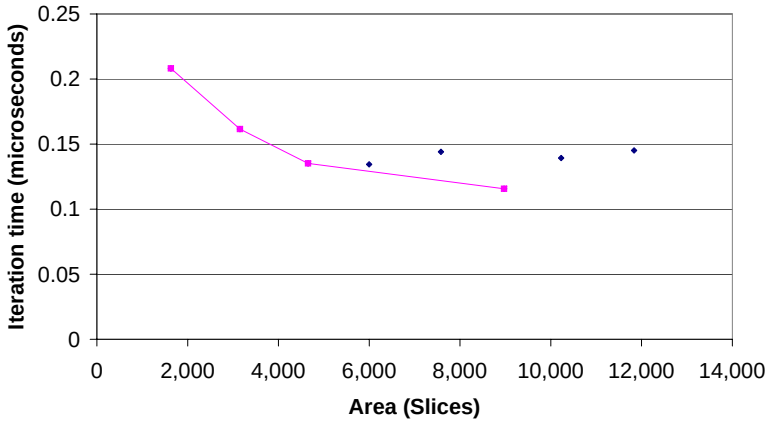


Figure 3.2: Pareto-optimal kernel instances.

the configuration instances at this stage. Rather, the savings of a particular configuration instance is estimated using Equation 3.2 along with the area requirement.

For example, Figure 3.2 shows all the instances corresponding to a loop kernel taken from the JPEG encoding benchmark. The estimated rFPGA area in terms of slices is plotted against the expected execution time of a single loop iteration for each of the kernel instances. Among the eight kernel instances, we only keep the ones on the pareto-optimal front.

3.2.1.2 Loop Trace Compression Using `SEQUITUR` Graph

We can compute the reconfiguration cost of any given partition by going through the entire trace. However, this step could be costly in terms of computation due to the size of the traces. Therefore, we compress the loop trace using `SEQUITUR`, in a format amenable for reconfiguration cost computation, as shown in the later subsections.

The `SEQUITUR` algorithm developed by Nevill-Manning [45] compresses a sequences of symbols (loop ids) by building hierarchical structures of frequently repeated sub-sequences. The `SEQUITUR` algorithm represents a finite string σ as a context free grammar. whose language is a singleton set $\{\sigma\}$. The `SEQUITUR` grammar can be represented as a directed

acyclic graph. Each leaf vertex in the DAG corresponds to a candidate loop. Each intermediate vertex in the DAG represents a sub-trace and the root vertex represents the entire loop trace. An in-order traversal of the sub-graph rooted at a vertex retrieves the corresponding sub-trace. For example, an in-order traversal of the graph shown in Figure 3.1 generates the sequence `abababacacbcbcababacacbcbcd`. It should be noted that the same vertex can be a direct sibling of itself, as shown in the figure.

3.2.1.3 Counting Reconfigurations

We can efficiently compute the number of reconfiguration of a partition through a single bottom-up traversal of the `SEQUITUR` DAG $G = (V,E)$ where V is the set of vertices and E the set of edges with complexity O(V+E). During the traversal for a particular partition, each vertex v in the DAG is labeled with the following: (1) the first and last hardware kernel in the the loop sub-trace represented by v, and (2) total number of reconfigurations for the loop sub-trace represented by v. During the same bottom-up traversal, we can compute the labels corresponding to an intermediate vertex by looking at the labels of its children as follows. Let v be an intermediate vertex with children $v_1 \ldots v_k$. Let $n(v), f(v)$, and $l(v)$ represent the number of reconfigurations, first and last configuration of vertex v. Then $n(v) = \sum_{i=1}^{k} n(v_i) - \sum_{i=1}^{k-1} x_i$, where x_i is equal to 1 if $l(v_i) = f(v_{i+1})$ and 0 otherwise. The leaf vertices would be the base case where the loop sub-trace consists of only one candidate loop corresponding to the leaf vertex. Let v be a leaf vertex. $n(v)$ would be 1 if the candidate loop has been designated for hardware, 0 otherwise. $f(v)$ and $l(v)$ would be the candidate loop if the candidate loop has been designated for hardware, null otherwise. At the end of the traversal, the label at the root vertex yields the number of reconfigurations corresponding to the entire loop trace.

3.2.2 The Neighborhood Relationship

The neighbor of a partition (in the design space) is obtained by either (1) removing a hardware kernel from any of the member configurations (removing the entire configuration if the configuration becomes empty) or (2) adding a kernel currently in software into the partition(thus designating it for hardware implementation), either into one of the existing configurations or as a new configuration containing only this new kernel.

Figure 3.3 shows a partition $\{\{a\},\{b,c\}$ with all of its neighboring partitions. The removal of kernel c from the partition gives us the neighboring partition $\{a\},\{b\}$. There are 3 ways to add kernel d into the partition. Thus, adding d gives us partitions $\{\{a\},\{b,c,d\}\}$, $\{\{a,d\},\{b,c\}\}$ and $\{\{a\},\{b,c\},\{d\}\}$. Removal of kernel b gives us partition $\{\{a\},\{c\}\}$ and removing kernel a leaves us with $\{b,c\}$. There are 6 neighbors in all. The partition $\{\{c\}\}$ cannot be $\{\{a\},\{b,c\}$'s neighbor because they differ by more than one kernel.

A partition $\{\{K_c\}\}$ cannot be P's neighbor because they differ by more than one kernel. In general, a partition P has $|SW(P)| \times (|P|+1) + |HW(P)|$ neighbors, where $HW(P)$ and $SW(P)$ are the set of hardware and software kernels for partition P, respectively. $|P|$ is the number of configurations in partition P. This relationship is complete in the sense that any partition may be constructed starting from an empty partition (by adding the kernels one by one) and the empty partition may be reached by deconstructing any partition as well (by removing the kernels one by one).

From Figure 3.3, we observe that the reconfiguration cost of the neighboring design points cannot be computed simply by adding or subtracting the number of occurrence of the kernel added or removed to the design point. For example, when kernel c is removed, the reconfiguration cost does not decrease by 2 even though c's occurrences in the loop trace is 2. In the next section, we propose a way to compute the reconfiguration cost of neighbors efficiently by making use of the `SEQUITUR` graph.

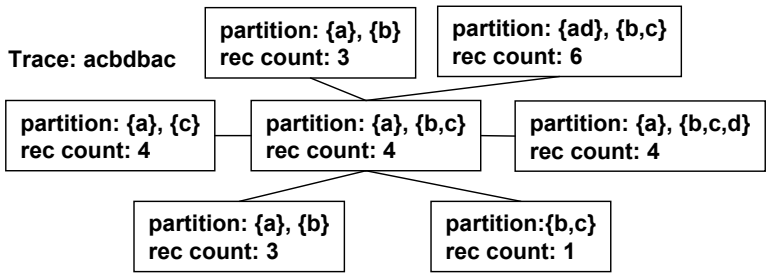


Figure 3.3: An example of a partition with its neighboring design points and the associated reconfiguration costs.

3.2.3 Simulataneous Evaluation of Neighbors

In Section 3.2.1.3, we have shown how to compute the number of reconfigurations of a partition efficiently using a compressed loop trace. However, the number of neighbors of a partition can be quite large. Therefore, traversing the `SEQUITUR` graph for each neighbor can be quite expensive. Instead, given a partition P, we propose a method to compute the reconfiguration cost of *all* its neighbors through a single bottom-up traversal of the `SEQUITUR` graph.

Our method is based on the observation that only certain sequences in the loop trace need to be considered in order to compute the reconfiguration cost of a neighboring partition. Let K be an arbitrary kernel in configuration C of partition P, i.e., $K \in C \in P$. The loop trace contains many sequences of the form of $< K_x, S, K, S', K_y >$ where $K_x, K_y \in HW(P)$ and S, S' are (possibly empty) sequences of software kernels. In each of these sequences, there are three mutually exclusive possibilities:

1. K_x or K_y is in the same configuration as K. In this case, removing K has no effect on the number of reconfigurations.

2. K_x and K_y are in the same configuration, but not in the same one as K. In this case, removing K results in the savings of two reconfigurations.

3. K_x, K_y and K are in distinct configurations. In this case, removing K results in the saving of one reconfiguration.

The effect of removing a kernel K can thus be computed after identifying all distinct sequences of the form $s =< K_x, S, K, S', K_y >$ and the number of times, $w(s)$, each of these sequences occurs in the trace. The decrease in number of reconfigurations $d(s)$ can then be computed based on the three cases above. The total savings in number of reconfigurations is $\sum_s d(s) \times w(s)$. The effect of adding kernels can be computed in a similar way.

Therefore, given a partition P, we need to enumerate all sequences of the form $< K_x, S, K_i, S', K_y >$ and their frequency for each candidate kernel K_i $(1 \leq i \leq N)$. This will allow us to compute the number of reconfigurations of a partition obtained through addition (if in software) or removal (if in hardware) of kernel K_i from partition P. This can be performed efficiently though a single bottom-up traversal of the `SEQUITUR` graph by appropriately labeling the vertices through an extension of the labeling proposed in Section 3.2.1.3.

3.2.3.1 Labeling Extension and Sequence Enumeration

We observe that these sequences $< K_x, S, K_i, S', K_y >$ will always span two consecutive sub-traces. The extreme case of these sub-traces would be one sub-trace having one loop and the other sub-trace having two loops. Given that the each vertex in the `SEQUITUR` graph represents a sub-trace, we need to label the vertices in a way that allows such sequences to be identified easily.

Consider sub-traces represented by (not necessarily distinct) vertices v_i and v_{i+1} that are next to each other in the original trace (i.e., v_i and v_{i+1} would be children of the same parent vertex direct siblings). Assume further that the sub-trace represented by v_i to be $< \ldots, K_1, S_1, K_2, S_2 >$ and the sub-trace represented by v_{i+1} to be $< S_3, K_3, S_4, K_4, \ldots >$

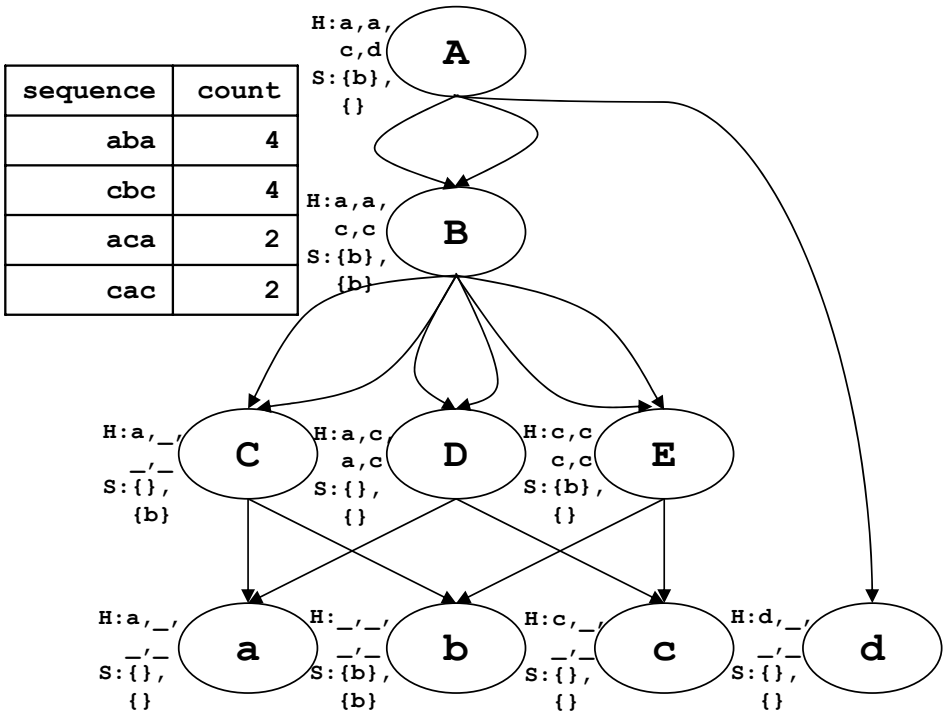


Figure 3.4: A SEQUITUR graph labeled with H and S tags given a partition that has put kernels *a*, *c* and *d* in hardware.

where $K_1, K_2, K_3, K_4 \in HW(P)$ and S_1, S_2, S_3, S_4 represents (possibly empty) sequences of software kernels. In order to enumerate the $< K_x, S, K, S', K_y >$ sequence that spans these 2 sub-traces, we need to consider two cases. If K is in hardware, then both K_2 and K_3 are candidates for K. If K is in software, then all kernels occurring in S_2 and S_3 are candidates for K. We further note that once K_i is identified, K_x and K_y can be easily identified by finding the nearest preceding and subsequent hardware kernel.

The above consideration leads to the conclusion that both the first two and the last two hardware kernels of the sub-traces are needed to identify K_x, K and K_y. Any software kernels in the sub-trace occurring before the first hardware kernel and after the last hardware kernel are also needed. Recall from section 3.2.3 that an in-order traversal of any vertex recovers a sub-trace. Thus, We label each vertex, v_i, with a H tag and an S tag, as shown in Figure 3.4, where kernels `a`, `c` and `d` have been chosen for hardware implementation.

The H tag consists of two pairs of indices The first pair would be the first two hardware kernels of the sub-trace represented by v_i. The second pair would be the last two hardware kernels in the same sub-trace. In cases when the sub-trace represented by the v_i does not contain at least 2 hardware kernels (e.g., in the case of leaf vertices), the non-existent hardware kernels would be labelled with '_'.

The S tag consists of two (possibly empty) *sets* of indices. The first set contains the software kernels that occur in the sub-trace represented by v_i *before* the first hardware kernel. The second set contains the software kernels that occur in the sub-trace represented by v_i *after* the last hardware kernel. In cases when the sub-trace does not contain any hardware kernels, all the kernels contained in the sub-trace are added to both sets.

This labeling process, i.e., computing the H and S tags, is done in a single bottom-up traversal of the `SEQUITUR` tree. Assuming that all the children vertices of v_i are properly

labelled, the H and S tags of v_i can be computed using the H and S tags of the first and last child of v_i.

With the H and S tags in hand, we can now enumerate the $< K_x, S, K_i, S', K_y >$ sequences of v_i. It turns out that this can be done in the same bottom-up traversal by examining the labels of v_i's siblings and concatenating the possible sequences. For example, according to the tags of vertex `C`, there is only one hardware kernel `a` that occurs in the sub-trace represented by vertex `C` and `b` is the only software kernel that occurs after `a`. According to the tags of vertex `D`, the first hardware kernel of the sub-trace represented by vertex `D` is `a`. Thus the the sequence <a,b,a> spans the two sub-traces represented by vertex `C` and `D`. In fact, the sequence <a,b,a> also spans the sub-traces represented by two consecutive occurrences of vertex `C`. Thus, this sequence occurs twice in the sub-trace represented by vertex `B` and since vertex `B` itself has an occurrence count of 2, the sequence <a,b,a> occurs four times in total.

With all the necessary sequences enumerated, all the neighbors of a design point can be evaluated easily based on Equation 2.

3.2.4 Employing the Entire Framework

A crucial step during a neighborhood search usually involved the following steps: evaluation of the current design point, comparison with neighboring design points and eventually selecting one particular neighboring design point to be the next step of the search. Figure 3.5 shows what happens during such a step in a search. It shows a partial view of the relevant design space, enumerated sequences and labeled `SEQUITUR` graph for 2 consecutive steps of a search. The current design point is shown in bold while the ignored design points during the step are shaded.

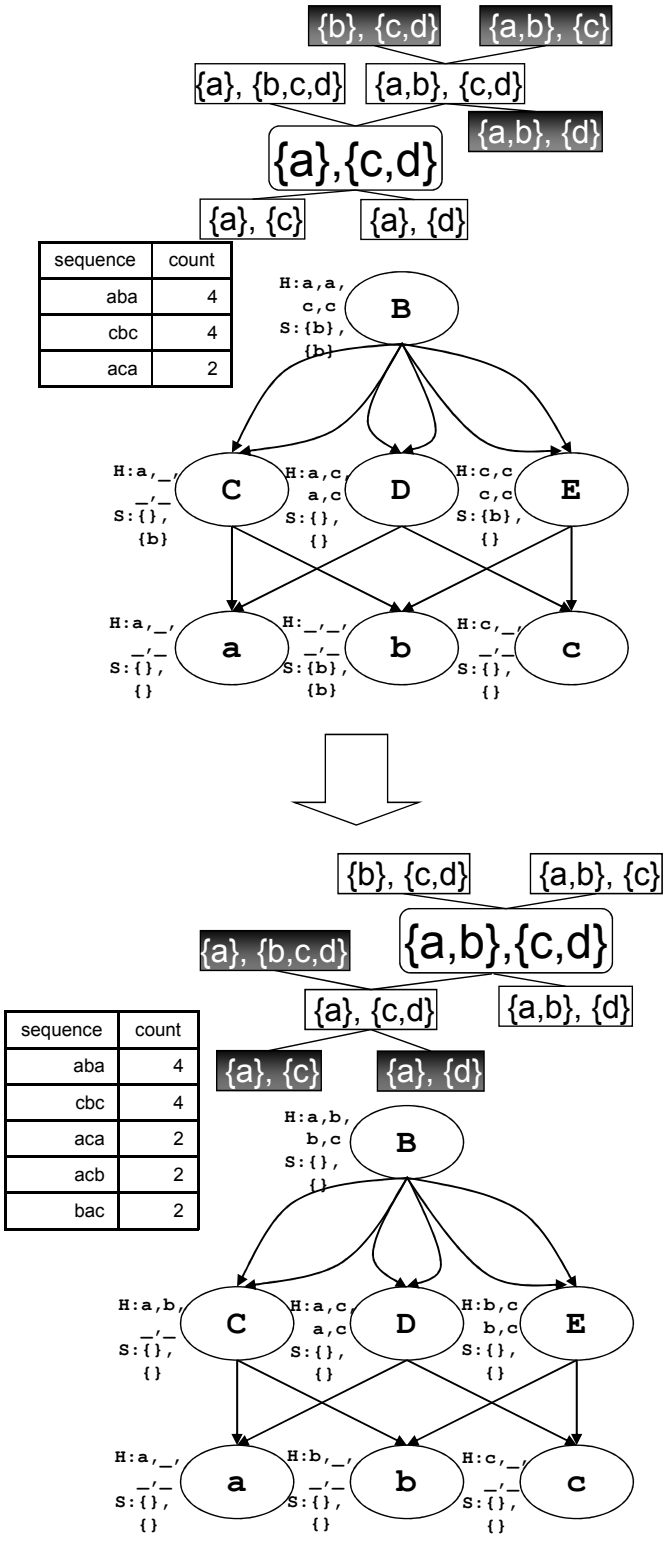


Figure 3.5: An example showing the change in annotation of the SEQUITUR graph and enumeration of sequences after a move between neighboring design points.

Initially, the current partition of the search is $\{\{a\},\{c,d\}\}$. When considering the move of adding kernel b into configuration $\{a\}$ (i.e., move to $\{\{a,b\},\{c,d\}\}$), the occurrence count of enumerated sequences $< a,b,a >$ and $< c,b,c >$ used to compute the change in the reconfiguration cost in such a move. Since kernel a and b would be in the same configuration, the increase in reconfiguration count is 4. Similar computations can be made for the other neighbors and are left as an exercise for the reader. The partition $\{\{a,b\},\{c,d\}\}$ is selected(the criterion depends on the search algorithm) for the next step in the search. Consequently, the `SEQUITUR` tree needs to be relabeled according to the methods described in section 3.2.3.1. To complete the move, sequences $< a,c,b >$ and $< b,a,c >$ are enumerated to reflect the case that kernel b is now in hardware. The search can thus continue after the move is completed.

3.3 Experimental Evaluation

3.3.1 Experimental Setup

We use four non-trivial benchmarks for our experimental evaluation: a JPEG encoder(`cjpeg`), a JPEG decoder (`djpeg`), an encryption key exchange program (`dh`), and an MPEG encoder (`mpegenc`). `cjpeg`, `djpeg` and `mpegenc` are taken from the Mediabench [38] benchmark suite while `dh` is taken from NetBench[44]. We use the Trimaran compiler infrastructure [9] to generate the input parameters for the design space exploration problem. In particular, we have implemented a loop profiler that selects a loop kernel (both inner and outer) as a candidate if its computation time exceeds more than 1% of the entire application.

In view of a lack of estimation tools, we have to pre-generate the area and timings estimation. We applied loop unrolling with various loop unroll factors to each candidate loop kernel. To obtain hardware performance and area required for each kernel instance,

we automatically generate Handel-C code [8] from Trimaran's Elcor intermediate representation. The timings and area estimations of these alternate hardware implementations are subsequently obtained through synthesis using the Celoxica DK design suite and Xilinx ISE tools. The target rFPGA for synthesis is the Xilinx 2000E model[69]. To evaluate our framework, we developed three algorithms: Exhaustive, Hill-Climb and Tabu search.

Exhaustive search In a pre-processing phase, we compute the optimal configuration instances corresponding to all possible configurations of the candidate kernels using the method described in section 3.2.1.1. The main phase then enumerates all possible partitions. The enumeration algorithm used is by Kreher and Stinson [37]. This algorithm ensures the proper enumeration of all the partitions. The savings of a partition is defined as the savings of its optimal instance. Evaluation of the savings of a partition is described in Section 3.2.1. The partition instance with the maximum savings is chosen as the optimal partition instance. It should be noted that apart from how the optimal configuration instances are chosen, the Exhaustive search algorithm does not make use of the rest of the framework.

Hill-Climb search We start with an empty partition. This ensures that our solution is at least as good as an all-software solution to begin with. We then evaluate all its neighbors using the technique described in the previous subsection. We choose the neighbor with the maximum savings (i.e., minimum execution time). The search then moves to this new design point and examines its neighbors. We always maintain the best partition obtained so far. The search terminates at a design point if we cannot find any partition in the neighborhood that is better than the current best partition. It should be noted that our Hill-Climb search draws heavily on the framework described in Section 3.2, making full use of the neighborhood relationships and the efficient evaluation of the neighbors.

Tabu search We modify the Hill-Climb search to obtain the Tabu search. The main difference being that the search does not terminate when a local maximum is reached. Instead, we maintain a tabu list of design points which have been visited and the most profitable neighbor is always visited, irregardless of whether the neighbor yields more savings than the current design point. If the particular neighbor design point is on the tabu list, the next most profitable neighbor not present on the tabu list is visited. The search terminates when the number of moves made reaches a certain limit. In our experimentation, we have fixed the number of entries on the tabu list to be 100 and the limited number of moves to be a logarithm of the design space size to base 1.05.

Benchmark	**Num. of Candidate Kernels**	**Size of Comp. Trace KBytes**	**Avg. Exhaustive Search Time (sec)**	**Avg. Hill-Climb Search Time (sec)**	**Avg. Tabu Search Time (sec)**
`cjpeg`	11	1	17719.72	0.24	3.17
`djpeg`	7	4.3	17.34	0.04	1.06
`dh`	7	72	3837.87	3.73	112.04
`mpegenc`	6	74	245.88	0.96	18.44

Table 3.1: The running times of exhaustive search, Hill-Climb algorithm and compressed trace sizes.

We have implemented the search algorithms in `C++` compiled by `gcc` version 4.1.2. We run the experiments on a 2.8 GHz Pentium 4 machine in the GNU-Linux environment. All the run-time of the search algorithms reported are based on Pentium's hardware cycle counters. The Trimaran framework allows us to define a VLIW machine with 4 integer

units, , 1 branch unit and and 1 load/store unit. We obtain the cycle-accurate measure of the all-software solution based on the simulator reports of the Trimaran framework.

Table 3.1 show the number of candidates kernels for each benchmark and the average running times of the implemented searches for all benchmarks. These values are obtained by running the experiments with varying input parameters described in section 3.3.2. This table demonstrates the infeasibility of the exhaustive search approach. The number of kernels increases the running exponentially, even though `cjpeg` has the smallest compressed trace among all the benchmarks, the running time was close to 4 hours to run the exhaustive search. Table 3.1 shows the average running time of the Hill-Climb search and Tabu search as well. It should be noted that the length of the trace dominates the running time when the number of kernels is the same. We can conclude this by observing that the running time of `dh` is longer compared to `djpeg` even though the number of kernels is the same. Our experiments show that Hill-Climb is able to find the optimal design point more than 90% of the cases while Tabu search found the optimal design point in all of our experiments.

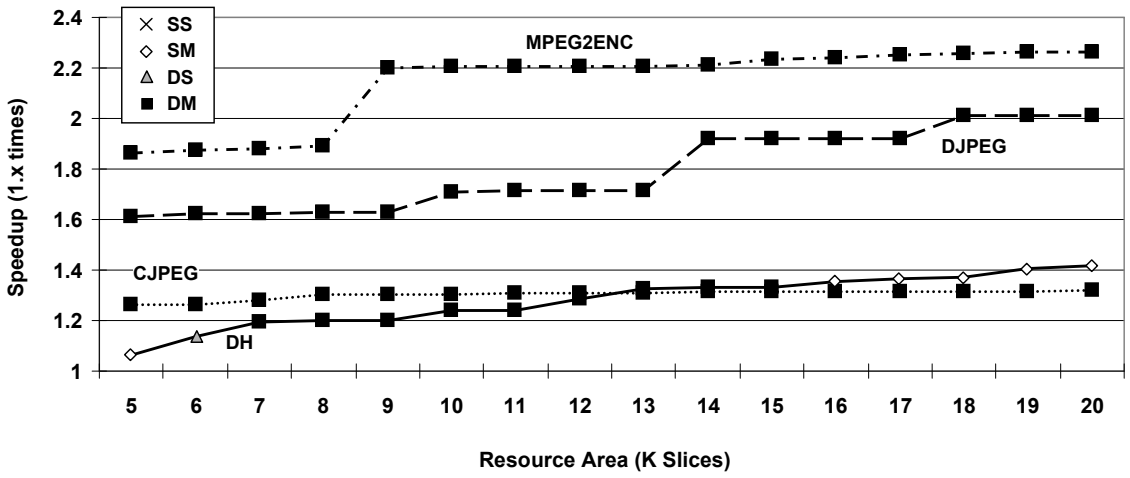


Figure 3.6: Optimal speedups plotted against increasing hardware resource.

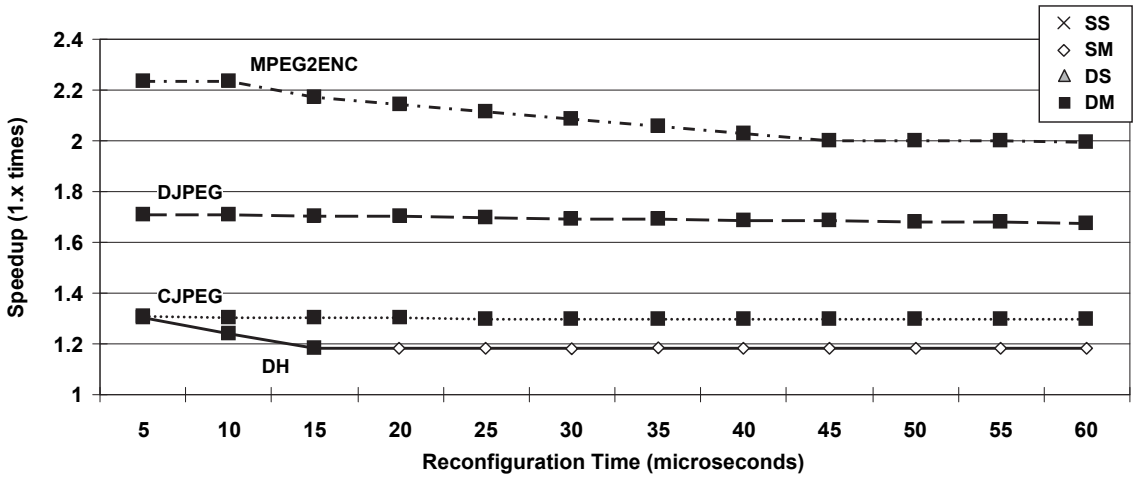


Figure 3.7: Optimal speedups plotted against increasing reconfiguration time.

3.3.2 Scaling Hardware Resources and Reconfiguration Time

Both Figure 3.6 and Figure 3.7 plots the results of exhaustive search in order to give an idea of the design space. The lines on the graphs have been labelled with the benchmark name and the plotted points with shapes to indicate the type of the solution. For example, following Figure 3.6, benchmark `dh` yields a SM partition under a resource constraint of 5K slices and then a DS partition under the resource constraint of 6K slices. Beyond, the resource constraint of 16K slices, `dh` is optimally implemented with SM. It should be noted that while many of the plotted points show DM to be the optimal partition, the kernels included in the partition are not uniformly the same for the same benchmark. Sometimes, as resource constraints increase or decrease, certain kernels has to be moved to software or hardware, though the resulting partition is still ostensibly DM.

Figure 3.6 plots the speedups of the optimal design point through exhaustive search against increasing resource area. The reconfiguration time is set at 10 μseconds. We observe that placing multiple kernels in hardware yields the optimal results in most cases except for `dh`. For the `dh` benchmark, if the resource available goes beyond 15K slices,

the SM will give better speedup than DM. This is because when the area is large enough to be shared by all the kernels, we no longer gain from dynamic reconfiguration. If the resources available decrease below 7K slices, DS and SM gives better speedup. This could be because the available resources becomes too small to hold multiple kernel. It should be mentioned that though the graph shows DM to be an optimal design point most of the time, the partition solutions for each benchmark are not the same throughout.

Figures 3.7 plot speedups of the optimal design point as reconfiguration time increases. The area is fixed at 10,000 logic slices. The speedups of `cjpeg` and `djpeg` remain almost constant because the optimal design point gives a partition which yields quite a small reconfiguration cost while achieving the speedup at the same time. As a result, the change in the reconfiguration cost is insufficient to alter the optimal design point. For the `dh` benchmark, if the reconfiguration time is small, it will still employ dynamic reconfiguration with multiple kernels. The trade-off between more kernels and reconfiguration cost comes in when the reconfiguration time increases beyond 15μseconds.

3.3.3 Impact of Using `SEQUITUR` and Label Extensions

In order to demonstrate the difference made when the `SEQUITUR` compressed trace and label extensions are used, we implemented `-trace` and `-seq` versions of the Tabu and Hill-Climb search. The `-trace` version traverses the *uncompressed* loop trace stored in memory to compute the reconfiguration cost of a design point. The `-seq` version traverses the compressed `SEQUITUR` loop trace and calculates reconfiguration cost *without* the label extensions, i.e. using the technique described in section 3.2.1.3 every time a design point is evaluated. Table 3.2 shows the various slow-downs of these implementation compared to the Tabu and Hill-Climb searches that employ *both* the `SEQUITUR` compressed trace and neighborhood relationship. The slow down is significant. Although using the `-seq`

version gives about an order of magnitude of speedup compared with the `-trace` version, employing the neighborhood relationship makes the search a further order of magnitude faster in general, except in the case of Hill-Climb search for `cjpeg`.

Benchmark	**tabu-trace**	**tabu-seq**	**hc-trace**	**hc-seq**
`cjpeg`	97.24x	10.34x	10.12x	1.13x
`djpeg`	21.77x	8.57x	6.20x	13.92x
`dh`	45.35x	11.06x	31.62x	8.33x
`mpegenc`	439.18x	18.72x	242.92x	9.29x

Table 3.2: Experiment results showing how many times Tabu and Hill Climb slowed down when *not* using SEQUITUR and neighborhood relationship.

3.4 Summary

In this chapter, we considered the problem of exploring the design space of dynamically reconfigurable SoCs for spatial and temporal partitioning . Specifically, we proposed a means of speeding up neighborhood searches of such design spaces by a novel method of estimating the design points near the current one in a compressed trace. We showed that our technique works for both Hill-Climb and Tabu search. On four benchmarks, we found that using our neighboring design point computation method, the searches were faster by up to two orders of magnitude while reporting near-optimal solution most of the time.

The framework we proposed is generic. It can be easily extended to apply to any granularity of code fragment besides loop kernels and other non-FPGA reconfigurable architecture as well.

Chapter 4

Optimal Scheduling of Hardware Reconfigurations

The design space search framework presented in Chapter 3 has assumed an rFPGA architecture where the configuration time is not reduced through configuring the FPGA in parallel with application execution. In this chapter, we consider the configuration scheduling problem for pFPGAs.

A typical architecture that we consider here is shown in Figure 1. One of the key challenges in achieving real speedups using such an architecture is that hardware reconfiguration of today's massive pFPGAs can be very costly. It often takes thousands, if not hundreds of thousands of clock cycles to reconfigure. If this high reconfiguration cost cannot be reduced, then all benefits of hardware acceleration may be lost as the application has to wait for reconfiguration to complete. *Configuration prefetching* [26] seeks to address this problem by overlapping (partial) reconfiguration with the execution of the application in pFPGA. However, a prefetch miss is costly because of the additional reconfigurations that may be needed to recover from the miss. Therefore, the scheduling of reconfiguration is crucial.

In this context, this chapter solves the following problem. Given a sequence (trace) of *actors* (an invocation of a hardware module):

- Determine whether for a given actor in the trace, it is necessary to schedule a reconfiguration task before it. This will depend on whether part or all of the resources required by the hardware module is currently being used by another (different) module. In other words, the two modules' placements overlap.

- Compute the earliest possible time a required reconfiguration task may be scheduled. For the current technology, at most one reconfiguration task is typically allowed at any time.

In essence, we will present a *polynomial-time* (in terms of the length of the trace and the number of distinct hardware modules) algorithm that schedules all the required reconfigurations such that the overall execution time (latency) of the given actor trace is provably minimized. To the best of our knowledge, this is the first time an algorithm of this nature has been proposed.

In the following, we present an overview of the contents of the chapter: Section 4.1 is a preliminary description of our architectural and scheduling model. Next, we provide an analysis of the dependencies between actor invocations into (a) data dependencies, (b) resource conflicts, and (c) reconfiguration dependencies in Section 4.2. Our scheduling algorithm is then described in Section 4.3. In the same section, we will sketch the proof of its optimality. Finally, in Section 4.4, we provide a detailed case study to illustrate the benefits of this algorithm before we conclude with an outlook of future work.

4.1 Preliminaries

4.1.1 Architecture Model

We consider an architecture with one micro-processor that receives a trace of actors (Figure 4.1). For each actor, we assume a corresponding hardware accelerator module that may be loaded into the FPGA for subsequent execution by means of partial hardware reconfiguration.

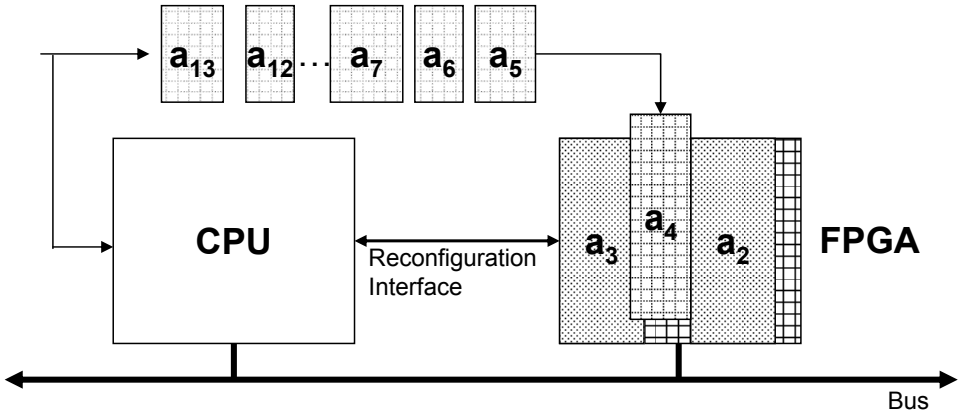


Figure 4.1: Architecture model: A CPU (left) controlling the reconfiguration interface of an FPGA (right) used as a hardware accelerator for an incoming task sequence.

4.1.2 Scheduling Model

Example 4.1.1 *Figure 4.2 shows an example of a given application consisting of a sequence of five actors (corresponding to four tasks) with data dependencies, and a given conflict relation concerning the shared use of FPGA resources. For example, when task B conflict with C, this would mean that they share some common hardware resources on the FPGA which may be either I/O pins, memory resources (such as block rams), or slices.*

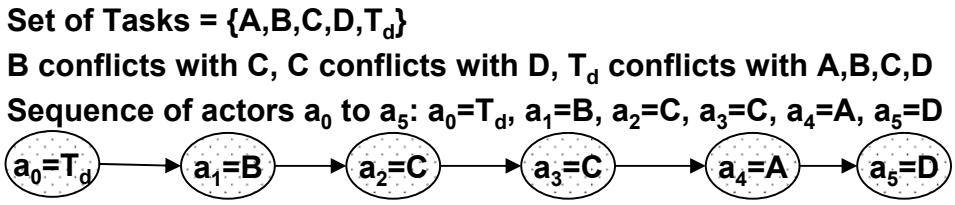


Figure 4.2: Example of actor trace.

Assume for now that this conflict relation is given statically, i.e., no module relocation is allowed. Thus we know the conflicts between every pair of actors at compile time. More formally, we define an actor trace and the corresponding conflicts as follows:

1. *Trace of actors*: $S_a = (a_0, a_1, a_2, \ldots, a_n)$ with $a_i \in T, i \neq 0$. T is a set of tasks, and $\|T\| = N$, where N is number of tasks. We define a dummy task T_d that always corresponds to the dummy actor a_0. T_d takes zero execution time and it can always be inserted at the beginning of any sequence of actors without loss of generality.

2. *Resource conflicts*: The relation $C = \{(T_i, T_j) | T_i \nsim T_j\}$ denotes that the placement of T_i conflicts with placement of T_j. T_d is by definition in conflict with all the members of T.

3. Any actor $a_i \in S_a$ can only be scheduled for execution on the FPGA if all its preceding tasks have completed execution. Furthermore, if the corresponding module is not yet resident on the FPGA, it needs to be loaded, i.e., the corresponding resources reconfigured, prior to execution.

4.2 Problem Formulation

Before defining the scheduling problem, we need to distinguish three different types of dependencies: The first one, data dependencies, is obvious. The second is the conflict

relation introduced above that is due to the sharing of FPGA resources among the hardware modules. Finally, the third kind of dependencies arise because some actors cannot begin execution until its corresponding configuration task is completed. In order to compute this, we first need to discuss the problem of reconfiguration task generation.

4.2.1 Reconfiguration Tasks Generation

Definition 4.2.1 (True dependence) *Given a sequence of actors S_a. a_i is called truly dependent on a_j, written $a_j \prec a_i$ iff*

$$\nexists k, j < k < i : (a_k \nsim a_i) \wedge (\forall k', k < k' < i : a_k \neq a_i)$$

True dependence is based on the intuition that, for an actor a_i of task $t \in T$, not every occurrence of conflicting predecessors in the trace matters. It is the conflicting predecessor a_k that is closest to a_i that will have an impact on the reconfiguration decision for a_i. Furthermore, a_i must be the first actor of task type t in the trace subsequent to a_k.

Example 4.2.1 *In Figure 4.2, a_2 is truly dependent on a_1 but a_3 has no true dependence because it executes after another actor, a_2, of the same task. Also, because task A does not conflict with any tasks except task T_d, actor a_4 is truly dependent only on actor a_0.*

Now, each first appearance of a task in a trace will also necessitate exactly one reconfiguration task. Hence, the set of required reconfiguration tasks $S_r = (r_1, \ldots, r_l)$ may be found by inspecting the given trace once.[1]

Theorem 4.2.1 (Reconfiguration task instantiation) *For an actor a_i in a given trace S_a, there needs to be a corresponding reconfiguration actor (task) r_i if, and only if, $\exists a_j \in S_a$: $a_j \prec a_i$. In other words, if there exists a predecessor a_j in S_a on which a_i is truly dependent.*

[1]Note that the subscripts of reconfiguration tasks in S_r are in sequence but not necessary running as they correspond to the subscript of the associated actor, and not all actors need reconfiguration.

For each reconfiguration task r_i, two additional dependencies must be created. First, each r_i must complete before the corresponding actor a_i starts executing. Second, for a_j such that $a_j \prec a_i$, reconfiguration task r_i for a_i cannot start earlier than the completion of a_j on which a_i is truly dependent on because r_i affects the execution of a_j. The two dependencies are shown by adding an outgoing edge from r_i to a_i and one incoming edge from a_j to r_i.

Example 4.2.2 *Figure 4.3 shows the set of reconfiguration tasks generated for the running example as introduced in Example 4.1.1 as well as the additional scheduling dependencies in Figure 4.4. Note that actor a_3 does not induce a reconfiguration task to be created because it is preceded by a_2, an actor of the same task. It should be noted that while r_1 and r_4 should be preceded by a_0, the constraint is not enforced in practice since T_d has zero execution time.*

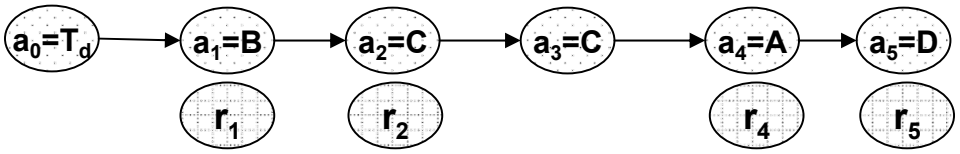


Figure 4.3: Reconfiguration task generation.

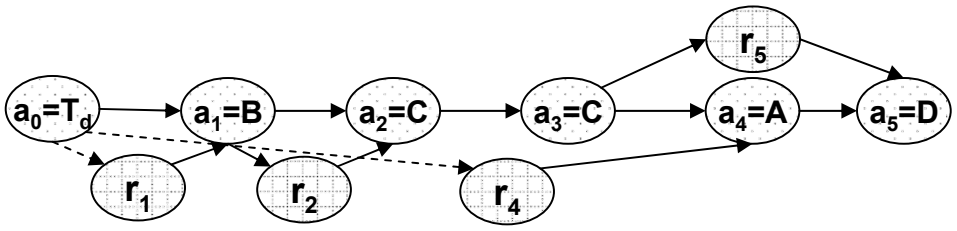


Figure 4.4: Dependence relations.

In summary, we have to consider the following three types of dependencies for scheduling after having all the required reconfiguration tasks generated:

- *Sequential precedence*:

  $P_s = \{(a_i, a_j) | (0 \leq i \leq n-1) \wedge (j = i+1)\}$;

- *Conflict (resource) precedence*:

  $P_c = \{(a_j, r_i) | a_j \prec a_i\}$; and

- *Reconfiguration precedence*:

  $P_r = \{(r_i, a_i) | (\exists r_i \in S_r)\}$.

The complete dependence relation is thus $P = P_s \cup P_r \cup P_c$.

4.2.2 Minimizing Schedule Length

Given the above, we are now in a position to state the scheduling problem formally. The following notation will be used throughout this chapter:

- $l(a_i)$: latency of actor a_i
- $s(a_i)$: the start time of actor a_i
- $f(a_i)$: the end time of actor a_i
- $l(r_i)$: latency of reconfiguration task r_i
- $s(r_i)$: the start time of reconfiguration task r_i
- $f(r_i)$: the finishing time of reconfiguration task r_i

Definition 4.2.2 (Feasible schedule) *A feasible schedule is an assignment of end times $f(a_i)$ and $f(r_i)$, respectively, to every actor $a_i \in S_a$ and reconfiguration task $r_i \in S_r$ such that*

all the above mentioned precedence constraints are satisfied, i.e., $\forall j$ *such that* $(X_i, X_j) \in P$ *then* $s(X_j) \geq f(X_i)$.

Example 4.2.3 *Figure 4.5 shows an example of a feasible schedule for five actors* a_1, a_2, a_3, a_4, a_5 *and the associated reconfiguration tasks.*

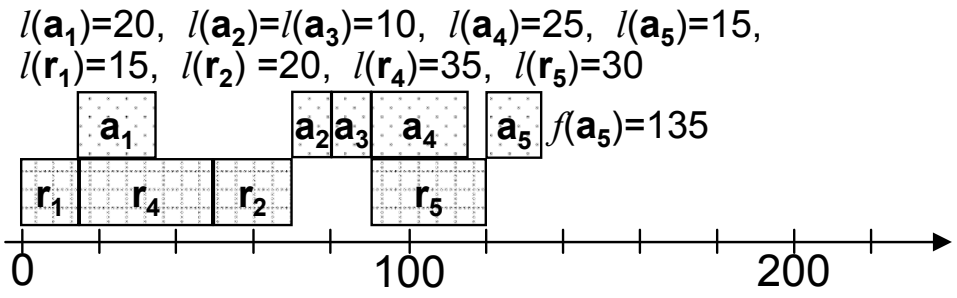


Figure 4.5: Feasible schedule for the problem introduced in Example 4.2.3.

Obviously, the reconfiguration interface may be regarded as a separate resource. The aim of a scheduling algorithm for this problem is to find a feasible schedule where $f(a_n)$ is minimized for a trace of actors $S_a = (a_0, a_1, a_2, \ldots, a_n)$.

4.3 Algorithm MLS

We shall now present the main result of this chapter, namely a polynomial time, latency-optimal scheduling algorithm for actors and reconfiguration tasks that we call *Modified List Scheduling* (MLS). The algorithm assumes that reconfiguration tasks can be pre-empted and resumed later. This is based on the way frame-based reconfigurable devices operate. Configuration for frame-based devices such as Xilinx FPGAs is achieved by writing a set of frames into the SRAM configuration memory of the device. It does not matter whether the reconfiguration process is carried out in 1, 2, or more phases as long as the affected

area is not again rewritten by other module configurations in between. Also, the algorithm prioritizes reconfiguration tasks by the order of appearance of their corresponding actors in the actor trace.

The MLS algorithm is shown in Algorithm 1. It consists mainly of two passes through the actor trace. In the first pass, the algorithm seeks to discover true dependences between the actors and generate the corresponding reconfiguration tasks S_r. To do this, we maintain a flag f_t for each task $t \in T$ and an index $prev_t$. We traverse the trace from a_1 to a_n. Assume that a_i is the current actor. If flag f_{a_i} is true, a corresponding reconfiguration task r_i will be created, and if $prev_t \neq -1$, r_i is to be preceded by actor a_{prev_t} (i.e., truly dependent on a_{prev_t}). $prev_t = 1$ when the reconfiguration task created is needed for the first occurrence of a_i. Furthermore, we record all ready reconfiguration tasks in a heap data structure H, ordered by the relative appearance order of the associated actor in the actor trace. In order to facilitate the preemptive scheduling of reconfiguration tasks, we maintain a **TimeRemaining** attribute for each of the tasks and this is initialized to the full reconfiguration latency required.

The second pass through the trace computes the actual scheduling time using preemptive scheduling of reconfiguration tasks. **current_A** is the current ready actor. In the case when there are no ready actors, we schedule a ready reconfiguration task r whose associated actor has the earliest appearance order in the actor trace. Otherwise, we schedule actor **current_A**. In the time $l($**current_A**$)$, we will try to schedule as many reconfiguration tasks sequentially as possible to configure the FPGA in parallel with the execution of **current_A**. However, the space given by the scheduled actor may not be enough for the **TimeRemain-**

ing of r to fill up. Such r's are inserted back into H with updated **TimeRemaining**. The algorithm terminates when the last actor a_n is scheduled.

Algorithm 1: MLS algorithm.

```
Input: Trace of actors: S_a;
Set of Conflicting Hardware Modules: C;
Set of tasks: T;
Result: Optimal Schedule Length
ForAll (f_t, prev_t : t ∈ T)  f_t ← true; prev_t ← −1;
for a_i ← a_1 to a_n : a_i ∈ S_a do
    if f_{a_i} is true then
        CreateReconfigurationTask (r_i);
        (r_i).TimeRemaining ← l(r_i);
        if prev_{a_i} ≠ −1 then AddEdge (a_{prev_{a_i}}, r_i;);
        AddEdge (r_i,a_i);
        ForAll (t ∈ T)  if  (t, a_i) ∈ C then  f_t ← true; prev_t ← i;
        if r_i has no preceding tasks then Insert (H, r_i);
if TaskReady (a_1) then current_A ← a_1; else current_A ← empty;
length ← 0;
while current_A ≠ a_n do
    if current_A is empty then
        r ← ExtractMax (H);
        length ← length +(r).TimeRemaining;
        current_A ← NextTask (r);
    else
        length ← length +l(current_A );
        T ← l(c);
        while H not empty ∧T ≠ 0 do
            r ← ExtractMax (H);
            if l(r) < T then  T ← T − l(r);
            else
                r..TimeRemaining ← r.TimeRemaining −T;
                T ← 0;
                Insert (H, r);
        ForAll (r ∈ DependsOn (current_A))  Insert (H, r);
        if TaskReady (NextTask (current_A)) then
            current_A ← NextTask (current_A);
        else  current_A ← empty;
length ← length +l(a_n);
return length;
```

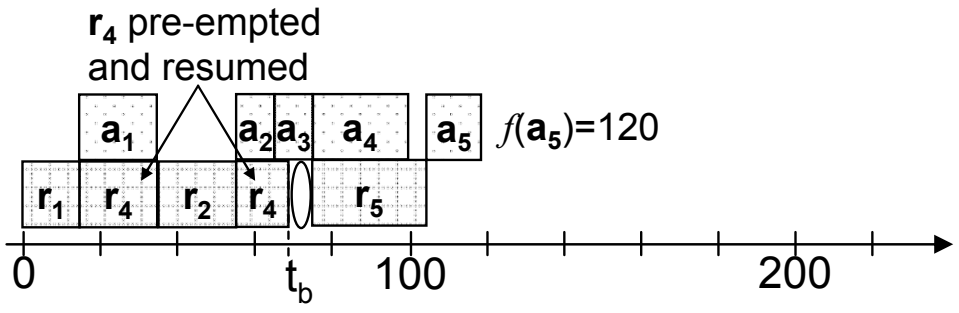


Figure 4.6: Example of optimal feasible schedule produced by MLS.

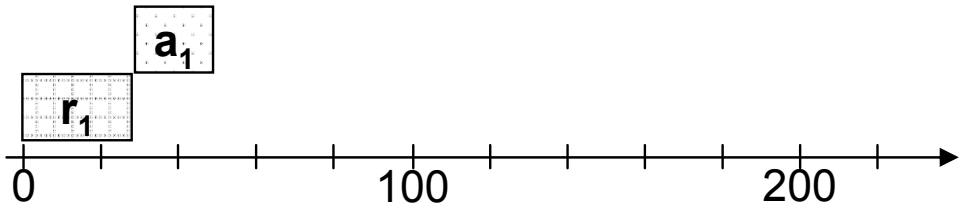
4.3.1 Bubbles in the Reconfiguration Schedule

Before describing the optimality proof of our algorithm, we will illustrate a key idea used in the proof by means of an example.

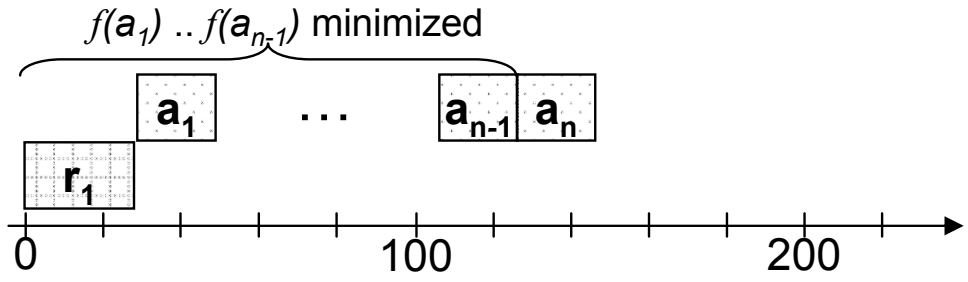
Example 4.3.1 *Figure 4.6 shows the result of applying the MLS algorithm to our running example. In the schedule of the reconfiguration tasks, we can see* 'bubbles'*, i.e., time intervals in which there are no reconfiguration tasks occupying the reconfiguration interface of the FPGA. By the greedy nature of list scheduling, if a ready task exists, it will always be scheduled. Therefore, a bubble can only exist at a time instant t_b when there are no ready reconfiguration tasks to be scheduled at t_b. In the example, a bubble exists between r_4 and r_5 because there are no ready tasks before the completion of a_3. The figure also shows the preemption of reconfiguration task r_4. r_4 is preempted at time 35 before being continued at time 55.*

4.3.2 Proof of Optimality

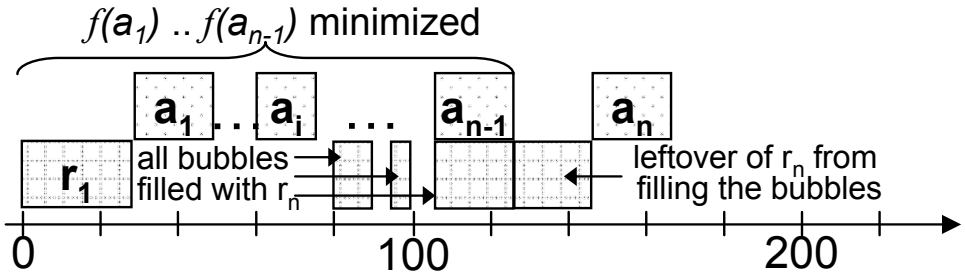
We shall now present the induction proof for the optimality of the MLS algorithm. Note that while in general list scheduling is but a heuristic, in the case of MLS, the order of an actor's appearance in the schedule is pre-determined by the trace. Furthermore, due to the fixed priority function, the time slots occupied by a reconfiguration task cannot be used



(a) Base case



(b) Inductive step: a_n has no reconfiguration task



(c) Inductive step: a_n has reconfiguration task r_n

Figure 4.7: Induction proof cases for MLS.

by any other once the actor trace and conflicts are known. These two facts combined to guarantee the optimality of MLS.

Theorem 4.3.1 (Optimality) *Given a trace of actors S_a of length n with the required reconfiguration tasks introduced in accordance to Theorem 4.2.1 and the corresponding dependencies added, the MLS algorithm with (a) preemption of reconfiguration tasks, and (b) task priority given by the number of successor tasks in the precedence graph will yield a schedule with the smallest possible $f(a_n)$.*

Proof.

We show Theorem 4.3.1 by induction on the length of the actor trace S_a.

Base case: $|S_a| - 1 = n = 1$. Figure 4.7(a) shows the base case. The dummy actor $a_0 = T_d$ that has zero latency is not shown. r_1 followed by a_1 is the optimal schedule. MLS would trivially generate exactly this schedule.

Inductive Step:

Consider now a trace of length $n > 1$ and assume that $f(a_i)$ is minimal for $1 \leq i < n$. We need to show that when a_n is considered: (1) the resulting schedule created by including a_n is still optimal, and (2) that this schedule can be computed by MLS.

There are 2 possibilities:

- a_n has no reconfiguration task as a predecessor.

  This trivial case is shown in Figure 4.7(b). a_n is simply appended to the actor schedule. Since $f(a_{n-1})$ is optimal, $f(a_n)$ is also optimal since there is no bubble between a_{n-1} and a_n. It is easy to see that MLS will yield exactly this schedule.

- a_n has a preceding reconfiguration task r_n.

  There must then exist an actor $a_i, 0 \leq i < n$ such that $a_i \prec a_n$. Observe from the assumption made in the induction step that $f(a_i)$ is minimal, i.e., a_i cannot be scheduled earlier. Due to the precedence constraints and inductive assumption, r_n cannot be scheduled earlier than $f(a_i)$. So, the time to schedule r_n is in the interval from $f(a_i)$ to $s(a_n)$. Now preemption allows us to schedule and split the latency $l(r_n)$ of the reconfiguration task r_n to fill up all the bubbles between $f(a_i)$ and $f(a_{n-1})$ left in the reconfiguration schedule. Any leftover time of $l(r_n)$ will be simply scheduled after $f(a_{n-1})$. Figure 4.7(c) shows this case. The bubbles between $f(a_i)$ and $f(a_{n-1})$ exist because there are no reconfiguration task r_j, $j < n$, that can use them. Either

the total length of these bubbles is (a) completely sufficient to absorb $l(r_n)$, or (b) insufficient and hence there is some amount of outstanding $l(r_n)$. For the former, the schedule is optimal since all we need to do is to append a_n immediately after a_{n-1}. For the latter, the schedule obtained by first scheduling the remainder of r_n followed immediately by the start of a_n is also optimal. Finally, note that scheduling r_n using MLS will not interfere or preempt other previously scheduled reconfiguration tasks $r_j,\ j < n$ as these will have higher priorities, and only those bubbles starting from $f(a_i)$ (unusable by any $r_j, j < n$) are used for r_n.

Thus, we have shown that if the inductive step holds, the resulting schedule for S_a of length n is optimal. Since the base case is true, the proposed property holds true for all traces S_a of any size more than one.

4.3.3 Further Clarifications

The optimality of the proposed algorithm is constrained by 2 factors: 1) available hardware resources on the FPGA and 2) the conflict relationships between the tasks. Furthermore, it is important to note that while the pre-emption of reconfiguration tasks frees the reconfiguration port so that other reconfiguration tasks can proceed, the configured resources (e.g. frames, columns) are not freed to be occupied by other tasks. We shall further illustrate clarify the workings of the algorithm with one pathological case here.

In the first pathological case, all tasks are placed on the FPGA. Each tasks require significant hardware resources for implementation and thus every task more or less occupies the whole FPGA. In this case, every tasks conflicts with one another. An analogous situation would be the case where the hardware resources available are so few that all tasks placed on the FPGA are forced to be in conflict with one another. Our algorithm will still return an optimal schedule in this case. It should be noted that since each task is in re-

source conflicts with all other tasks, it is not possible to configure the FPGA in parallel with the execution of the tasks. The only and hence optimal schedule in this case would be to configure the FPGA each time a different task is being executed.

4.4 Case Study

4.4.1 H264-encoder Case Study

We use a H.264 [67] encoder application as a case study of the effectiveness of our algorithm. Based on profiling, we identified 15 loops that take up most of the computation time in the application. The hardware implementation of these loops were synthesized using Xilinx's ISE. Table 4.3 gives the details of the loops. The loops are named by their containing functions' names and identifiers assigned by the compiler. For example, `biari_encode_symbol-6` is loop 6 in function `biari_encode_symbol`. The target device for synthesis is Xilinx Virtex-II XC2V6000 device[72].

Table 4.1 shows the characteristics of the application using two actor traces obtained with the 15 loops. It shows the length of the actor traces and the number of unique patterns occurring within the trace. A *pattern* is a maximal acyclic sequence of actors that occurs repeatedly in the trace. Two patterns are considered different if they differ in at least one actor. Intuitively speaking, the more unique patterns they are in the trace would imply greater adaptability and variation for the given input. We obtained the shorter trace by encoding one frame and the longer trace by encoding two consecutive frames. The frames are 704 by 576 pixels in size. All the hardware modules are assumed to be running at a frequency of 50 MHz.

We tested the effectiveness of Algorithm MLS by observing the scheduling results for different sets of resource conflicts. Prefetching is beneficial only if there is sufficient

amount of time between the execution of conflicting hardware modules. Otherwise, there is no concurrency between execution and reconfiguration of the hardware modules. We obtained different sets of resource conflicts by allowing conflicts (i.e., placement overlap) to occur if the minimum average number of execution cycles between the actors involved are at least above a given threshold number of cycles. Table 4.2 shows the resource conflicts for thresholds between 700 to 1300 cycles. Obviously, the number of conflicts decreases when the threshold is increased. We shall report the impact of different number of conflicts on the schedule length in Section 4.4.3.2.

Trace	**Num. of Frames Encoded**	**Num. of Actors**	**Num. Of Unique Patterns**
Short	1	35,622,092	52
Long	2	185,232,537	100

Table 4.1: Characteristics of the two traces.

4.4.2 Experiment Setup

To demonstrate the effectiveness of our approach, we compared it against three algorithms: two different online Least Mean Square Predictor, and a simple scheduler.

Simple Scheduler The Simple Scheduler does not attempt any form of prefetching. Instead, it simply maintains a record of the currently FPGA configuration and only schedules a reconfiguration on demand if the actor to be executed is not yet in the FPGA. It is reasonable to expect that any prefetching approach should do no worse than the Simple Scheduler.

Minimum avg. cycles btw conflicting resource	Num. of conflicts
700	48
800	42
900	39
1000	36
1100	25
1200	15
1300	7

Table 4.2: Resource conflicts.

We therefore used the schedule length computed by the Simple Scheduler as the baseline for our comparisons.

Least Mean Square Online Predictor A (LMSA-a) This is an online predictor that is similar to that described in [42, 11]. The Least Mean Square Filter is used as the predictor function. However, because the target FPGA architecture considered in this thesis is different (their architecture [13] supports relocation and defragmentation), our approach does not use the priority function that is based on the configuration sizes and the different eviction policies. Rather, the hardware module evicted are those in conflict with the module current being prefetched.

Least Mean Square Online Predictor B (LMSA-b) This is a modification of LMSA-a. Instead of predicting the next hardware task, the algorithm predicts the next conflicting task. An important difference here is that instead of keeping historical information for only

Loop Name (func_name-loop_id)	**Num. Of Slices**	**Num. Of Cycles (long trace)**	**Num. Of Cycles (short trace)**
biari_encode_symbol-1	1552	1,787,969,064	393,559,992
biari_encode_symbol-6	1486	458,442,739	103,953,560
dct_luma-1	1597	979,861,500	159,476,940
dct_luma-3	3316	8,100,188,400	1,318,342,704
dct_luma-4	1428	740,339,800	120,493,688
dct_luma-5	3314	2,917,809,800	474,886,888
dct_luma-8	1052	152,422,900	24,807,524
dct_luma-9	1388	609,691,600	99,230,096
Mode_Decision_for_4x4IntraBlocks-4	1234	106,317,960	21,263,592
Mode_Decision_for_4x4IntraBlocks-5	1472	567,029,120	113,405,824
reset_coding_state-1	1268	18,720,876	3,593,348
RDCost_for_4x4IntraBlocks-1	1222	106,317,960	21,263,592
RDCost_for_4x4IntraBlocks-2	1351	425,271,840	85,054,368
writeLumaCoeff4x4_CABAC-1	1812	466,152,012	107,780,832
write_significant_coefficients-1	1985	1,674,754,726	365,849,786

Table 4.3: Hardware modules, the hardware area occupied and the number of cycles taken up in the application.

the currently executing task, LMSA-b requires the information for all tasks to be kept in order to predict the next conflicting task.

4.4.3 Experimental Results

4.4.3.1 Scaling the Reconfiguration Overhead

We seek to show the effect of increasing configuration overhead on the scheduling length. For the FPGA we used, empirical measurements showed a configuration time of about 400μsec per CLB column. Using this high overhead, most applications stand to gain little by hardware acceleration via dynamic reconfiguration. Fortunately, recent works [43, 12] have shown that the overhead of configuring partial bitstreams can be potentially reduced by factors of 20 or more. In particular, assuming the geometry of a Xilinx device, we ran experiments by varying the reconfiguration speed from between 1μsec to 20μsec per CLB column.

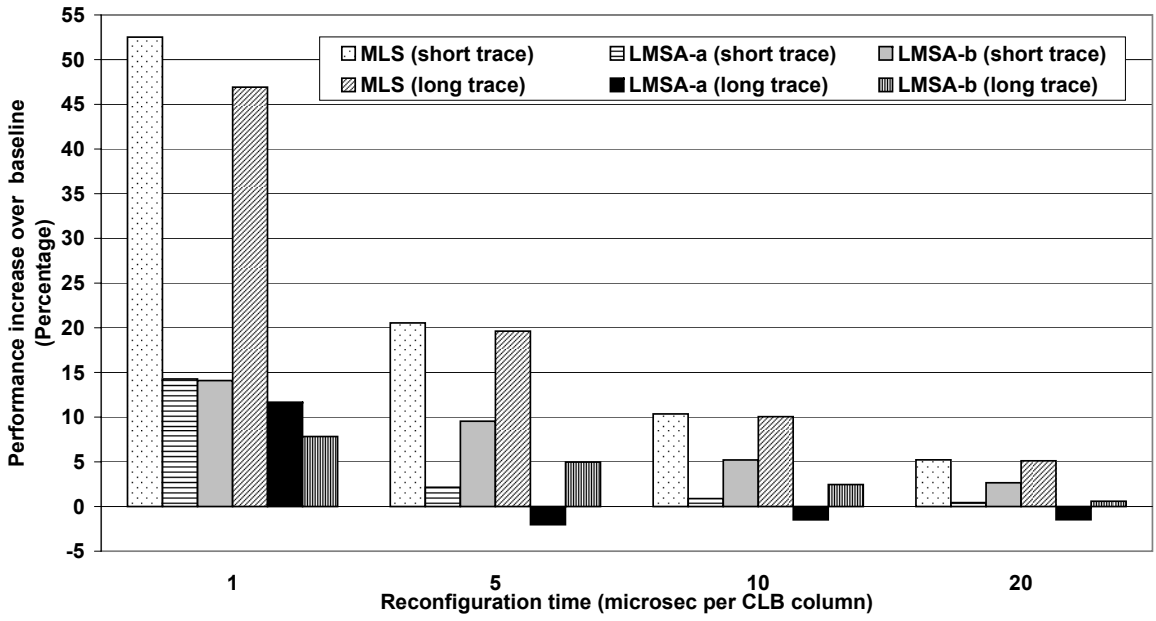


Figure 4.8: Speedup over baseline plotted against increasing reconfiguration time.

Figure 4.8 shows the performance increase of the different approaches over the schedule produced by the Simple Scheduler. The threshold of the minimum average execution cycles between two conflicting hardware module is set to 1000 cycles for this experiment. We observe that as reconfiguration speed decreases, the performance gain achieved by all the approaches decreases. With a high reconfiguration overhead, execution just has to wait till reconfiguration completes. The single reconfiguration port also becomes a bottle-neck. Over the range of reconfiguration overheads we considered, the schedule produced by MLS outperforms the others in every case. At best, it can be 30 percent better than those produced by the other schemes.

Another interesting observation is that LMSA-b performs better than LMSA-a in general. This could be because predicting the next conflicting hardware module gives prefetching more time and a miss in the prefetching is less costly. LMSA-a and LMSA-b also perform worse in the longer trace because the execution order is more complex, as shown in the higher number of unique patterns. Another interesting note is that LMSA-a can perform worse than the Simple Scheduler. Prefetch misses can sometimes increase the number of reconfigurations beyond what is normally needed because incorrectly predicted prefetches can evict hardware modules which are otherwise not evicted by the Simple Scheduler.

4.4.3.2 Scaling the Number of Conflicts

Figure 4.9 shows the performance of the different approaches under the different conflict sets listed in Table 4.2. We set the reconfiguration speed to be 10μs per CLB column for this experiment. We observe that as the number of conflicts decreases, the performance gain of the schedules increases for MLS. However, the same cannot be said of LMSA-a and LMSA-b. We attribute this to the fact that different conflict sets can cause different mispredictions for the same actor trace. MLS outperforms LMSA-a and LMSA-b in all

cases. In the case of the long trace, the difference between the schedules are significant. This shows that the more complex the execution order, the more difficult it is for the online prefetcher to yield a good schedule. Interestingly, in the longer trace, the LMSA-a performs better than LMSA-b when the number of conflicts is larger.

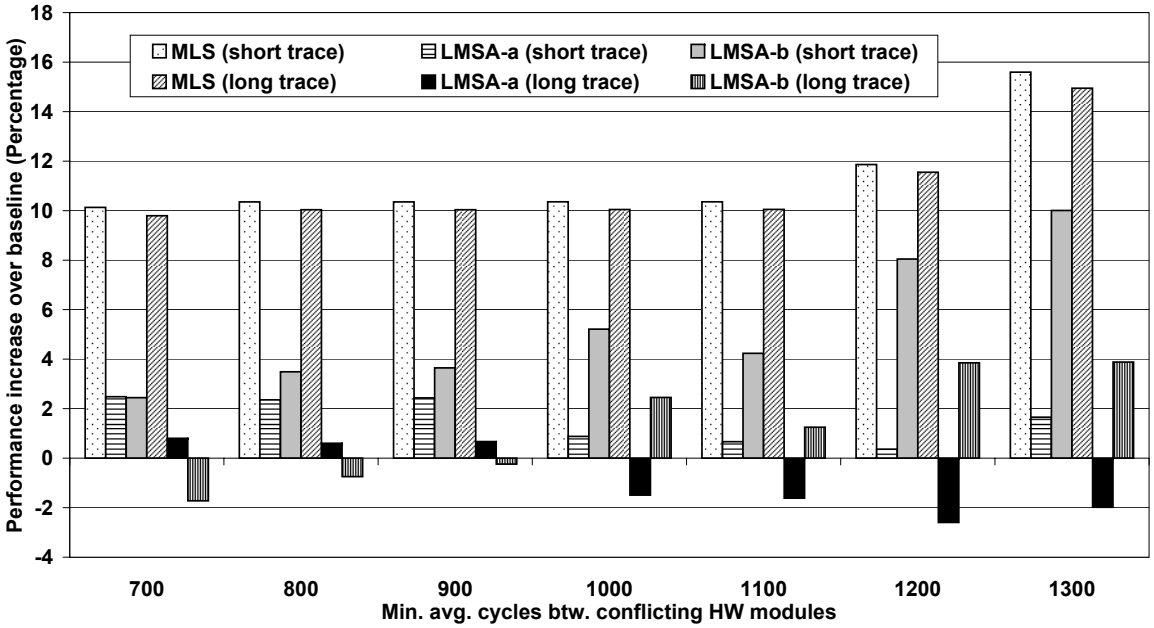


Figure 4.9: Speedup over baseline plotted against decreasing number of conflicts.

4.5 Summary

In this chapter, we presented an algorithm for the scheduling of reconfiguration tasks for FPGA-based hardware acceleration at the electronic system level. For a given trace of complex computational kernels for which there exist hardware accelerators, we analyze the dependencies between actors into (a) data dependencies, (b) resource dependencies (conflicts), and (c) reconfiguration dependencies. Our algorithm inspects each actor in the trace and determines whether a reconfiguration task is needed and if so, schedules such

a task in accordance with given dependencies. We provided a proof that the algorithm always yields the optimal result in terms of the overall execution time (latency) of the trace. Furthermore, it is polynomial in the length of the given trace of actor activations. A realistic case study using the H.264 encoder has been provided to show the benefits and sensitivity of the results.

We had assumed that the conflict relation between actors are given. Conflicts are dependent on the placement of the corresponding hardware modules in the FPGA. In the future, we would like to extend and optimize also the following scenarios: Assuming that an actor may be executed either in software or has several instances that can be placed onto different locations in the reconfigurable fabric, then we would like to find the best implementation and placement decisions that will optimize the overall execution time of the application.

Due to its polynomial nature, our algorithm scales better than previously proposed ILP approaches. Nonetheless, the analysis of a long trace is still quite time-consuming. Hence, we would like to investigate heuristics that work at the level of the (static) control flow graphs of the application. While such heuristics may not always produce the optimal solutions, they may yield solutions that are "good enough" in practice, without the need for obtaining and processing long traces.

Chapter 5

Interprocedural Placement-Aware Configuration Scheduling

In general, hardware implementations of computation are faster than the equivalent computation performed on general purpose processors. However, the speedup obtained by computation executed in FPGAs are offset by the huge reconfiguration latency required to configure the FPGA during run-time. Chapter 1 and 2 has shown that pFPGAs provide an additional opportunity for this reconfiguration to occur in parallel with both hardware and software execution. Thus reconfiguration scheduling becomes critical for the reduction of the reconfiguration overhead. The reconfiguration schedule needs to maximize the parallelism between the necessary reconfigurations and the execution, both hardware and software, of the application,

The configuration scheduling problem is made complicated by the fact that the hardware modules that are to be executed may compete with each other for resources on the FPGA. Such modules are said to be in 'conflict'. In cases where two hardware modules A and B conflict with each other with module A is already loaded in the FPGA, it follows that it is necessary to load B into hardware before B is executed. Therefore, whether a re-

configuration is necessary is dependent on the conflicts relationship between the hardware modules.

In this chapter, we propose an algorithm that provides appropriate cues for the compiler to insert configuration commands into the control flow graph of the application so that the overall execution time is minimized. This algorithm relies on characteristics of the compiled software application and the conflict information of the hardware modules to achieve this goal.

The chapter is organized as follows. Section 5.1 gives the background information that forms that context of the problem we are solving. After illustrating our motivation with two examples in Section 5.2, we present the problem formulation in Section 5.3. Section 5.4 describes the proposed algorithm. In Section 5.5, we present the experiment results before concluding in Section 5.6.

5.1 Background

5.1.1 Architecture Model

We consider the architecture model as shown in Figure 5.1. The model is realistic for archtectures such as the Xilinx Virtex Family of FPGAs, especially Virtex-II Pro, IV and V. We show the major components of interest in Figure 5.1. Memory is where the software code and data are stored, together with the bitstreams to be loaded onto the reconfigurable region. The CPU is the main controller of application execution and is responsible for initiating reconfiguration of the reconfigurable region. The reconfiguration manager is a hardware module that loads bitstream data from the memory upon requests issued by the CPU.

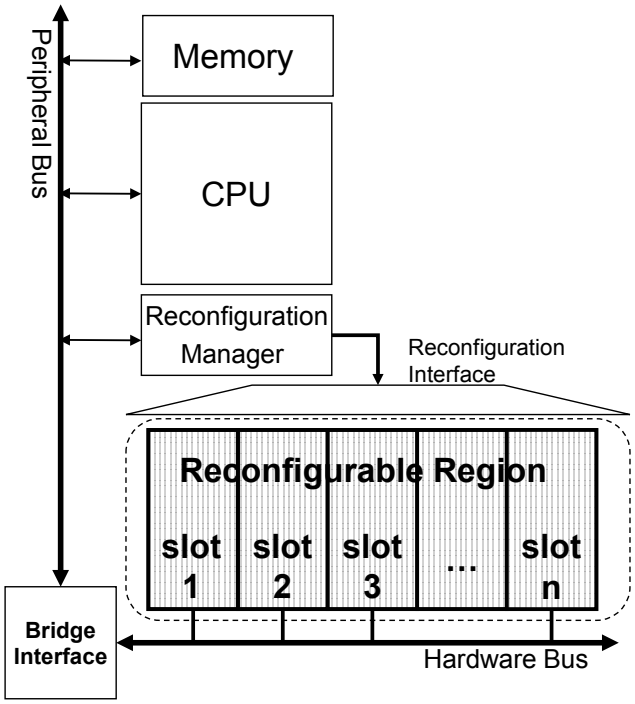


Figure 5.1: Architecture model for interprocedural placement-aware configuration scheduling.

The reconfigurable region is where the hardware modules of the application are executed. It contains n slots where hardware modules can be placed. Each hardware module must be placed on contiguous slots within the reconfigurable region. Through the bridge interface, the hardware modules can read the memory in bursts and share the same address space as the CPU. Although it is possible for hardware modules to be relocated during run-time, it could be computationally expensive because the bitstreams for relocation needs to be generated at run-time. As such, the placement of the hardware modules are decided during design time. We consider any two placements of the hardware modules that overlap with each other to be in 'physical placement conflict' (or just 'conflict' for the rest of the chapter). For example, if one hardware module is placed on slots 1 and 2 and the another hardware module is placed on slots 2 and 3, these two hardware modules are in conflict.

Conflicting hardware modules cannot be loaded and run on the reconfigurable region simultaneously.

Informally, we aim to minimize the execution time of a single, sequential application for this platform. The application consists of a combination of a program and m hardware modules. These hardware modules are required to be loaded on the reconfigurable region prior to their execution.

5.1.2 Reconfiguration Library Support

The architecture described above supports the preemption and subsequent resumption of a hardware module. This is based on the insight that frame-based devices such as Xilinx FPGAs allow the configuration loading to occur in non-contiguous temporal segment as long as previously loaded bits are not overwritten. To support the software control of reconfiguring the reconfigurable region, we define a set of library calls that interfaces with the reconfiguration manager and yet hide the underlying architecture details from the programmer.

The library requires some internal data structures to maintain the following information: (a) the state of FPGA(i.e. what hardware modules are currently loaded on the FPGA), (b) the hardware module being loaded onto the FPGA (if any), (c) the conflict information between modules, (d) the location and length of the hardware module bitstreams, and (e) the reconfiguration data required for resumption of reconfiguration. The last one requires some further explanation. Suppose we preempt the reconfiguration of a hardware module that consists of 5 CLB columns and 3 columns have been loaded thus far. We need to remember the information about how much of the hardware module has been reconfigured so as to support a future resumption of the reconfiguration of this module. We shall now proceed to explain how the information (d) and (e) mentioned above are stored.

We maintain a structure called HW_load_info for each hardware module to store the information needed to support the reconfiguration of the module. It has 3 fields: address, length and recon_unit_size. address is where the bitstream is located in the memory. length is the size of the bitstream that needs to be loaded. recon_unit_size is the basic reconfiguration granularity at which the hardware module is to be atomically configured each time. If recon_unit_size is 1 CLB column, then 1 CLB column of bitstream data will be loaded onto the reconfigurable region at a time. In other words, when a reconfiguration is preempted, the library ensures that a multiple of recon_unit_size have been loaded always. The actual value of recon_unit_size is target-device dependent, given that different devices have different column lengths. We store the structures of each hardware module in in an internal data structure HW_LOAD_TABLE that is declared in the following in C-like pseudo code. It should be noted that by each hardware module is assigned a unique hw_id that is also used as an index of this table.

```
typedef struct hw_load_info
{
  void *address;
  int length;
  int recon_unit_size;
} HW_load_info;
HW_load_info HW_LOAD_TABLE[NUM_OF_HARDWARE];
```

We maintain the information needed to support resumption of reconfiguration in a structure called HW_resume_info for each hardware modules. HW_resume_info is identical to the HW_load_info except except that it contains a field called valid that indicates whether resumption is still valid for the hardware module. RESUMPTION_Q holds all the HW_resume_infos of the hardware modules.

```
typedef struct hw_resume_info
{
  void *address;
  int length;
  int recon_unit_size;
  int valid;
} HW_resume_info;
HW_resume_info RESUMPTION_Q[NUM_OF_HARDWARE];
```

The following library calls support the software control (i.e. initialization, pre-emption, and resumption) of run-time partial reconfiguration:

`load(hw_id)`: This is a non-blocking library call that loads the bitstream of a hardware module `hw_id`. `load` looks up the `valid` field of `hw_id`'s entry in the `RESUMPTION_Q`. If it is invalid (i.e. the hardware module is not yet loaded on the FPGA), `load` looks up `hw_id`'s entry in `HW_LOAD_TABLE` for the starting address and length of the bitstream to be passed to the reconfiguration manager. If it is valid (i.e. configuration resumption is possible), `load` will pass the values stored in `RESUMPTION_Q` to the reconfiguration manager. This is a non-blocking call because the reconfiguration manager will start the loading of the hardware module. When a hardware module is loaded, all it's conflicting modules' entries in `RESUMPTION_Q` will be invalidated.

`currently_reconfiguring()`: Returns the id of the hardware module currently being reconfigured, if any. Returns -1 if there are no hardware modules being reconfigured.

`is_loaded(hw)`: This returns a boolean value indicating whether a hardware module is loaded on the reconfigurable region.

`hw_exec(hw_id, ...)`: A blocking call that returns upon the completion of the execution of the hardware module indicated by `hw_id`. The rest of the parameters are inputs (usually some register and address values) that are needed to be transferred to the hardware module. If `hw_id` is already loaded on the FPGA, the execution starts immediately. If the `hw_id` is not yet loaded, execution of **hw** may be delayed by either full or partial loading of the hardware module. This delay forms the reconfiguration cost paid in expense of better performance. However, we seek to reduce this delay through configuration prefetching.

5.1.3 Interprocedural Control Flow Graphs

The *control flow graph*(CFG) is a common, intermediate-level data structure used by compilers to represent applications. The CFG displays all the possible paths that might be traversed for the procedure that it represents. Every node in the graph is a *basic block*, that has only one entry instruction and one exit instruction and no jump instructions in between the entry and exit instructions.

The CFG usually represents the control flow of a single procedure. While the CFG is useful for intra-procedural (i.e. within a procedure) optimizations, it may not be suitable for optimizations that spans across procedure call boundaries. In such cases, an *interprocedural control flow graph* (ICFG)[25] may be a better choice as a representation of the application for optimization. All possible paths that might be taken during run-time are represented completely in an ICFG.

As an example, we present in Figure 5.2 the C code for computing `HeapSort` and its associated ICFG in Figure 5.3. The ICFG contains the control flow of all the procedures of the HeapSort program. It should be noted that the CFG of the `swap` procedure is not included because it is inlined after applying compiler optimizations. Observe the following about our construction of the ICFG:

```c
#include <stdio.h>
#include <stdlib.h>

void HEAPSORT(int heap[100], int n);
void swap(int *p, int *q);
int BUILD_HEAP(int heap[100], int n);
void HEAPIFY(int heap[100], int i, int heap_size);

#define PARENT(i) ((i)/2)
#define LEFT(i)   (2*(i))
#define RIGHT(i)  (2*(i)+1)


static inline void swap(int *p, int *q)
{
    int t;
    t = *p; *p = *q; *q = t;
}

void main()
{
    int i, j, n, heap[110];
    while (1) {
        printf("Enter the number of element (0 to exit): ");
        scanf("%d", &n);
        if (n == 0) break;
        for (i = 1; i <= n; ++i) scanf("%d", &heap[i]);
        HEAPSORT(heap, n);
        printf("The sorted List is:\n");
        for (i = 1; i <= n; ++i) printf("%d ", heap[i]);
    }
}
```

Figure 5.2: HeapSort C code example.

```
void HEAPSORT(int heap[100], int n)
{
    int i, heap_size;
    heap_size = BUILD_HEAP(heap, n);
    for (i = n; i >= 2; --i) {
        swap(&heap[1], &heap[i]);
        --heap_size;
        HEAPIFY(heap, 1, heap_size);
    }
}

int BUILD_HEAP(int heap[100], int n)
{
    int i, heap_size;
    heap_size = n;
    for (i = floor(n / 2); i >= 1; --i) HEAPIFY(heap, i, heap_size);
    return heap_size;
}

void HEAPIFY(int heap[100], int i, int heap_size)
{
    int l, r, largest;
    l = LEFT(i);
    r = RIGHT(i);

    if (l <= heap_size && heap[l] > heap[i]) largest = l;
    else largest = i;
    if (r <= heap_size && heap[r] > heap[largest]) largest = r;
    while(largest!=i)
    {
        swap(&heap[i], &heap[largest]);
        i = largest; l = LEFT(i); r=RIGHT(i);
        if( l<=heap_size && heap[l] > heap[i]) largest=l;
        else largest=i;
        if (r <= heap_size && heap[r] > heap[largest]) largest = r;
    }
}
```

Figure 5.2: HeapSort C code example.

- **Entry and Exit Nodes.** In the ICFG, every procedure has a single entry and single exit. While it is possible to have multiple return instructions for a procedure written say in a high-level language such as C, we add an additional exit basic block and replace the multiple return instructions with a branch to the added exit basic block. Also, we have included a `start` and `end` node to indicate where the program begins and ends normally. Normally, the program begins at the start of the `main` procedure and end after the exit basic block of the `main` procedure.

- **Call and Return Control Flow Edges.** Apart from edges that indicate the usual control-flow transfers (i.e. branch taken or fall-through), ICFGs include two additional types of control-flow edges. For example, the edge from node 3 to node 7 denotes a procedure call being made by the `main` procedure to the `HEAPSORT` procedure. Similarly, the edge from node 18 to node 9 indicates that upon the completion of the procedure call to `BUILD_HEAP`, the control returns to node 9 of procedure `HEAPSORT`. In Figure 5.3, the call edges are indicated by thicker lines in bold while the return edges are indicated by dotted lines. It should be noted that if a call site makes a call through a procedure pointer, it is possible for the call site to have multiple out-going edges.

- **Invalid Paths.** While the ICFG contains all the possible paths that could be traversed during run-time by the application, it contains paths that are invalid. For example, the path $15 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 22 \rightarrow 26 \rightarrow 11$ is invalid. $15 \rightarrow 29$ is a call edge that indicates a procedure call made by `BUILD_HEAP` to `HEAPIFY`. We normally expect the call to return to the caller, but $26 \rightarrow 11$ is a return edge to `HEAPSORT`, thus making the above given path invalid. It should be noted that for every call edge there is exactly one corresponding return edge.

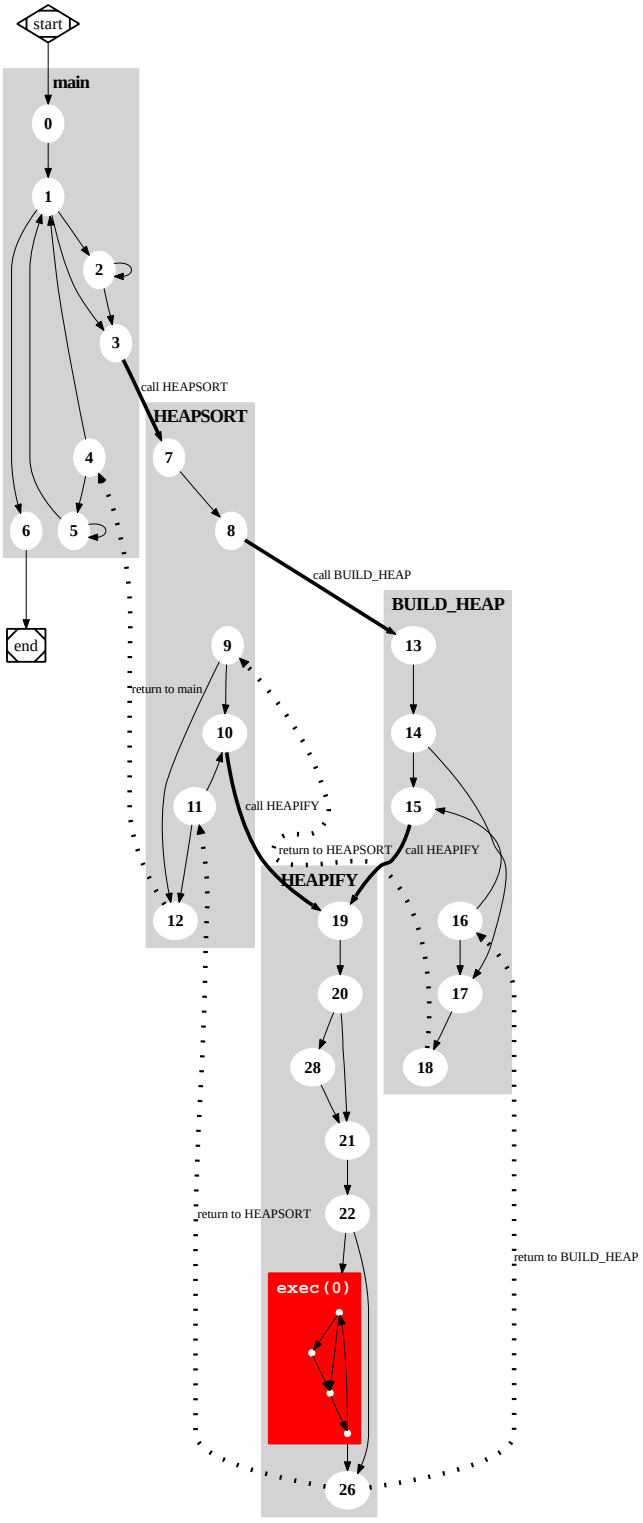


Figure 5.3: HeapSort interprocedural control flow graph.

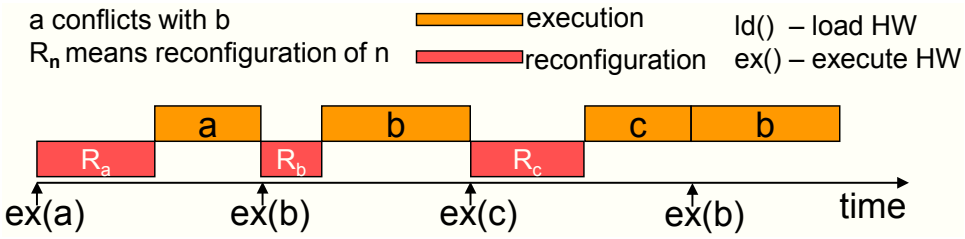
- **Hardware Nodes.** The regions that are designated for hardware implementation are collapsed into a node in the ICFG. In Figure 5.3, the loop in procedure **HEAPIFY** is converted into a single block of code that begins with a call to `hw_exec(0)`, followed by necessary control flow to other basic blocks in the ICFG. We refer to these converted blocks of code that begins with a `hw_exec` call as 'hardware nodes' because these blocks of code initiate the execution of hardware modules. It should be noted that these hardware nodes could contain multiple jump instructions, akin to superblocks[31] that has a single entry instruction and multiple exit instructions. The hardware nodes are depicted by a rectangular box and the software basic blocks are indicated by ovals in figures showing ICFG. Furthermore, every hardware node contains exactly one `hw_exec` call.

For the rest of the chapter, we denote an ICFG as a directed graph $G = (V, E, C, I, U, HW)$. V is the set of all the nodes in the ICFG. E is the set of edges denoting all possible control flow transfers in the program. $head(e)$ and $tail(e)$ refers to the begin and end nodes of edge e respectively. C is the set of all call sites and $C \subset V$. I is the set of all entry nodes of each procedure and $I \subset V$. U is the set of all exit nodes of each procedure and $U \subset V$. $HW \subset V$ is the set of all the hardware nodes in the ICFG. Each hardware node is assigned a unique ID. Recall that every hardware node contains exactly one **hw_exec** call. For convenience's sake, we shall refer to the hardware node and its corresponding hardware modules interchangeably. Thus, two conflicting hardware nodes $hw1$ and $hw2$ are written as $hw1 \nsim hw2$.

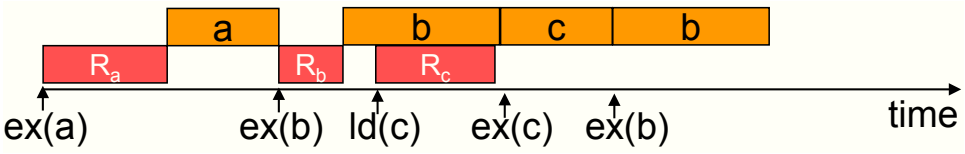
5.2 Motivation

After replacing the regions designated to run on FPGA with hardware nodes, the compiled ICFG should execute on the reconfigurable platform. However, without inserting the `load` library calls for optimization, the execution of such executables results in what we call "fetch-on-demand"(FOD) schedules during run-time. Consider the execution sequence *abcb* for hardware modules a, b and c . Figure 5.4(a) shows how the execution of the hardware modules pan out during run-time. According to the `hw_exec` call semantics, since no hardware modules are preloaded, execution of the hardware modules are preceded by a reconfiguration phase if the desired hardware module is not yet present on the FPGA. Thus, in the example shown in Figure 5.4(a), we need to reconfigure *a*, *b*, *c* prior to their execution. However, it should be noted that the last execution of *b* did not require a reconfiguration since there are no conflicts between *b* and *c* and thus hardware module *b* is still present on the FPGA after the execution of *c*.

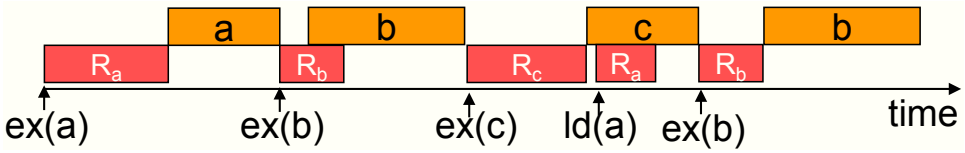
The FOD schedule is sub-optimal and it can be improved upon with appropriately placed `load` library calls. Figure 5.4(b) shows that by inserting an appropriate `load` *c* call during the execution of *b*, the overall execution time is reduced. This is because the reconfiguration of *c* can occur in parallel with the execution of *b* as *c* and *b* do not conflict with each other. On the other hand, a misplaced `load` library call may result in a schedule that is longer than FOD. Figure 5.4(c) shows the case that a `load` *a* call during the execution of *c* results in an additional reconfiguration of *b* later, hence lengthening the original FOD schedule. Although `load` library calls are necessary to improve upon the execution time, these calls must be appropriately placed and the current chapter aims to insert these library calls so that overall execution time is minimized. The following examples show how the insertion of load library calls to prefetch hardware module is a complex issue.



(a) The fetch-on-demand schedle



(b) Inserting prefetch loads decrease execution time

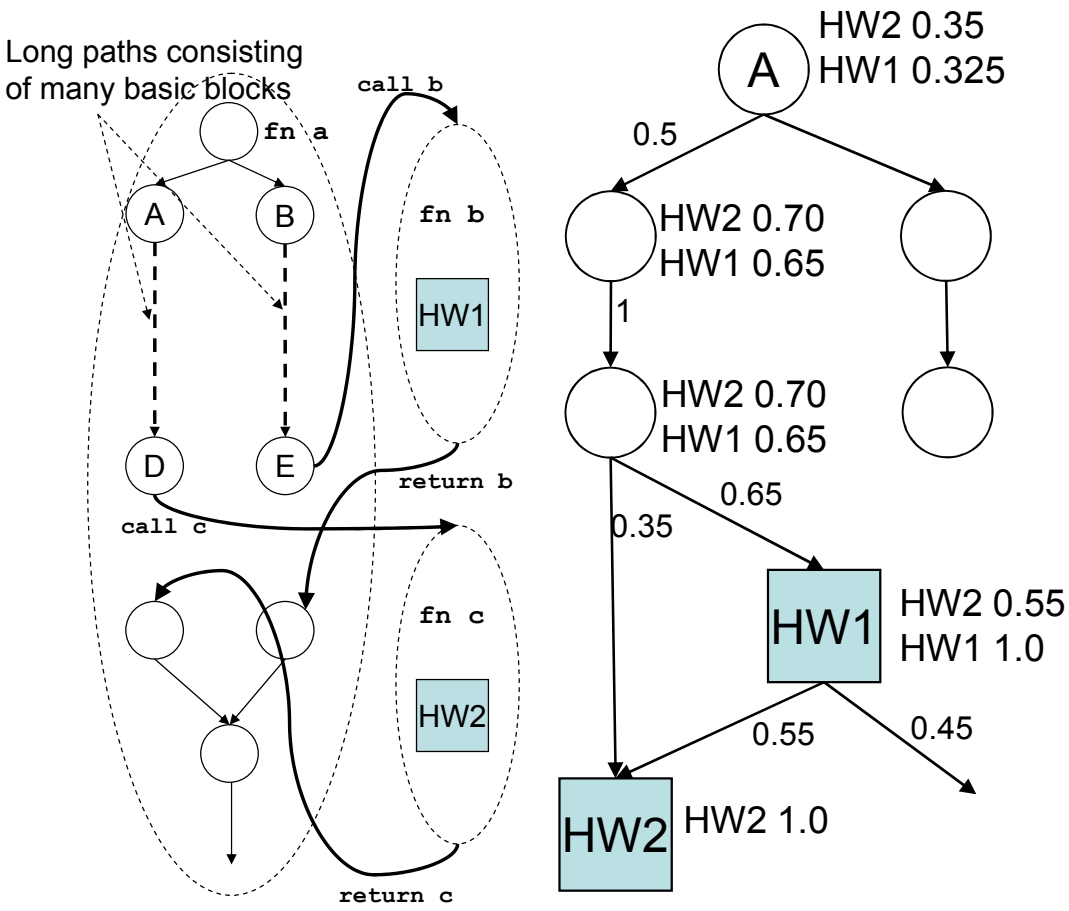


(c) Inserting prefetch loads increase execution time

Figure 5.4: How prefetching affects overall execution time.

To our knowledge, the two works that are most closely related to our work are done by Panainte et al. [50] and Li et al. [42]. Panainte proposed a static interprocedural analysis on call graphs that determined regions not shared between 2 conflicting hardware modules. It should be noted that their paper gave no details as to exactly which basic block should the load instructions be inserted. In our view, this approach could be too conservative and loses chances of hiding more configuration latency. Figure 5.5(a) shows the control flow graph of a function `a` will either call function `b` or `c`, depending on the branch taken at the beginning of the function. Function `b` and `c` will execute hardware modules `HW1` and `HW2` respectively. Given that `HW1` and `HW2` conflicts with each other, the approach by Panainte will not prefetch them beyond the boundaries of their respective functions (because the call graph loses detailed path information). However, basic blocks `A` and `B` are probably at least the safest earliest points where `HW1` and `HW2` can be prefetched.

Li proposed a probabilistic algorithm where a probability is attached to each edge in the control flow graph, to indicate how probable those edges will be taken, should their source be executed. After simplifying the control flow graph(that involves removing all cycles in the graph), the probability to reach each hardware reconfiguration can be computed by propagating the probabilities using a bottom-up approach. However, this approach is less satisfactory when applied to situations where there are placement conflicts between hardware modules. Figure 5.5(b) shows the case where the probability of reaching `HW1` and `HW2` are computed for basic block `A`. The probabilities indicate that `HW2` should be loaded first. Given that `HW1` and `HW2` conflicts with each other, we should load `HW1` first, contrary to what is indicated by the computed probabilities. This is because if we will reach `HW2` from basic block `A`, we are most likely to reach it through `HW1`. Furthremore, it is not known whether the removal of cycles in the CFG will lead to the loss of precious path



(a) Motivating example 1

(b) Motivating example 2

Figure 5.5: Motivating examples.

information. These examples show that both path and conflict information are important in order to improve the prefetching of configurations.

5.3 Problem Formulation

PROBLEM Given a directed, weighted ICFG $G = (V, E, C, I, U, HW)$, we would insert `load` calls into the ICFG so that a compiler system, together with the reconfiguration library support described in Section 5.1.2 will produce an executable that runs on the platform described in Section 5.1.1 so as to minimize the necessary reconfiguration overhead of

the application. One assumption that we make is every computation region is only either executed in software or hardware.

5.4 Interprocedural Placement-Aware Configuration Scheduling

The algorithm that we propose has the following 5 major stages:

1. Using profiling information, obtain the frequency of executing each control-flow edge and prune the edges accordingly.
2. Compute the *immediate post dominator* for every node.
3. Compute the *intra post dominator paths* (IPDP) i.e., paths that do not extend beyond the immediate post-dominator of the starting node of the path.
4. Compute for every node on the graph the estimated placement-aware probability of reaching each hardware node with the IPDP and post-dominator information using an iterative method.
5. Reduce the redundant prefetches and generate code for prefetching for each basic block.

In the first stage, we profile the application by inserting a profiling instrumentation code at the beginning of every basic block of the ICFG. By doing this, we are able to obtain the execution trace information and execution frequencies for each control-flow edge. In order to improve the efficiency of the algorithm, all edges with zero frequencies are removed at this stage. The weight function w for each edge is computed using the below equation:

$$w(e) = \frac{\text{frequency of edge } e}{\text{total frequency count of node } head(e)}$$

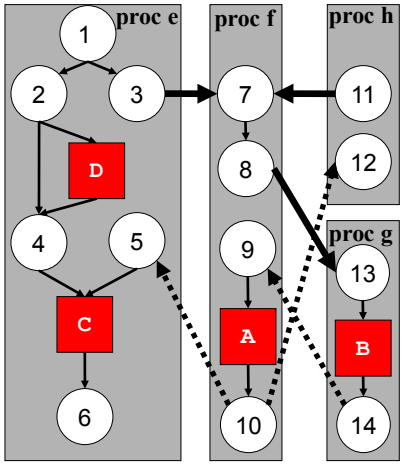


Figure 5.6: An example ICFG. The squares represent hardware nodes while ovals represent basic blocks. The thick edges represent call edges between procedures and the dotted lines represent the return edges from the procedures.

A node g of the CFG *post dominates* node v if every path from v to the exit node of their procedure passes through g. We denote the set of post-dominators for each node v to be $pdoms(v)$. For each node $v \in V - \{U\}$, there exists an immediate post-dominator g where $g \in pdoms(v)$ and $\nexists n \in pdom(v) : n \in pdom(g)$ (i.e., g post-dominates v but does not post-dominate any other post-dominators of v). We denote the immediate post-dominator of each v to be $ipdom(v)$. Classic algorithms [39] exist for obtaining the post dominator information. We proceed to describe steps 3 to 5 of the algorithm in more detail for the rest of this section.

5.4.1 Finding the Intra Post Dominator Paths

As mentioned above, intra post dominator paths are paths begins with a node v in the ICFG but never extends beyong the immediate post dominator of v. It is important for our algorithm to find these paths. Before defining what it is formally, we give an intuition as to why we require this information. Consider the ICFG shown in Figure 5.6 and procedure

e in particular. Suppose that the probability with which node 1 will reach the hardware node C is to be computed. A naive way of doing so would be to compute all possible paths between node 1 and C. Otherwise, we can observe that C is a postdominator of 1 and hence the probability of reaching C is 1. However, as hinted in section 5.2, this is insufficient as this probability is not *placement-aware*. If C were to conflict with D or A, we need to estimate the probability with which node 1 will reach C without encountering A or D on the way. Intuitively speaking, a node should have the same probability of reaching all hardware nodes as its immediate post dominator *provided it does not encounter conflicting ones* on all paths from it to its postdominator. In order to know whether this is the case, we need path information for every node before its immediate post-dominator.

Definition 5.4.1 Intra Post Dominator Paths (IPDP) *Given a ICFG* $G = (V, E, C, I, U, HW)$*, a path* p *of length* j *from node* m *to node* n *is a sequence of* j *edges, which will be denoted by* $[e_1, e_2, \ldots, e_j]$ *such that for all* $i, 1 \leq i \leq j-1$*,* $head(e_i) = tail(e_{i+1})$*. For convenience, we also denote that* $begin(P) = head(e_1)$ *and* $end(P) = tail(e_j)$*. Although* p *is a path and a sequence of edges, we abuse notation by referring to nodes along the path using the set notation. Hence,* $v \in p$ *means that node* v *occurs in path* p*. The estimated probability of taking a path* $P_{path}(p)$ *is the product of the weightage of the edges* $P_{path}(p) \prod_{i=1}^{j} w(e_i)$*. An IPDP* p *is a path as defined above with the following added properties: a)* $\forall v \in p, \nexists n \in p : n \neq v \wedge v \in pdom(n)$*. There does not exist any node along the path that is a post-dominator of any other node along the path and b)* $P_{path}(p) >$ threshold*, the estimated probability of this path being taken is greater than a threshold value. This threshold value is set to be* 0.0005 *in our experiments.*

Algorithm 2 shows the pseudo-code for how the IPDP information is computed for each node $v \in V$. The set of IPDPs for each node $v \in V$ is denoted as $IPDP_v$. The algorithm consists of two loops. In the first loop, we initialize $IPDP_v$ for each node v with the immediate

outgoing edges of v if the destination of the edge is not a post-dominator of v. The second loop is a classic working list algorithm loop, where outgoing edges of $end(p)$ are being added to the set $IPDP_v$ as long as the destination of the edge does not post-dominate any nodes in the path being concatenated to. It should be noted that the IPDP information does not extend beyond procedure boundaries (i.e., all paths leading to call sites or exit nodes terminate there).

Algorithm 2: Obtaining Intra-PostDominator Paths

```
Input: ICFG: (V, E, C, I, U, HW);
Result: Intra-PostDominator Paths Collected ∀v ∈ V
forall all nodes v ∈ V − U do
    forall all outgoing edges (v, s) ∈ E of v do
        if s is not a postdominator of v i.e. s ∉ pdoms(v) then
            initialize p with edge (v, s);
            insert path p into set IPDP_v;
forall all nodes v ∈ V − U do
    Change ← true;
    while Change do
        Change ← false;
        foreach path p ∈ IPDP_v do
            foreach all outgoing edges s : (end(p), s) ∈ E of end(p) do
                if s is either an exit node or call node i.e. s ∈ U ∨ s ∈ I then
                    continue;
                if ∀n ∈ p : s ∉ pdoms(n) then
                    Concatenate path p with edge (end(p), s) to get path p_new;
                    if probability of path p_new is higher than threshold then
                        Insert p_new into the set IPDP_v;
                        Change ← true;
```

5.4.2 Iterative Placement-Aware Estimated Probability Updating

As mentioned above, each node should have the same estimated probabilities reaching hardware nodes as its immediate post dominator. However, conflicting hardware nodes

may exist in a path between the node and its immediate post dominator. To avoid enumerating all possible paths (which may also be inter-procedural) between each node and its immediate postdominator, we compute the estimated probabilities of reaching each hardware node through a fixed point iterative process starting from the hardware nodes. Algorithm 3 shows a main loop that iterates through all the nodes in the graph during each iteration and continues doing so until a fixed-point (i.e., the estimated probabilities for each node have stabilized.) is reached. For this stage of the proposed algorithm, we maintain two two-dimensional vectors $IPDP\_Prob$ and P. $IPDP\_Prob(v,hw)$ maintains the estimated probabilities that a node v may reach a hardware node hw through its IPDP paths. $P(v,hw)$ maintains the estimated probabilities that a node v may reach a hardware node hw through all possible paths while $P(v)$ refers to the vector of estimated probabilities for node v. All $P(v,hw)$s are initialized to zeros except when $v = hw$, where $P(v,hw)$ is initialized to 1. A procedure may have multiple callers. Due to the uncertainty of the call context, we do not update the estimated probabilities for the exit nodes of the procedures.

We distinguish between the general case and call sites for the updating of estimated probabilities. Algorithm 4 shows how the estimated probabilities for a general node v is updated. The main thing is to compute a vector of estimated probabilities $temp\_p$ that will be used to update $P(v)$ if these 2 vectors are different. In the case when $P(v)$ is updated, a change is reported.

The computation of vector $temp\_p$ is done by computing a max_prob for each $hw \in HW$. During each iteration of the main loop, max_prob is the greater of two probabilities: One of them is the estimated probability of reaching hardware node hw through its IPDPs. This probability is given by $new\_prob = P\_path(p) \times P(end(p),hw)$ for path p. The other probability is the estimated probability of reaching hardware node hw through its post-dominator. This probability is given by $factor \times P(ipdom(v),hw)$ where $factor$

is computed by a summation of all the possibilities of reaching conflicting nodes of hw through its IPDPs, $factor = 1 - \sum_{hw' \sim hw} IPDP\_Prob(v, hw')$.

Algorithm 3: Iterative Probability Updating

```
Result: Final Placement-Aware Probabilities Computed For Each Basic Block
        ∀v ∈ V
forall v ∈ V do
    forall hw ∈ HW do
        IPDP_Prob(v, hw) ← 0;
        if v = hw then
            P(v, hw) ← 1;
        else
            P(v, hw) ← 0;
change ← true;
while change do
    change ← false;
    forall v ∈ V do
        if v is an exit node i.e. v ∈ U then
            continue;
        else if v is a call site i.e. v ∈ C then
            tmp_change ← update_probabilities_for_call_site(v);
        else
            tmp_change ← update_general_probabilities(v);
        if tmp_change then
            change ← true;
return P;
```

Furthermore, $IPDP\_Prob(v, hw)$ is updated whenever a larger estimated probability of reaching hw through a node's IPDP is found. We compute new_IPDP_prob by deducting $P(ipdom(v), hw)$ from new_prob. This is done to avoid double counting since it is possible

for the end of a IPDP path to reach a hardware node through the immediate post dominator of node v. *new_IPDP_prob* is used to update $IPDP\_Prob(v,hw)$ if it is a greater value.

Algorithm 4: update_general_probabilities(v)

```
Input: v
Result: Update probabilities for v based on post-dominator and IPDP_Prob
        information
change ← false;
forall hw ∈ HW do
    if v is hw or conflicts with hw i.e. v = hw ∨ v ≁ hw then
        continue;
    max_prob ← -1;
    forall p ∈ IPDP_v do
        if no nodes in p conflicts with hw (i.e. ∄n : n ∈ p ∧ n ≁ hw) then
            new_prob ← P_path(p) × P(end(p), hw);
            new_IPDP_prob ← new_prob − P(ipdom(v), hw);
            if new_IPDP_prob > IPDP_Prob(v, hw) then
                IPDP_Prob(v, hw) ← new_IPDP_prob;
            if new_prob > max_prob then
                max_prob ← new_prob;
    factor ← 1.0;
    forall hw' ∈ HW : hw' ≁ hw do
        factor ← factor − IPDP_Prob(v, hw');
    if factor × P(ipdom(v), hw) > max_prob then
        max_prob ← factor × P(ipdom(v), hw);
    if max_prob < threshold then
        temp_p(hw) ← 0;
    else
        temp_p(hw) ← max_prob;
if ∃ hw ∈ HW : temp_p(hw) ≠ P(v,hw) then
    change ← true;
    P(v) ← temp_p;
return change;
```

Similarly, in the case of updating the estimated probabilities for call sites, we compute a variable $temp\_p$ that will update $P(v)$ if these 2 vectors are different. The pseudo-code is given in Algorithm 5. Recall that $w(e)$ is the weight function for edge e, $temp\_p$ is

computed using the following equation in the first loop:

$$temp\_p(hw) = \sum_{\text{e} \in \text{out(v)}} w(e) \times P(tail(e), hw)$$

Here, $out(v)$ is the set of outgoing edges of node v. It should be noted that while we normally expect a call site to call only one callee, this is not generally true for call sites that make calls through procedure pointers. We rely on profiling results to determine which procedures are being called.

Algorithm 5: update_probabilities_for_call_site(v)

```
Input: v
Result: Update probabilities for call node v based on post-dominator and
        IPDP_Prob information
change ← false;
Initialize all values of array temp_p to 0;
forall outgoing edges e of v do
    foreach hw ∈ HW do
        temp_p(hw) ← temp_p(hw) + w(e) × P(tail(e), hw);
forall hw ∈ HW do
    if temp_p(hw) = 0 then
        val ← P(ipdom(v), hw);
        forall hw' ∈ HW : hw' ≁ hw do
            val ← val − temp_p(hw');
        temp_p(hw) ← val;
if ∃ hw ∈ HW : temp_p(hw) ≠ P(v,hw) then
    change ← true;
    P(v) ← temp_p;

return change;
```

We post-process the $temp\_p$ computed thus far by updating it with the estimated probabilities of the corresponding return site of v where needed. It should be noted that the corresponding return site of v will be its immediate post-dominator. $temp\_p(hw)$ will be updated when it is zero. We deduct the estimated probabilities of reaching the conflicting hardwares of hw in $temp\_hw$ from the estimated probability of reaching hw from the

return site $P(ipdom(v), hw)$. If this value is greater than zero, it will be used to update $temp\_p(hw)$.

5.4.3 Prefetch Reduction and Code Generation

Thus far, we have computed the estimated probabilities of reaching the hardware nodes from each basic block and the results are stored in the two-dimensional vector P. Consider a node v with its associated probability vector $P(v)$. We generate the prefetching code for node v using two code templates shown in Figure 5.7 and Figure 5.8 in pseudo-C code[1]. Firstly, we sort the hardware probabilities in descending order. After that, we insert the code template in Figure 5.8 into the beginning of node v based upon the sorted probabilities. It should be added that only hardware nodes that do not conflict with the most probable hardware are considered. In other words, we do not generate **load** calls for hardware nodes that conflict with the most probable hardware. Next, we fill in the inserted template with the loading template shown in Figure 5.7. The final condition for calling **load** for a hardware node is predicated upon a) the difference in probability between reaching this hardware node and its conflicting hardware node is small enough and b) whether the hardware node is already loaded. In cases where both conditions are true, we do not reconfigure the hardware node in question. In general, we will attempt to load the most probable reachable hardware node if it is not loaded and only consideer loading other hardware nodes if there are no hardware modules being reconfigured currently.

Naively, every basic block that has a non-zero probability of reaching a hardware node should be a candidate for inserting the **load** library call. However, this is needlessly expen-

[1] A compiler will insert these codes using low-level intermediate representation. We omit details here for the sake of brevity

```
//Suppose hardware nodes conflicting with hw
// are c0, c1 c2 ...
if( (P[v][hw]-P[v][c0]>THRESHOLD || !is_loaded(c0)) &&
   (P[v][hw]-P[v][c1]>THRESHOLD || !is loaded(c1)) && ...)
{
  load(hw);
}
```

Figure 5.7: Loading code template for hardware node hw. The condition for is expressed as a product of sums.

```
if(!is_loaded(most probable hardware node A))
   //Insert loading template for A
else if(currently reconfiguring()==-1)
{
   if(!is loaded(2nd most probable hardware node B))
     //Insert loading template for B
   else if(!is loaded(3rd most probable hardware node C))
     //Insert loading template for C
   else if(!is loaded(4th most probable hardware node D))
     //Insert loading template for D
}
```

Figure 5.8: Cascading ifs code template to be inserted at prefetch points

sive. The prefetch points can be reduced by clearing the probabilities for nodes where all its parents have the same probabilities of reaching the hardware nodes as itself.

5.5 Experimental Evaluation

5.5.1 Experimental Setup

We performed experiments with three applications `401.bzip2`, `429.mcf` and `h264enc` to study the effectiveness of our algorithm. `401.bzip2` and `429.mcf` were taken from the SPEC2006 benchmark suite [58]. `h264enc` was taken from the MediaBench II video benchmark suite [20]. `401.bzip2` is a block-sorting compression application while `429.mcf` is a program used for single-depot vehicle scheduling. `h264enc`[67] is an implementation

of H.264/AVC(Advanced Video Coding) encoder, the latest state-of-the-art video compression standard.

In our implementation of the architecture model, we employed the concept of ReCoBus by Koch et al[36] to support complex run-time reconfiguration. The ReCoBus's reconfiguration regions are organized in terms of reconfigurable slots i.e. the slots are the smallest granularity that the hardware modules will occupy on the pFPGA. The minimum size of each slot is 6 CLB Columns. Through profiling, we have identified 6 compute-intensive regions for `429.mcf` and `401.bzip2`. For `h264enc`, 7 such regions have been identified. These compute-intensive regions mapped to either basic blocks or loops in the original program. Table 5.1 shows these regions and the estimated number of slots (based on the software code size) that they occupy on the pFPGA. It has been assumed that the hardware performance is faster than software by 5 times for our experiments. We refer to the hardware regions by their indexes for the rest of this section.

For our experiments, we assumed a hardware device that has a similar geometry as Xilinx Virtex II Pro[70] FPGAs (i.e. column based), which is organized as a CLB matrix of 80 rows and 56 columns. The PowerPC CPU is operating at 300MHz. Every CLB column consists of 22 frames and each frame in turn requires 6,592 bits of configuration data. Thus, each CLB requires 145,024 bits of configuration data. Different FPGA architectures support different bit-widths of reconfiguration. For example, the Virtex IV Family supports a bitwidth of 32 bits for the SelectMap interface for the reconfiguration of the FPGA while the Virtex II Family supports a bitwidth of 8 bits. We assumed a single reconfiguration port running at 100MHz and performed experiments for reconfiguration bitwidths 8 and 32. Table 5.2 shows the different reconfiguration overheads of reconfiguring a single ReCoBus slot for different bitwidths. Obviously, the wider the bitwidth the lower the overhead.

Benchmark	Index	**Region from Procedure**	**No. Of Slots**
401.bzip2	B0	mainQSort3	2
401.bzip2	B1	fallbackSort	3
401.bzip2	B2	copy_input_until_stop	3
401.bzip2	B3	generateMTFValues	2
401.bzip2	B4	mainSort	1
401.bzip2	B5	mainSimpleSort	2
h264enc	H0	FastFullPelBlockMotionSearch	2
h264enc	H1	SetupFastFullPelSearch	2
h264enc	H2	SATD	2
h264enc	H3	writeRunLevel_CABAC	2
h264enc	H4	biari_encode_symbol	2
h264enc	H5	dct_luma	3
h264enc	H6	dct_luma	1
429.mcf	M0	primal_bea_mpp	2
429.mcf	M1	price_out_impl	2
429.mcf	M2	sort_basket	2
429.mcf	M3	refresh_potential	2
429.mcf	M4	primal_iminus	3
429.mcf	M5	primal_bea_mpp	2

Table 5.1: The regions selected for hardware implementation in the h264enc and 429.mcf benchmarks.

Bit Widths	Reconfiguration Overhead for 1 ReCoBus Slot (PowerPC cycles at 300MHz)
8	$\frac{6592}{8} \times \frac{100}{300} \times 22 \times 3 = 326304$ cycles
32	81576

Table 5.2: Reconfiguration Overhead of 1 ReCoBus Slot for different bit-widths.

The benchmarks were compiled using the Open IMPACT compiler[48]. While Open Impact was targeted for the Itanium machine[32], we made changes so that the compiler backend generated code for PowerPC 405[74] instead. The CPU cores embedded in Xilinx Virtex II Pro chips are of the PowerPC 405 model. The changes made enabled us to compile applications that targets the Xilinx FPGA platforms. Information such as the control-flow graph and basic block IDs were obtained from the Open IMPACT compiler.

Through code instrumentation, we were able to obtain a trace of basic block IDs from the execution of the application and measure the average execution time for each basic block. The average execution time for the basic blocks of the `h264enc` application was measured by running the instrumented code on the Xilinx University Program Board[68]. For `429.mcf` and `401.bzip2`, the measurements were taken by running the instrumented code on a PowerPC machine and the execution times were later scaled back to match the execution frequency of 300MHz. An inhouse developed trace-based simulation used the trace and the execution time information to compute the expected execution time. Finally, we compared the performance of our algorithm by comparing it against three scenarios: fetch-on-demand(FOD), optimal and the placement-blind probabilistic algorithm.

Fetch-On-Demand: The Fetch-On-Demand schedule has been described earlier in Section 5.2. Basically, there are no load library calls being made at all. The hardware modules

are loaded onto the pFPGA if they are encountered during execution and if it is not already loaded onto the pFPGA. It is reasonable to expect that any prefetching approach to improve upon this case. We used the expected execution time of the Fetch-On-Demand scenario as the baseline for comparison in our experiments.

Optimal: The optimal case is when the entire execution trace is already known beforehand and every prefetching decision is made based upon this foreknowledge. Our implementation of this scenario relied on the algorithm described in [55]. We do not expect a static approach described in this chapter to be as good as the optimal case, but the gap between the Optimal and Fetch-On-Demand is useful for gauging the effectiveness of our approach.

Placement-blind Probabilistic algorithm: The implementation of the placement-blind probability algorithm is based on [42]. Some changes such as identifying back-edges and removing them need to be made to the control flow graph before we use this algorithm. Basically, it is a bottom up approach of propagating the probability of reaching the hardware nodes. This technique is developed for relocatable and defragmentable FPGAs and not for the Xilinx FPGA architectures. Therefore, this approach does not account for the placement conflicts between the hardware modules and serves as a good gauge of what happens when we are not placement-aware.

5.5.2 Experimental Results

The placement of the hardware modules determines the conflict relationships between them. To evaluate the effect of different conflict sets for our algorithm, we generate different placements for the selected regions in Table 5.1 so that the number of conflicts/overlap

between the hardware modules is minimized. We omit the placement details and instead abstract them by showing the different conflicts in Table Table 5.3.

Placement Labels	**Conflicts**
bzip2-s3-1	$\{B0 \nsim B1, B1 \nsim B3, B0, \nsim B3, B2 \nsim B4, B2 \nsim B5\}$
bzip2-s3-2	$\{B1 \nsim B4, B1 \nsim B5, B2, \nsim B3, B2 \nsim B0, B0 \nsim B3\}$
bzip2-s3-3	$\{B1 \nsim B4, B0 \nsim B1, B2, \nsim B3, B2 \nsim B5, B3 \nsim B5\}$
bzip2-s4-1	$\{B1 \nsim B2, B0 \nsim B5\}$
bzip2-s4-2	$\{B1 \nsim B2, B0 \nsim B3\}$
bzip2-s4-3	$\{B1 \nsim B2, B3 \nsim B5\}$
h264-s3-1	$\{H0 \nsim H5, H3 \nsim H5, H0 \nsim H3, H1 \nsim H4, H2 \nsim H4, H1 \nsim H2\}$
h264-s3-2	$\{H4 \nsim H5, H1 \nsim H4, H1 \nsim H5, H0 \nsim H2, H0 \nsim H3, H2 \nsim H3\}$
h264-s3-3	$\{H0 \nsim H3, H0 \nsim H1, H1 \nsim H3, H4 \nsim H5, H2 \nsim H5, H2 \nsim H4\}$
h264-s4-1	$\{H1 \nsim H3, H0 \nsim H5, H2 \nsim H4\}$
h264-s4-2	$\{H1 \nsim H5, H0 \nsim H3, H2 \nsim H4\}$
h264-s3-3	$\{H1 \nsim H4, H0 \nsim H3, H2 \nsim H5\}$
h264-s3-4	$\{H1 \nsim H2, H0 \nsim H3, H4 \nsim H5\}$
mcf-s3-1	$\{M0 \nsim M1, M0 \nsim M2, M1 \nsim M2, M5 \nsim M3, M5 \nsim M4, M3 \nsim M4\}$
mcf-s3-2	$\{M5 \nsim M1, M5 \nsim M2, M1 \nsim M2, M0 \nsim M3, M0 \nsim M4, M3 \nsim M4\}$
mcf-s3-3	$\{M5 \nsim M1, M5 \nsim M3, M1 \nsim M3, M0 \nsim M2, M0 \nsim M4, M2 \nsim M4\}$
mcf-s3-4	$\{M5 \nsim M2, M5 \nsim M3, M2 \nsim M3, M0 \nsim M1, M0 \nsim M4, M1 \nsim M4\}$
mcf-s4-1	$\{M1 \nsim M5, M4 \nsim M3, M4 \nsim M2\}$
mcf-s4-2	$\{M0 \nsim M4, M5 \nsim M4, M2 \nsim M3\}$
mcf-s4-3	$\{M4 \nsim M1, M4 \nsim M2, M3 \nsim M5\}$
mcf-s4-4	$\{M4 \nsim M1, M4 \nsim M3, M2 \nsim M5\}$

Table 5.3: Benchmarks with different placements.

The labels in Table 5.3.for each different placement bear some explanations. All the labels are named after the corresponding applications. Specifically, labels starting with `bzip2-` refers to placements for `401.bzip2`. Labels starting with `h264-` refers to placements for `h264enc` and labels starting with `mcf` refers to placements for `429.mcf`. The placements that are labeled with 's3' are placements generated for a ReCoBus implementation with 2 separate Reconfigurable Region of 3 configurable slots while placements labeled with 's4' are generated for a ReCoBus implementation with 2 separate Reconfigurable Region of configurable 4 slots. Each of these placements form a separate test case for our experiments. Obviously, we observe that the number of conflicts decreases when the amount of resources available increases.

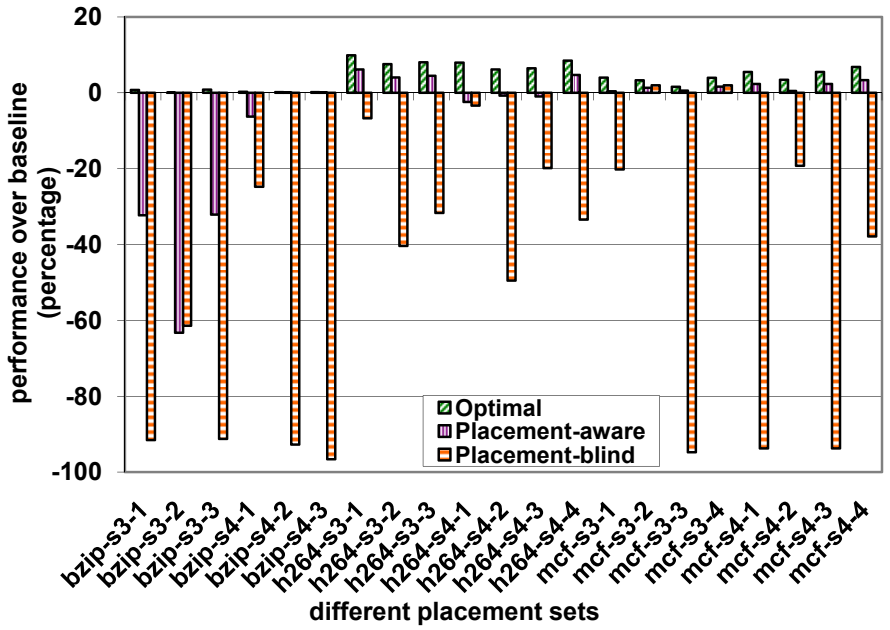


Figure 5.9: Speedups over baseline for 8-bits wide reconfiguration port running at 100MHz.

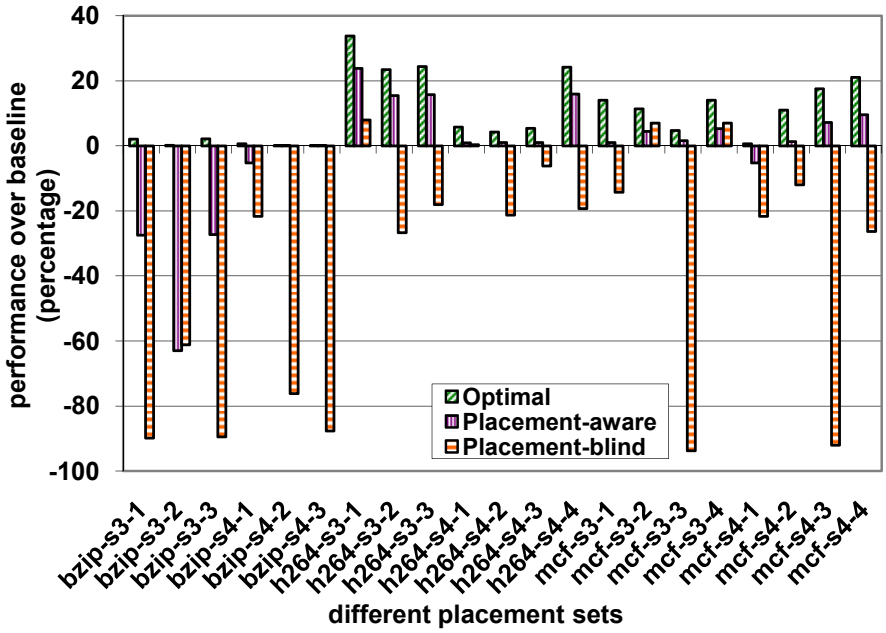


Figure 5.10: Speedups over baseline for 32-bits wide reconfiguration port running at 100MHz.

Figure 5.9 and 5.10 show us the various speedups/slowdown over the baseline for reconfiguration bit-widths of 8 bits and 32 bits respectively, after applying the placement-blind probability, optimal and our placement aware algorithm to the various placement sets. We make the following observation of the results shown:

- All algorithms performs better in the case when reconfiguration port's bitwidth is 32 bits. This shows that higher reconfiguration speeds creates more temporal space during execution for prefetch to occur.

- The performance of our algorithm is worse in placements for `401.bzip2`. However, we observe that the gap between the optimal and baseline is almost negligiable for `401.bzip2`. This implies that there is not much space for the execution to be optimized through configuration prefetching. However, our algorithm still manages to perform better than baseline in two of the test cases for `401.bzip2` despite the nar-

row gap. It should also be observed that the size of the gaps between baseline and optimal is dependent on the Hardware-Software partitioning of the application. In this case, the partitioning for `401.bzip2` is not ideal in the first place.

- We observe that performance degrades seriously for when conflicts are not taken into account. The placement-blind probability suffers a maximum of 90% slowdown and a 20% degradation in performance for most of the placement sets tested in our experiments. This shows the inadequacy of the placement-blind algorithm for the pFPGA architecture we are targeting.

- For the same benchmark, the speedup that can be gained through configuration scheduling differs across varying placement sets. In particular, `h264-s3-1` is the best for `h264enc`, achieving a speedup of almost more than 30% for the optimal algorithm. This shows how the placements affect both the overall performance and the opportunities available for configuration prefetching.

Another way to measure the quality of our algorithm is by showing how close the result of our algorithm is to the optimal when it performs better than baseline. To do this, we compute what is called *optimal proximity score* by

$$\text{Optimal Proximity Score} = \frac{\text{Performance increase of placement-aware algorithm}}{\text{Performance increase of optimal algorithm}} \times 100$$

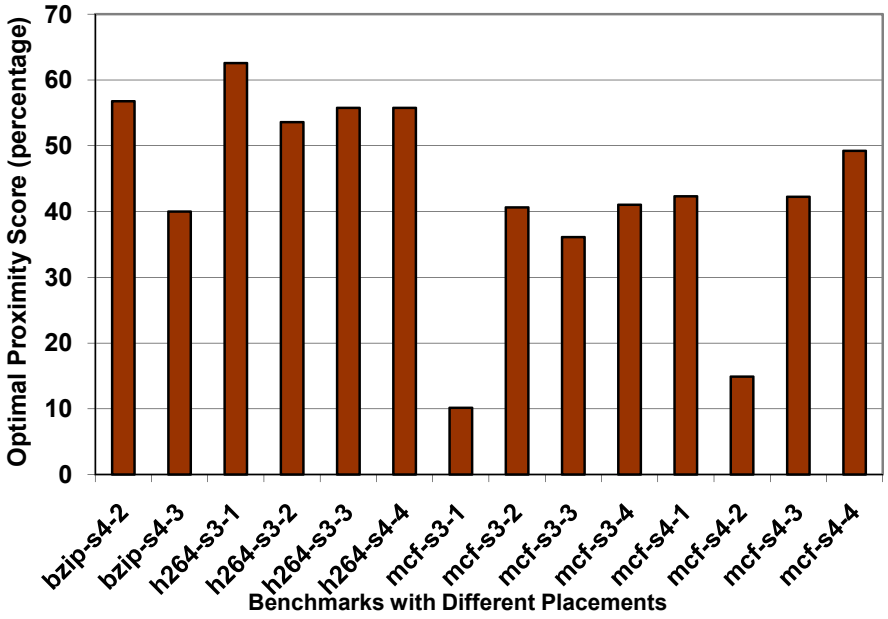


Figure 5.11: Proximity to optimal by normalizing the range between baseline and optimal (8 bits wide reconfiguration port).

The optimal proximity score shows where the results of the placement-aware algorithm falls within the normalized range between the baseline and the optimal. Figure 5.11 and 5.12 show the optimal proximity for 8 bits and 32 bits wide reconfiguration ports respectively. Higher optimal proximity score indicates better proximity to the optimal. For example, in the case when the reconfiguration port is 32 bits wide, `h264-s3-1` has a score of 72% while `h264-s4-1` has a score of 17%. This shows that the result of `h264-s3-1` is much closer to optimal compared to `h264-s4-1`.

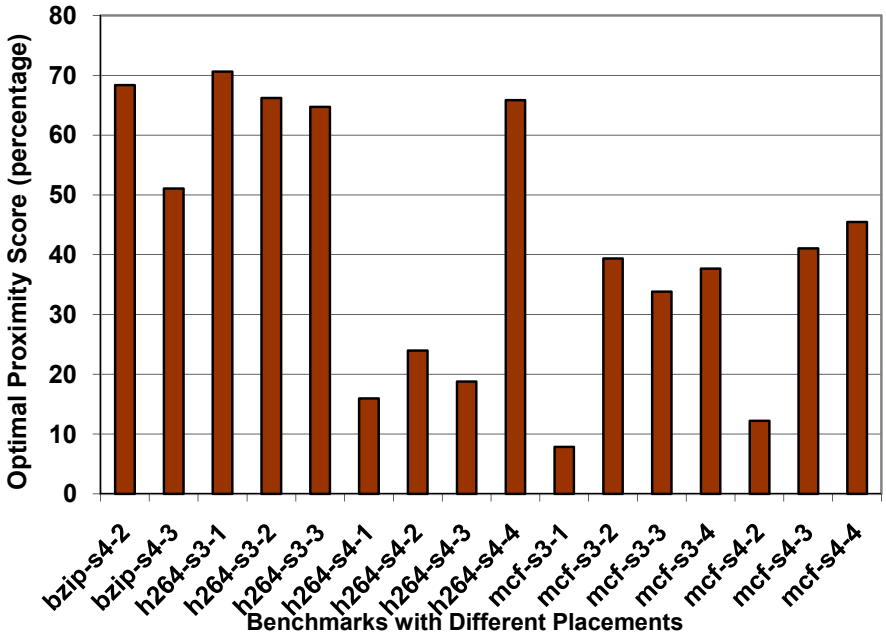


Figure 5.12: Proximity to optimal by normalizing the range between baseline and optimal (32 bits wide reconfiguration port).

5.6 Summary

In this chapter, we have described a novel method that statically determines for each basic block what it ought to pre-load into the FPGA so as to reduce the reconfiguration overhead. Our approach is consistently better than baseline and performs better than state-of-the-art prefetching algorithms based upon static analyses. However, our experiments show that there is still room for improvement in our approach. For a static approach, it is important to avoid being too conservative and too speculative at the same time. The former will lead to less reconfiguration latency hiding while the latter will cause mis-prefetches that may increase the number of reconfigurations initiated. A better approach would need to sensitive to the context of the execution i.e. the code becomes 'aware' of the phase it is executing in and prefetches according to the probabilities estimated for that phase instead.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this thesis, we have studied the hardware software co-design for FPGA-based systems with the aim of improving overall execution time by reducing run-time reconfiguration overhead. This overhead can potentially wipe out any speed up obtained by implementing the computation in a FPGA. This consideration and the opportunities afforded by the advances in architectural support for more efficient reconfiguration form the motivation of this thesis. The main contributions of this thesis are as follows:

- In Chapter 3, we presented a framework for the efficient implementation of neighborhood searches of the temporal and spatial partitioning design space. It is demonstrated in this chapter that both temporal and spatial partitioning affects the number of reconfigurations, thus making it difficult to estimate the run-time reconfiguration overhead incurred by a particular design point. A naive solution would be to scan through the entire trace every time a design point is evaluated. Apart from the intractable size of the design space, this solution does not scale with increasing length

of the execution trace. Our framework provides an efficient way of computing the run-time reconfiguration cost through (a) a novel definition of the neighboring relationship between design points, (b) using a loop trace encoded with `SEQUITUR` grammar, and (c) an algorithm that leverages the former two to compute the change in run-time reconfiguration cost when moving between 2 neighboring points. This way, once an initial computation of the reconfiguration cost is known for one design point, we can efficiently compute the reconfiguration cost for all its neighbors and transitively, for the rest of the design space as well when required. We evaluated the efficiency of this framework with the implementation of 2 neighborhood searches, namely hill climbing and Tabu search using this framework. Our experiments showed that hill climbing is able to find the optimal design point more than 90% of the cases while Tabu search found the optimal design point in all of our experiments. It was also shown that the searches were sped up by up to *two orders of magnitude* when the proposed framework is employed.

- While the framework presented in Chapther 3 allows the design space of both temporal and spatial partitioning to be searched, it does not consider the possibility of further configuration overhead reduction on pFPGAs. In Chapter 4, we examined the following sub-problem: Given an execution trace, the associated hardware modules and their placements on a pFPGA, find the optimally feasible schedule that minimizes the overall execution time. This is a complementary, orthogonal problem to the one solved in Chapter 3. To solve this problem, we present a novel, polynomial time algorithm that solves this problem by scheduling the reconfiguration of hardware modules to occur in parallel with application execution whenever possible. The resultant schedule is shown to be provably optimal. A key to the algorithm is a dependence analysis that determines whether for each instance of the hardware module

execution, a prior reconfiguration is needed. Experiments performed using the H.264 benchmark shows that the current state-of-the-art online prefetching algorithms perform considerably worse than the schedule returned by our algorithm. The difference between the two in terms of speedup over the baseline can be as large as 40%.

- Although the algorithm in Chapter 4 returns an optimal schedule for a particular execution trace, a program's execution pattern may differ with varying inputs. It is not feasible to schedule for every possible input of a program. In Chapter 5, we examined the following sub-problem: Given a program that is represented in an interprocedural control flow graph, together with the hardware modules and their associated placements on a pFPGA, find suitable prefetch points in the graph for the insertion of library calls that will load the hardware modules ahead of time so that overall execution time is minimized. By making use of profiled execution frequencies of control-flow edges, our proposed novel algorithm solves this problem through an iterative approach that estimates placement-aware probabilities of reaching hardware execution for each basic block. Placement-aware probability refers to the probability of reaching the execution of a hardware module without encountering conflicting modules on the path of the interprocedural control flow graph. Experiments show that our proposed algorithm makes significant improvements over state-of-the-art prefetching strategies that do not consider placement conflicts.

6.2 Future Works

6.2.1 Granularity of Reconfiguration and Configuration Scheduling

To our knowledge, there have been no previous work that discusses how the possibility of splitting a single reconfiguration of a hardware module into distinct temporal phases affects configuration scheduling. The work presented in this thesis has assumed that the cost of pre-emption and resumption to be minimal compared with the entire reconfiguration overhead. However, in practice, reconfiguration needs to occur in multiple of frames and there is a setup-cost involved in initiating a reconfiguration. For example, it is possible to break down the configuration of a hardware module consisting of ten frames into ten distinct stages (i.e., during each stage, only one frame is loaded.). While the flexibility of scheduling the configuration of this hardware module has increased, the overall configuration overhead has increased as well because a setup cost hsa to be paid for each of the ten stages. However, reconfiguring at a larger granularity will result in a loss of this flexibility. Therefore, the granularity of reconfiguration affects the problem of configuration scheduling and this should be considered in future works.

6.2.2 Hardware-Software Co-Placement and Partitioning

Both Chapters 4 and 5 have assumed that the hardware partitioning is already done and the placements of the hardware modules are decided beforehand. The focus was on the relative speedup between the fetch-on-demand schedule and the desired optimized schedule. This problem is orthogonal to that of selecting a suitable conflict set so that overall execution time is minimized. Our experimental data from these studies show that the difference in execution time between 2 different sets of conflicts could be as large as 60%. The problem of hardware-software co-placement and partitioning could be expressed as follows: Given a

single sequential program and its constituent compute-intensive regions, how do we decide which of these regions should be implemented in hardware? For the selected hardware implementations, how shall we place these hardware modules on the FPGA so that the conflict relationships between them will minimize the run-time reconfiguration overhead? Obviously, these two questions are inter-related and need to be answered to obtain a quality solution that minimizes the overall execution time. Therefore it makes sense to combine the two into a unified co-design problem.

6.2.3 Configuration Management for Multi-core Reconfigurable Computing

This work so far has concentrated on architecture models that consist of a single CPU and a single FPGA co-processor. However, given the advent of multi-core architectures like FSB-FPGA[33], the challenge would be for general purpose programs to harness the potential speedup possible from the attached reconfigurable device. Although there are ongoing research in this area[29], many open problems remain unsolved especially in the domain of general purpose reconfigurable computing, where the set of applications running in the system is dynamically changing according to the demands of the users.

Bibliography

[1] J. M. Arnold. S5: the architecture and development flow of a software configurable processor. In *ICFPT '05, Proceedings of International Conference on Field-Programmable Technology 2005*, pages 121–128, Singapore, Dec. 2005. IEEE.

[2] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, Mar. 1993.

[3] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 335–340, New York, NY, USA, 2005. ACM Press.

[4] S. Banerjee, E. Bozorgzadeh, N. Dutt, and J. Noguera. Selective bandwidth and resource management in scheduling for dynamically reconfigurable architectures. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 771–776, New York, NY, USA, 2007. ACM.

[5] K. Bondalapati and V. K. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90:1201–1217, 2002.

[6] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architecture*. PhD thesis, University of California at Berkeley, Berkeley, California, United States, 2002.

[7] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, 2000.

[8] Celoxica Ltd., Oxfordshire, UK. *DK3: Handel-C Language Reference Manual*, 2002.

[9] L. N. Chakrapani, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In *LCPC '04, Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, volume 3602, pages 32–41. Springer, 2005.

[10] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *FPL '99: 9th International Workshop on Field-Programmable Logic and Applications*, pages 175–184. Springer, 1999.

[11] Y. Chen and S. Y. Chen. Cost-driven hybrid configuration prefetching for partial reconfigurable coprocessor. In *RAW '07: Proceedings of International Parallel and Distributed Processing Symposium Reconfigurable Architectures Workshop (RAW)*, pages 194–200, Los Alamitos, CA, USA, 2007. IEEE.

[12] C. Claus, F. H. Müller, J. Zeppenfeld, and W. Stechele. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In *RAW '07: Proceedings of International Parallel and Distributed Processing Symposium Reconfigurable Architectures Workshop (RAW)*, pages 1–7, Los Alamitos, CA, USA, 2007. IEEE.

[13] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for reconfigurable computing. In *FCCM '00: Proceedings of the 8th*

IEEE Symposium on Field-Programmable Custom Computing Machines, page 279, Washington, DC, USA, 2000. IEEE Computer Society.

[14] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, 2002.

[15] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems*, 1(4):1–23, 2009.

[16] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Kluwer Journal on Design Automation For Embedded Systems*, 2(1):5 –32, 1997.

[17] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. pages 18–29, 2002.

[18] S. P. Fekete, J. C. van der Veen, J. Angermeier, C. Göhringer, M. Majer, and J. Teich. Scheduling and communication-aware mapping of HW/SW modules for dynamically and partially reconfigurable SoC architectures. In *ARCS '07: 20th International Conference on Architecture of Computing Systems 2007*, pages 151–160. VDE-Verlag, Berlin, 2007.

[19] T. Feo and M. Resende. Greedy randomized adaptive search procedures. In *Journal of Global Optimization*, volume 6, pages 109–133, 1995.

[20] 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*, 33(4):301–318, 2009.

[21] W. Fu and K. Compton. An execution environment for reconfigurable computing. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–158, Washington, DC, USA, 2005. IEEE Computer Society.

[22] S. Ganesan and R. Vemuri. An integrated temporal partioning and partial reconfiguration technique for design latency improvement. In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 320–325, New York, NY, USA, 2000. ACM Press.

[23] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh. An optimal algorithm for minimizing run-time reconfiguration delay. *ACM Transactions in Embedded Computing Systems*, 3(2):237–256, 2004.

[24] F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing.

[25] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 11–20, New York, NY, USA, 1998. ACM.

[26] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 65–74, New York, NY, USA, 1998. ACM.

[27] J. R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based*

Custom Computing Machines, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[28] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. Wilton. Floating-point FPGA: Architecture and modeling. *IEEE Transactions on Very Large Scale Integration [VLSI] Systems*, 17(12):1709–1718, Dec. 2009.

[29] C. Huang and F. Vahid. Dynamic coprocessor management for FPGA-enhanced compute platforms. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 71–78, New York, NY, USA, 2008. ACM.

[30] H. P. Huynh, J. E. Sim, and T. Mitra. An efficient framework for dynamic reconfiguration of instruction-set customization. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 135–144, New York, NY, USA, 2007. ACM.

[31] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993.

[32] Intel Corp. *Intel Itanium 2 Processor Reference Manual for Software Development, Document Number 251110-001*, June 2002.

[33] Intel Corp. *Intel Quickassist Technology*, 2008. http://www.intel.com/technology/platforms/quickassist/index.htm.

[34] A. Kalavade and E. A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of*

the 3rd International Workshop on Hardware/Software Codesign, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[35] D. Koch, C. Beckhoff, and J. Teich. Bitstream decompression for high speed fpga configuration from slow memories. In *ICFPT '07, Proceedings of International Conference on Field-Programmable Technology 2007*, pages 161–168, Kokurakita, Kitakyushu, JAPAN, Dec. 2007. IEEE.

[36] D. Koch, C. Beckhoff, and J. Teich. A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 fpgas. In *FPGA '09: Proceeding of the 2009 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 253–256, New York, NY, USA, 2009. ACM.

[37] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms Generation, Enumeration and Search*. CRC Press Inc, 1998.

[38] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[39] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[40] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 507–512, New York, NY, USA, 2000. ACM.

[41] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *FCCM '00: Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines*, page 22, Washington, DC, USA, 2000. IEEE Computer Society.

[42] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field Programmable Gate Arrays*, pages 187–195, New York, NY, USA, 2002. ACM.

[43] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited paper: Enhanced architectures, design methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *FPL '06: Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, 2006.

[44] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: a benchmarking suite for network processors. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42, Piscataway, NJ, USA, 2001. IEEE Press.

[45] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[46] J. Noguera and R. Badia. A hw/sw partitioning algorithm for dynamically reconfigurable architectures. In *DATE '01: Proceedings of the Conference on Design, Automation and Test in Europe*, page 729, Piscataway, NJ, USA, 2001. IEEE Press.

[47] J. Noguera and R. M. Badia. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *ACM Transactions on Embedded Computing Systems*, 3(2):385–406, 2004.

[48] UIUC Open IMPACT Effort, "The Open IMPACT IA-64 Compiler." http://gelato.uiuc.edu.

[49] E. M. Panainte, K. Bertels, and S. Vassiliadis. Instruction scheduling for dynamic hardware configurations. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 100–105, Washington, DC, USA, 2005. IEEE Computer Society.

[50] E. M. Panainte, K. Bertels, and S. Vassiliadis. Interprocedural compiler optimization for partial run-time reconfiguration. *Journal of VLSI Signal Processing Systems*, 43(2-3):161–172, 2006.

[51] R. N. Pittman, N. L. Lynch, R. Forin, R. N. Pittman, N. L. Lynch, and R. Forin. eMIPS, a dynamically extensible processor. Technical report, Microsoft Research, Redmond, WA, United States, 2006.

[52] D. N. Rakhmatov and S. B. K. Vrudhula. Hardware-software bipartitioning for dynamically reconfigurable systems. In *CODES '02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 145–150, New York, NY, USA, 2002. ACM.

[53] F. Redaelli, M. D. Santambrogio, and D. Sciuto. Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 519–522, New York, NY, USA, 2008. ACM.

[54] J. Resano, D. Mozos, and F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 106–111, Washington, DC, USA, 2005. IEEE Computer Society.

[55] J. E. Sim, W. F. Wong, and J. Teich. Optimal placement-aware trace-based scheduling of hardware reconfigurations for fpga accelerators. In *FCCM '09: Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 279–282, Napa, CA, USA, Apr. 2009. IEEE Computer Society.

[56] S. Singh, J. Rose, P. Chow, and D. Lewis. The effect of logic block architecture on FPGA performance. *Solid-State Circuits, IEEE Journal of*, 27(3):281–287, Mar. 1992.

[57] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 165–176, New York, NY, USA, 2002. ACM.

[58] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmark Suite, 2006. http://www.spec.org/cpu2006.

[59] C. Steiger, H. Walder, M. Platzner, and L. Thiele. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 224, Washington, DC, USA, 2003. IEEE Computer Society.

[60] G. Stitt, F. Vahid, and S. Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *ACM Transaction on Embedded Computing Sys.*, 3(1):218–232, Feb. 2004.

[61] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 189–198, New York, NY, USA, 2003. ACM.

[62] S. Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, New York University, New York, United States, 2001.

[63] X. Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration latency in chameleon processors. In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 29–38, London, UK, 2000. Springer-Verlag.

[64] J. Teich, S. P. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The Journal of Supercomputing*, 19(1):57–75, 2001.

[65] F. Vahid. Modifying min-cut for hardware and software functional partitioning. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, page 43, Washington, DC, USA, 1997. IEEE Computer Society.

[66] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[67] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[68] Xilinx Inc., San Jose, CA, United States. *Xilinx Corp, XUP Virtex II Pro Development System*. Avaliable at http://www.xilinx.com.

[69] Xilinx Inc., San Jose, CA, United States. *Xilinx Virtex-E 1.8 V Field-Programmable Gate Arrays DataSheet*. Available at http://www.xilinx.com.

[70] Xilinx Inc., San Jose, CA, United States. *Xilinx Virtex-II Pro Platform FPGAs: complete data sheet*. Available at http://www.xilinx.com.

[71] Xilinx Inc., San Jose, CA, United States. *Virtex-II 1.5V Field-Programmable Gate Arrays*, Nov. 2001. Available from http://www.xilinx.com.

[72] Xilinx Inc., San Jose, CA, United States. *XC2V6000 data sheet*, 2001. Available from http://www.xilinx.com.

[73] Xilinx Inc., San Jose, CA, United States. *Virtex Series Configuration Architecture User Guide (XAPP151 v1.7)*, Oct. 2004. Available from http://www.xilinx.com.

[74] Xilinx Inc., San Jose, CA, United States. *PowerPC 405 Processor Block Reference Guide*, July 2005. Available at http://www.xilinx.com.

[75] Xilinx Inc., San Jose, CA, United States. *Virtex-4 User Guide (UG070 v2.6)*, 2008. Available from http://www.xilinx.com.

[76] Xilinx Inc., San Jose, CA, United States. *Virtex-6 Family Overview (v2.1)*, 2009. Available from http://www.xilinx.com/.

[77] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 225–235, New York, NY, USA, 2000. ACM.