# AUTOMATIC DESIGN OF EFFICIENT APPLICATION-CENTRIC ARCHITECTURES

by

**Kevin C. Fan**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:
Associate Professor Scott Mahlke, Chair
Professor Stéphane Lafortune
Professor Trevor N. Mudge
Associate Professor Todd M. Austin

# ACKNOWLEDGEMENTS

This dissertation would not have been possible without the guidance and support of many people. First and foremost, I would like to thank my advisor, Scott Mahlke. His insight, expertise, enthusiasm, and encouragement played a large part in my success in graduate school. Without his guidance, this dissertation would not exist. In addition, Scott was one of the first to encourage me to undertake graduate studies in the first place after I worked with him as an undergraduate.

I would also like to thank my thesis committee, Professors Trevor Mudge, Todd Austin, and Stéphane Lafortune. They donated their time, giving valuable comments and suggestions to help improve the research and refine the thesis. In addition, I would like to thank Bill Mangione-Smith, who first exposed me to compilers and to graduate school when I worked with him at UCLA.

The research presented in this dissertation was not the work of one person; I was fortunate to have the assistance of a number of other students in the Compilers Creating Custom Processors research group. Manjunath Kudlur provided significant help with the ILP and SMT scheduling formulations presented in this dissertation. Hyunchul Park assisted with creating the Verilog back-end of the synthesis system.

Ganesh Dasika implemented the Verilog simulation framework and also answered my numerous questions about the area and power analysis tools. In addition, my work is based on the Trimaran compiler infrastructure; Mike Chu, Nate Clark, Rajiv Ravindran, and Hongtao Zhong, among others, have done significant work in maintaining and improving this infrastructure.

Outside of the technical reasons, I appreciate the opportunity to have worked with a great group of people who provided moral support and made my graduate school experience enjoyable, namely: Amin Ansari, Jay Blome, Mike Chu, Nate Clark, Ganesh Dasika, Shuguang Feng, Shantanu Gupta, Jeff Hao, Amir Hormati, Po-Chun Hsu, Manjunath Kudlur, Steve Lieberman, Yuan Lin, Mojtaba Mehrara, Rob Mullenix, Pracheeti Nagarkar, Hyunchul Park, Yongjun Park, Rajiv Ravindran, Misha Smelyanskiy, and Hongtao Zhong. I have shared offices, and in many cases, homes with these friends and my time in Ann Arbor would not have been the same without them.

I would like to thank my family for their support, encouragement, and advice. Finally, and most importantly, I thank Jennifer Mato for her unconditional love and support over the years, even through separation over long distances and long periods of time. Her companionship added invaluable depth to my years in graduate school and has made its completion all the more sweet.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# ABSTRACT

AUTOMATIC DESIGN OF EFFICIENT
APPLICATION-CENTRIC ARCHITECTURES

by

Kevin C. Fan

Chair: Scott Mahlke

As the market for embedded devices continues to grow, the demand for high performance, low cost, and low power computation grows as well. Many embedded applications perform computationally intensive tasks such as processing streaming video or audio, wireless communication, or speech recognition. Often, performance requirements are on the order of 10-100 billion operations per second and must be implemented within tight power budgets on the order of 100 mW. Typically, general purpose processors are not able to meet these performance and power requirements. Custom hardware in the form of loop accelerators are often used to execute the compute-intensive portions of these applications because they can achieve significantly higher levels of performance and power efficiency.

Automated hardware synthesis from high level specifications is a key technology used in designing these accelerators, because the resulting hardware is correct by construction, easing verification and greatly decreasing time-to-market in the quickly evolving embedded domain. In this dissertation, a compiler-directed approach is used to design a loop accelerator from a C specification and a throughput requirement. The compiler analyzes the loop and generates a virtual architecture containing sufficient resources to sustain the required throughput. Next, a software pipelining scheduler maps the operations in the loop to the virtual architecture. Finally, the accelerator datapath is derived from the resulting schedule.

In this dissertation, synthesis of different types of loop accelerators is investigated. First, the system for synthesizing single loop accelerators is detailed. In particular, a scheduler is presented that is aware of the effects of its decisions on the resulting hardware, and attempts to minimize hardware cost. Second, synthesis of multifunction loop accelerators, or accelerators capable of executing multiple loops, is presented. Such accelerators exploit coarse-grained hardware sharing across loops in order to reduce overall cost. Finally, synthesis of post-programmable accelerators is presented, allowing changes to be made to the software after an accelerator has been created.

The tradeoffs between the flexibility, cost, and energy efficiency of these different types of accelerators are investigated. Automatically synthesized loop accelerators are capable of achieving order-of-magnitude gains in performance, area efficiency, and power efficiency over processors, and programmable accelerators allow software changes while maintaining highly efficient levels of computation.

# CHAPTER 1

# Introduction

Embedded devices such as cellular phones, personal digital assistants, digital cameras, gaming platforms, and music players continue to proliferate. The embedded computing systems that go into these devices must meet the demands of higher performance and greater energy efficiency to support new functionality and higher bandwidth communication. For example, the projected data rates for 4G wireless data communication are expected to increase 50 times over current 3G standards [71]. Conventional programmable processors are unable to meet the demands of these applications, so custom hardware is used to provide the desired levels of performance and energy efficiency. This custom hardware commonly takes the form of loop accelerators, which execute the compute-intensive portions of applications. Low cost, high performance, systematic verification, and short time-to-market are all critical objectives for designing these accelerators. Automatic synthesis technology to build loop accelerators from high-level specifications is critical to achieving these objectives.

A key challenge with automatic synthesis is creating quality designs. Quality

can be defined along many axes, including performance, cost, and energy. Tradeoffs among these axes can be made depending on the particular goals of the application. In this work, various loop accelerator designs are proposed, and the performance, cost, and energy tradeoffs are investigated. The objective is to automatically create customized hardware for a given application or set of applications, so that order-of-magnitude wins can be achieved in performance, cost, and energy consumption over general purpose processors.

Efficient accelerators are synthesized by optimizing the design in a number of ways. First, hardware structures are sized just large enough to meet the worst-case requirements of the application, both in terms of datapath width and number of entries in the storage structures. Second, connectivity between hardware structures is tailored to the application, decreasing interconnect costs, or conversely allowing the number of structures to be scaled up to exploit parallelism. Third, hardware can be shared by time multiplexing hardware components when either the hardware is required under disjoint conditions or the performance of dedicated hardware is not necessary.

This work examines the design of a loop accelerator synthesis system. The proposed system utilizes a compiler-directed approach for designing accelerators that was inspired by the PICO-NPA (Program-In Chip-Out Non-Programmable Accelerator) system [60]. The inputs to the system are a target loop expressed in C and the desired throughput. Synthesis is divided into three steps. First, a simple, single-cluster VLIW (Very Long Instruction Word) processor is designed to meet the throughput

requirements of the application. The simple processor consists of a set of function units, connected to a centralized register file with unlimited entries and an unbounded memory. It provides an abstract target to which the compiler can efficiently map algorithms. Next, modulo scheduling [58] is performed to map the application onto the simple processor. Finally, a stylized loop accelerator is synthesized from the resulting schedule. Since the cost of the resulting accelerator is highly dependent on the schedule, an intelligent *cost sensitive modulo scheduler* is proposed that minimizes accelerator cost during the scheduling phase.

In order to achieve greater hardware efficiency, multiple loops may be implemented on the same loop accelerator if the loops are to be executed disjointly. The accelerator synthesis system is augmented to create accelerators that can execute multiple target loops given their respective throughput requirements. By reusing a common datapath for multiple loops, coarse-grained hardware sharing is achieved, reducing the overall hardware cost from the baseline case of creating individual accelerators for each loop.

Finally, programmability is added to the accelerator hardware in order to combat a downside of ASIC design, namely the inability to change the software. Applications in the embedded domain evolve rapidly, and it is important to be able to reuse accelerator hardware when the loop is changed. By introducing a degree of programmability to the accelerator, its flexibility and usable lifetime is increased, while maintaining the efficiency advantages of customization.

## 1.1 Contributions

This dissertation makes the following contributions:

- An automated synthesis system, taking as input a loop expressed in C code and a throughput requirement, and generating Verilog code representing a custom loop accelerator.
- A cost sensitive modulo scheduler that schedules operations to minimize the hardware cost of the resulting accelerator.
- A system to synthesize a multifunction loop accelerator given a set of input loops and throughput requirements.
- Extensions to the loop accelerator architecture and toolchain to enable post-programmability, and evaluation of the tradeoffs between accelerator flexibility and efficiency.

## 1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of the motivations and tradeoffs of designing customized hardware, and the benefits of automated design and hardware reuse. Chapter 3 describes the hardware organization of a loop accelerator and how it may be integrated into a system of accelerators. It then describes the synthesis flow for a single-function accelerator. Chapter 4 presents the cost sensitive modulo scheduler used in the synthesis.

It also presents experimental data about the effectiveness of the scheduler and the methods of decomposing the scheduling problem to make it tractable for larger applications. Chapter 5 discusses multifunction loop accelerators and evaluates different methods of synthesizing them. Chapter 6 discusses the synthesis of programmable loop accelerators and the mapping of loops to such accelerators. The programmable accelerators are evaluated in terms of their area and power efficiency as well as their programmability. Finally, Chapter 7 concludes the dissertation and provides future research directions.

# CHAPTER 2

# Background and Motivation

## 2.1 The Costs of Computation

An algorithm or task has an inherent energy cost of computation, independent of what type of computing system is used to implement it. That is, given a task consisting of a set of operations to execute, there is a minimum amount of energy that must be expended in order to complete the task. If each operation corresponds to a computation by a function unit (FU), this overall minimum cost can be calculated by multiplying the energy cost of the FU executing each operation by the number of times that the operation needs to be executed, and summing up across all operations.

In reality, when the task is implemented on a computing system, there are additional costs that vary depending on the implementation, whether it is a traditional programmable processor, a coarse-grained reconfigurable architecture, an ASIC, or some other implementation. For example, intermediate values need to be stored in registers or memory, which have associated costs. Values need to be routed from pro-

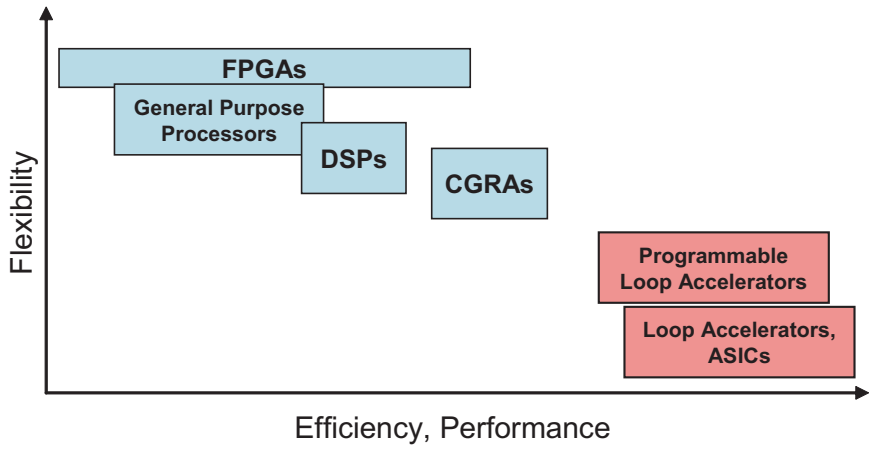


**Figure 2.1:** Architecture design space.

ducing operations to consuming operations, and the associated wires and multiplexers have costs associated with them. Also, if the task is implemented in a programmable machine, there are costs associated with fetching and decoding instructions and controlling the datapath.

For example, an ASIC has relatively low cost overhead over the inherent computation costs, because the storage and routing elements are tailored for the needs of the specific application, and there are no costs associated with control. Conversely, a general purpose processor has significant costs because it consists of a pipeline that fetches and decodes instructions, reads values from a central, wide register file, bypasses values between FUs, etc. The tradeoff is that the processor is usable for a wide variety of applications, while the ASIC, though much more efficient, is only usable for one. Figure 2.1 shows several implementation types in this space. The tradeoffs between flexibility and efficiency of different implementations will be further discussed in Section 6.2.1 in the context of designing programmable accelerators.

Thus, a typical embedded computing system is a heterogeneous mix of different types of computing implementations, each tailored towards different parts of the target applications. A critical loop nest, for instance, could be implemented on an ASIC, while more control-intensive out-of-loop code could be implemented on a general purpose core. In this way, different parts of the applications are matched with the most appropriate hardware implementations to achieve a design that is efficient overall.

In this dissertation, the focus is on applications in the embedded domain, including image, video, and audio processing, wireless communications, and encryption. Such applications typically consist of tight loops processing streaming data, and as such, the loops are critical to the overall performance. By creating efficient hardware accelerators targeted towards loops, significant wins are possible in performance and overall efficiency. These accelerators could be integrated into a heterogeneous system consisting of, for example, a general purpose core and a pipeline of several loop accelerators.

## 2.2 Automated Synthesis

When creating ASICs, much of the cost comes from the design and verification process, which must be repeated for each new ASIC; these costs are not as easily amortized as when designing general purpose processors. In addition, the design and verification process takes a significant amount of time, which is challenging in the rapidly evolving embedded domain where market requirements and standards evolve quickly. Thus, automated synthesis from high-level specifications, such as C, becomes

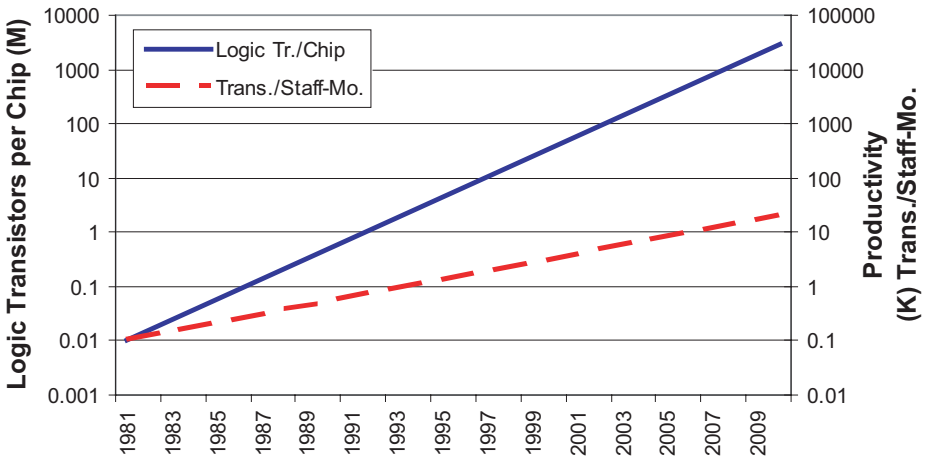


**Figure 2.2:** Design productivity gap [61].

key.

Automated synthesis provides a design system that is correct by construction, thereby reducing verification times significantly. In addition, design times are reduced from months or years to weeks, greatly speeding time-to-market of new devices. This translates into increased performance relative to other devices on the market. For example, if performance improves by 50% every year, and it takes a year longer to do a hand design than an automatic design, then the hand design must be 50% faster just to break even.

Furthermore, as hardware continues to grow more complex each year, non-automated design methods fail to keep up. Figure 2.2 shows that the number of transistors per chip is growing at an exponentially faster rate than the growth in designer productivity. Thus, automated synthesis systems are needed to create new devices within a competitive time frame and keep up with the embedded market.

Traditionally, hardware synthesis from high-level specifications consists of directly

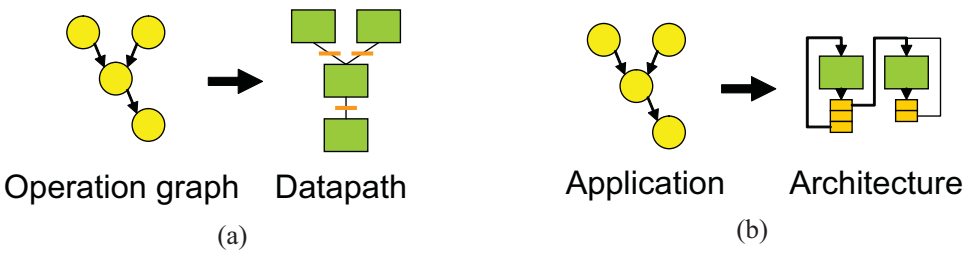


**Figure 2.3:** (a) Traditional behavioral synthesis. (b) Application-centric architecture synthesis.

translating C operators into hardware gates; see Figure 2.3. The synthesis system presented in this dissertation, however, takes a different approach, designing *application-centric architectures* for given applications. A performance requirement is specified in terms of the required loop throughput, and the system designs an application-specific loop accelerator to achieve the required throughput while maximizing hardware sharing, reducing overall cost and improving efficiency.

The synthesis system takes a compiler-directed approach, using compiler analyses to discover relationships between operations in a loop and expose instruction-level and loop-level parallelism. Compiler techniques are used to schedule the operations to maximize hardware sharing while meeting specified throughput requirements.

## 2.3 Hardware Reusability

As shown in Figure 2.1, the downside to creating an efficient application-specific accelerator is its lack of flexibility: the hardware cannot be reused to perform other tasks. Two methods are investigated to increase the reusability of the hardware.

The first method is to create an accelerator that can execute more than one loop.

In many cases, an embedded system will use hardware to accelerate multiple loops, and not all loops are executing simultaneously. When two or more loops are disjoint, one loop accelerator may be designed that is capable of executing some or all of the disjoint loops. This is referred to as a multifunction accelerator [20]. Coarse-grained hardware sharing is achieved, because a single multifunction accelerator is created rather than multiple single-loop accelerators.

Though multifunction accelerators increase hardware reuse and thus reduce cost, they still have the downside that all loops that will execute on the hardware must be known a priori. Therefore, the second method of increasing accelerator flexibility is to introduce post-programmability [21]. This allows a loop to be changed after the hardware has been created, and still be runnable on the hardware. Tradeoffs must be considered between the degree of customization of hardware to a given loop versus allowing more programmability, because as the datapath is generalized, additional cost overheads are introduced that reduce efficiency. An additional challenge is the question of how to quantify programmability of a hardware datapath.

## 2.4 Related Work

Datapath synthesis from behavioral specifications is a field that has been studied for many years. The basic techniques, including resource allocation and scheduling, have been well established [22]. Cathedral III represents a complete synthesis system developed at IMEC and illustrates one comprehensive approach to high-level synthesis [49]. It uses an applicative language for program specification and designs

customized datapaths for signal processing applications from this specification. Force-directed scheduling is used to synthesize datapaths for ASIC design in [57]. The Sehwa system automatically designs processing pipelines from behavioral specifications [56]. The PICO system synthesizes C loop nests into a synchronous array of customized processor datapaths [60].

The above systems produce standard cell based designs. Automatic mapping of applications to FPGA-based and other reconfigurable systems has also been investigated. One of the first efforts to automatically map applications onto an FPGA was Splash [23], subsequently productized as the NAPA system [24]. Other automatic compiler systems for FPGA-based platforms include GARP [6], PRISM [70], and DEFACTO [5]. Modulo scheduling has been used to map critical loops onto reconfigurable coprocessors [27, 62]. Compilation for architectures consisting of predefined FUs and storage with reconfigurable interconnect have been investigated, including RaPiD [13] and PipeRench [25]. The MOVE processor [10] is an application specific instruction-set processor (ASIP) based upon transport triggered architectures, where instructions direct the flow of operands rather than the computation.

In addition to the works discussed here, there are related works relevant to specific chapters of this dissertation. Those chapters will contain their own discussion of related work.

# CHAPTER 3

# Hardware Organization and Design Flow

## 3.1 System View

Embedded devices commonly execute streaming applications, in which multiple compute-intensive loops (such as filters) operate in turn on large amounts of streaming data. The natural realization of these tasks is a hardware pipeline of accelerators, each implementing one or more of the tasks that process the data. Figure 3.1(a) shows an example of a streaming application that consists of multiple loops; in some cases, different loops are executed depending on the type of input data. Figure 3.1(b) shows a pipeline of loop accelerators designed to execute the loops in the application. Each accelerator executes one or more of the loops, and SRAM buffers in between the accelerators allow the output of one accelerator to feed into the input of the next. The accelerator labeled LA2 is a multifunction accelerator, and can execute either Loop 2 or Loop 3.

The hardware pipeline is designed in an intelligent way, matching the throughput

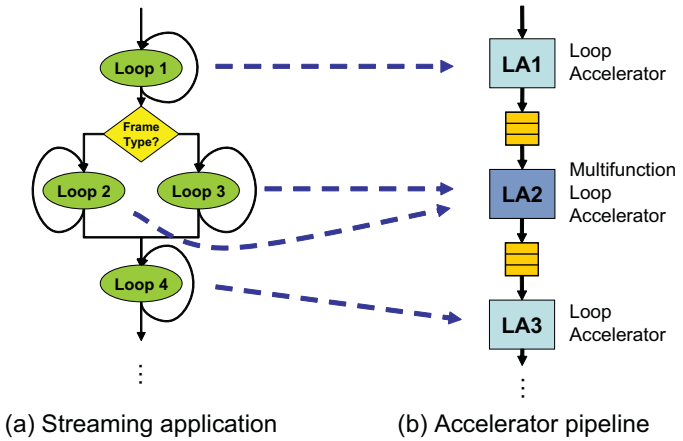


**Figure 3.1:** Streaming application mapped to pipeline of loop accelerators.

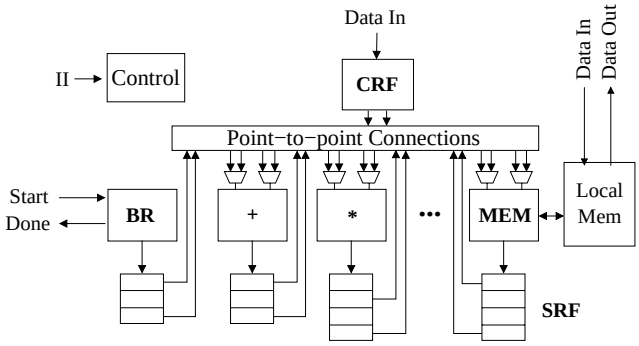


**Figure 3.2:** Loop accelerator template.

of each accelerator and sizing the buffers so that an overall performance goal is met while the overall system cost is minimized [34]. The specifics of the system-level design of the accelerator pipeline is outside the scope of this dissertation; this work focuses on synthesizing efficient individual accelerators.

## 3.2 Accelerator Template

Figure 3.2 shows the hardware schema used in this system. It is designed to exploit the high degree of parallelism available in modulo scheduled loops with a

large number of function units (FUs). Each FU writes to the top entry of a dedicated shift register file (SRF); entries move down at every cycle. Wires from the registers back to the FU inputs allow data transfer from producer to consumer. Multiple registers may be connected to each FU input; a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is simply a modulo counter. Other than this counter, no control signals are needed to address the registers.

Literals and static live-in register values cannot be stored in the SRFs. Therefore, live-in values are supplied by a central register file (CRF), and literals are hardwired to the inputs of FUs that require them. FUs that access memory are connected to a local memory structure such as a scratchpad, cache, or stream buffer. The loop accelerator begins execution when a *start* signal is asserted by the host processor. When the loop execution is complete, the branch FU asserts a *done* signal to the host processor.

Support for predication in the loop accelerator hardware is useful, because it allows loops with internal control flow to be modulo scheduled and accelerated. In addition, modulo scheduling itself uses predication to implement the prolog, kernel, and epilog phases of the schedule. Predication is supported in the loop accelerator via a valid bit associated with each register. In each cycle, each FU produces a value that is written to the top entry of the corresponding SRF. If the guarding predicate of the FU is true during this cycle, the corresponding valid bit is set to true, otherwise it is set to false. Later, when the value is used by an FU, a data-merge MUX at the FU

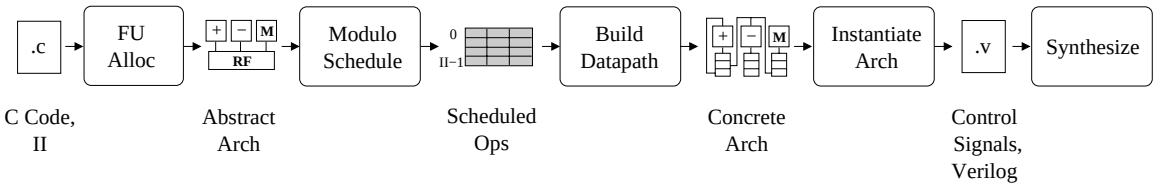


**Figure 3.3:** Loop accelerator design flow.

input selects the register whose corresponding valid bit is set. If multiple registers are valid, the MUX should select the most recently computed valid value. Recency is known statically, as the register closest to the top of its SRF will have been computed most recently.

## 3.3 Accelerator Design Flow

The overall flow of the synthesis system is presented in Figure 3.3. Each step of the flow is described in this section with an example from `sobel`, an edge detection algorithm.

### 3.3.1 FU Allocation

This step takes the inputs of the system and creates an abstract VLIW architecture that represents a high-level view of the accelerator's functionality. The abstract architecture is parameterized only by the number of FUs and their capabilities; a single unified register file with infinite ports/elements that is connected to all FUs is assumed. Given the operations in the loop, the desired throughput (expressed as the initiation interval of the loop or II [58]), and a library of hardware cell capabil-

ities and costs, the problem of FU allocation is to come up with a mix of FUs that minimizes cost while providing enough resources to meet the throughput constraint. In this phase, all FUs are assumed to be full width for cost purposes. (Bitwidth specialization is performed after the cost sensitive scheduling, when operations have been assigned to specific FUs.) In the simplest case where each operation can be executed by only one type of FU, $\lceil compatible\_ops/II \rceil$ instances of each FU type should be created. However, operations can generally be executed by multiple types of FUs, such as when both adder and adder/subtractor units are available. In this case, the FU allocation problem becomes more complex and can be formulated as an integer linear program, minimizing the sum of the FU costs while supporting all of the operations. Figure 3.4(b) shows the result of FU allocation for `sobel` with II=4. There are 22 ADD and 2 SUB operations in the loop, which are covered by the 5 ADD and 1 ADDSUB units.

### 3.3.2 Modulo Scheduling

The loop is modulo scheduled to the abstract architecture created in the previous step. A cost-sensitive modulo scheduler, to be described in Chapter 4, assigns operations to the resources and timeslots in the abstract architecture. At the completion of this phase, all of the loop operations are bound to resources and time, and the producer-consumer relationships between FUs have been determined. Figure 3.4(c) shows some operations from the modulo schedule for `sobel`, with edges indicating communication between operations. The number associated with each operation in-

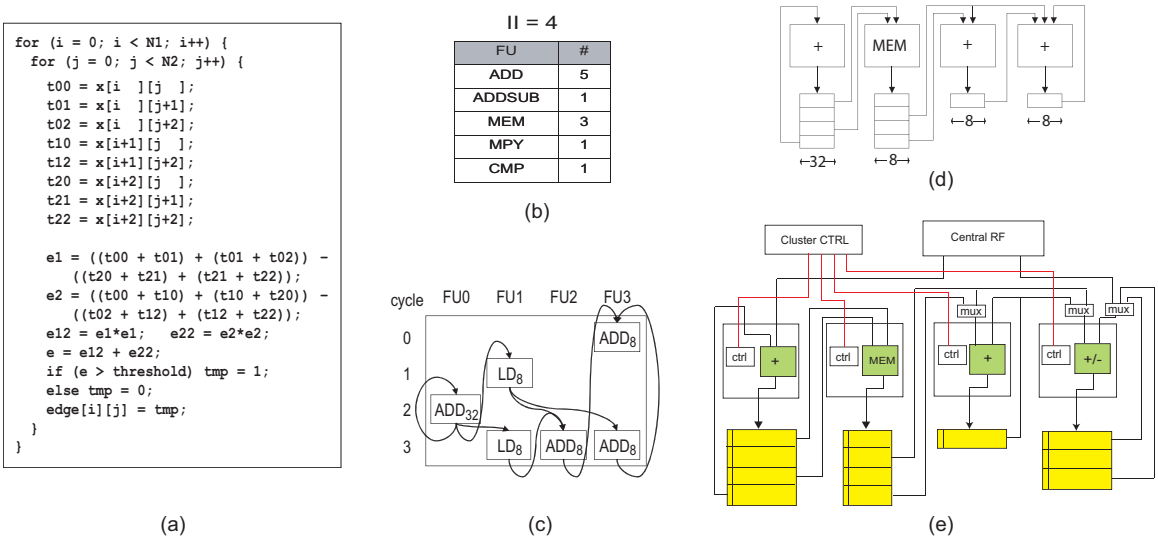


**Figure 3.4:** An example loop accelerator design. (a) `sobel` source code, (b) result of FU allocation with II = 4, (c) a portion of the `sobel` modulo scheduled loop, (d) datapath derived from the modulo schedule shown in (c), (e) lowered datapath.

dicates its width; the width of each FU is set to the width of the largest operation assigned to it.

### 3.3.3 Datapath Construction

The virtual FUs of the abstract architecture, concretized by operation assignments, directly become the FUs of the loop accelerator. The rest of the accelerator datapath is derived from the producer-consumer relationships in the modulo schedule. Wires connect a shift register entry at the output of a producing FU to the input of a consuming FU. The register entry that should be connected is determined from the difference in execution time between the producer and consumer, since register entries move down at every cycle. More specifically, the register number $r$ that should be connected to transfer a value from producing operation $p$ to consuming operation

$c$ is:

$$r = time(c) - time(p) + iteration\_distance(p, c) * II - latency(p)$$

assuming that the topmost register in each SRF is numbered 0.

The bitwidths of FUs and register files are determined by the maximum bitwidth of operations that are mapped to the FU or contained in the register. The depth of a register file is set to the longest lifetime of the values produced by the corresponding FU. Figure 3.4(d) shows the SRFs and connections resulting from the scheduled operations in Figure 3.4(c).

### 3.3.4 Architecture Instantiation

Lastly, the architecture created in the previous step is lowered into a Verilog realization of the accelerator. Each module in the datapath is translated into a set of primitive modules that have pre-defined behavioral Verilog descriptions. To reduce global wiring of control signals, we employ a distributed hierarchical control scheme that consists of three levels of control logic: FU control activates the appropriate primitive FU with the proper functionality and sets any internal MUX selects; cluster control converts the II value to generate high-level FU opcodes and sets the input MUXes select signals; and, processor control generates the II counter value. A subset of the final lowered datapath for `sobel` is presented in Figure 3.4(e). Input MUXes are added when multiple wires share the same FU input port, and the control path is generated to direct the MUXes and FUs.

# CHAPTER 4

# Cost Sensitive Modulo Scheduling

## 4.1 Introduction

This chapter focuses on the scheduling component of single-function accelerator synthesis. The scheduler is a key component of the synthesis system because the accelerator datapath is determined from the modulo schedule of operations, as described in Section 3.3.

Modulo scheduling is a method of overlapping iterations of a loop to achieve high throughput. The performance of the schedule is determined by the *initiation interval* (II), or the number of cycles between successive iterations of the loop. The modulo schedule contains a *kernel* which repeats every II cycles and may include operations from multiple loop iterations. As the modulo schedule implements a software pipeline, the execution of kernel operations must be orchestrated so that the pipeline fills and drains properly. The pipeline fill and drain phases are known as the *prolog* and *epilog*, respectively, and they are controlled during execution by a *staging predicate*.

Scheduling algorithms used in compilers traditionally focus on goals such as reducing schedule length and register pressure or producing compact code. In the context of a hardware synthesis system where the schedule is used to determine various components of the hardware, including datapath, storage, and interconnect, the goals of a scheduler change drastically. In addition to achieving the traditional goals, the scheduler must proactively make decisions to ensure efficient hardware is produced.

The objective of cost sensitive modulo scheduling is to create a schedule that not only achieves a specified throughput, but also yields the lowest cost accelerator design. To accomplish this objective, the accelerator design is modeled during scheduling, so the impact of binding decisions on cost can be assessed. Our first approach to this problem utilized a greedy strategy, wherein at each scheduling step, the alternative that produced the least cost increase to the current design was made. The greedy approach was generally better than the baseline cost insensitive scheduler, but not by a large amount. The scheduler got trapped in too many local minima and the overall quality did not improve much.

The central problem is that each portion of the accelerator architecture is not the result of an individual scheduling decision, but rather is determined by many interrelated scheduling decisions. Each decision for a single operation has cost implications on earlier and later decisions. Thus, a greedy approach inherently does not make sense as the cost saved by making one decision is often unrelated to the cost of the entire design. As a result, we decided to focus on two scheduling methods that provide exact solutions: branch-and-bound and integer linear programming. Our approach is

to develop cost sensitive formulations of both methods.

As with most exact formulations, these methods break down for moderate to large problem sizes as the run-time and memory usage of these methods explode. Thus, the scheduling problem is decomposed into a set of more manageable subproblems, where each subproblem is solved in a phase-ordered manner. We utilize three techniques to break down the problem: graph partitioning, space-time decomposition, and time-space decomposition. Graph partitioning divides loop bodies into smaller subgraphs, optimally scheduling the subgraphs, while space-time and time-space decomposition split the scheduling process into two separate phases, time slot and resource assignment. These methods sacrifice optimality of the schedule and thus of the cost of the accelerator, but enable realistic problems to be solved in a reasonable amount of time, while achieving substantial cost savings.

The contributions of this chapter are two-fold:

- Two exact methods for cost sensitive modulo scheduling are presented: branch-and-bound and integer linear programming. Each method can be applied to optimize for area, interconnect, or a simple combination of both. The effectiveness of these methods is compared to traditional cost insensitive and greedy cost sensitive modulo schedulers.
- To address the issue of problem size explosion common to exact scheduling methods, three methods for decomposing scheduling algorithms into phased solutions of simpler subproblems are utilized. They consist of graph partitioning, time-space decomposition, and space-time decomposition. The implementation

details of each are presented along with analysis of the performance tradeoffs.

## 4.2 Related Work

Cost sensitive scheduling in the context of data path synthesis has been studied for many years. Force-directed scheduling integrates resource allocation and scheduling into a common synthesis algorithm to minimize overall cost of synthesized datapaths [57]. Tradeoffs in allocating either low latency and expensive or high latency and inexpensive resources have been considered within an integrated scheduling and resource allocation algorithm [4]. [54] proposes a polynomial time scheduling algorithm based on heuristics that produces near optimal results. [36] presents an integer programming formulation for the scheduling problem in data path synthesis. Generation of more efficient designs by sharing hardware across basic blocks was recently proposed [46]. All of the above work handle only acyclic code regions. The optimization criteria usually is achieving shortest schedule length, or given a schedule length, achieving the least cost of data path. The focus of our work is cyclic scheduling. Though the components of the cost are the same, the optimization strategy is different because of the way in which function units are utilized in a cyclic schedule.

Heuristics that work as a preprocessing step to scheduling and try to minimize cost of the resulting hardware have also been studied. Clique-based partitioning algorithms were developed in the FACET project to jointly minimize function unit and inter-function unit communication costs [68]. Within the PICO system, width clustering is used to bind operations of narrow bitwidth to common resources to

reduce datapath cost [42]. Assignment of scheduled operations to resources with the goal of increasing interconnect sharing has been proposed [55]. The advantage of preprocessing heuristics is that they are fast and usually achieve good results when used in conjunction with a traditional scheduling algorithm. Our work intertwines the cost minimization into the scheduling algorithm to achieve greater cost savings.

In the compiler domain, software pipelining is a technique to exploit instruction-level parallelism by overlapping the execution of successive loop iterations. Modulo scheduling is a class of software pipelining algorithms that achieve high quality solutions and have been implemented in production compilers [58]. A number of extensions to modulo scheduling have been proposed to increase the quality of the solution, including reducing register requirements [14, 28, 40] and code size [41]. Reducing register requirements is most closely related to accelerator cost reduction. Swing modulo scheduling changes the core modulo scheduler to reduce register requirements by considering operations in different orders and changing how time slots are chosen [40]. Conversely, stage scheduling is a post-processing to shift the pipeline stage of instructions to reduce register requirements [14]. While the application of these techniques can reduce the cost of loop accelerators, the affect is limited as traditional compiler-based measures, such as register lifetimes, do not reflect the structure of a loop accelerator. For instance, a long lifetime may be free in an accelerator if it is scheduled to share a register with a similar lifetime. Hardware sharing and all aspects of cost must be considered to create efficient loop accelerators.

Many techniques for optimal modulo scheduling have been proposed in the lit-

erature. [15] proposes and efficient integer programming formulation for optimal modulo scheduling. [2] proposes an enumeration based approach for optimal modulo scheduling. Both of these techniques focus primarily on achieving a valid schedule. Minimizing register requirements has been the main optimization criteria for many of the works published on optimal modulo scheduling. [26], [16], and [17] formulate the modulo scheduling with minimum register requirements as an integer linear programming problem. Our work uses the basic ILP formulation from [15] and builds upon it significantly by adding variables and constraints to represent the cost of hardware and uses the hardware cost as the objective function.

## 4.3 Scheduling Techniques

Cost sensitive modulo scheduling focuses on reducing the cost of three components of the hardware: FUs, register storage, and interconnect wires. These components were found to dominate the hardware cost of loop accelerators; other components such as multiplexers and control signals are less significant and are not specifically targeted for cost reduction in this work. By reducing the sizes of FUs and shift registers required to support execution of a given loop, the resulting hardware implementation will achieve the same throughput with fewer logic gates and less power. In addition, with high numbers of FUs and registers to support loop level parallelism, the interconnection network feeding values from registers to FU inputs can grow very large. Decreasing the number of wires required to support these data transfers reduces chip area from the wires themselves as well as from simplifying the placement and routing

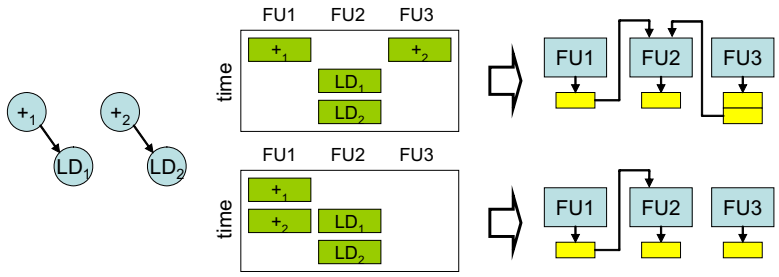


**Figure 4.1:** Effect of schedule on wire cost.

of other structures in the hardware layout.

FU and storage cost can be reduced by scheduling operations cognizant of their resource and communication requirements, such as bitwidth and register lifetimes; by maximizing hardware reuse, the total amount of hardware is reduced. Wire cost can be reduced by maximizing reuse of the same wires by different producer and consumer operations. Wires are reused if producers and consumers are scheduled on the same respective FUs, and the consumers read data from the same shift register entry (i.e., the time differences between producers and consumers are identical). In Figure 4.1, assume the two pairs of operations to be scheduled are 32 bits wide. An interconnect-unaware modulo scheduler might produce the upper schedule, which requires 64 wires, while the lower schedule would have required only 32.

The remainder of this section describes approaches for achieving these goals, assigning operations to FUs and time slots such that the cost of the hardware needed to support their execution is minimized.

### 4.3.1 Greedy Scheduling

The baseline (naïve) scheduler used in this work is the iterative modulo scheduler described in [58], with a stage scheduling postpass [14]. This scheduler arbitrarily selects an available scheduling alternative for each operation in order to meet a given II, and does not consider hardware cost. The stage scheduling postpass reduces register lifetimes, which may reduce hardware cost, but this is done without cognizance of the hardware.

A straightforward way to make the scheduler cost-aware is to augment the naïve modulo scheduler with a hardware cost model and a greedy heuristic to minimize cost. The cost aware scheduling framework is shown in Figure 4.2. The main component of this framework is the hardware cost modeler, explained in more detail in Section 4.3.2.1. The hardware cost model is able to represent the cost of a partial machine, that is, the cost of hardware resources required to support execution of just the scheduled operations. In addition, the cost modeler can estimate the cost of hardware that would be required to support the remaining, unscheduled operations. (This estimate is explained in more detail in Section 4.3.2.2.)

To choose the best local alternative, the greedy modulo scheduler makes queries about the machine cost to the hardware cost modeler. The cost modeler returns a cost estimate that includes both the partial machine cost as well as the estimated cost of unscheduled operations. Based on this cost, the scheduler chooses the best alternative and schedules the operation on that particular FU and time slot. This is done for all operations in priority order, backtracking as needed. After the completion

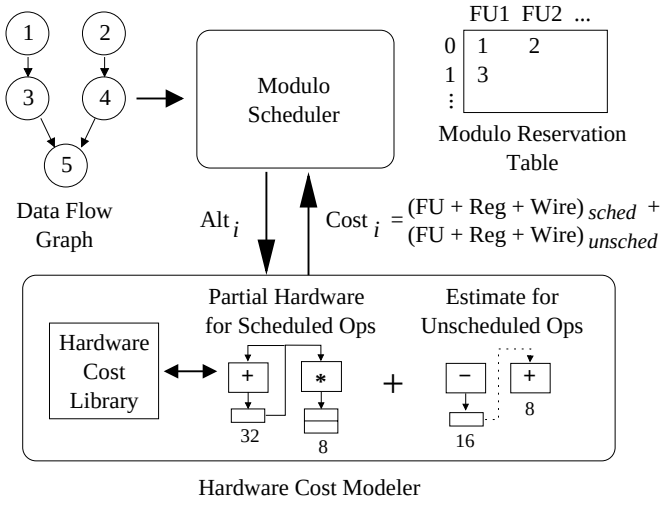


**Figure 4.2:** Cost sensitive scheduling framework used for greedy and branch-and-bound schedulers.

of greedy scheduling, the stage scheduling postpass is performed to decrease register lifetimes.

### 4.3.2 Branch-and-Bound Solution

A second method of obtaining a modulo schedule that minimizes hardware cost is to utilize an optimal branch-and-bound (BNB) solution. The goal is to search all possible schedules in order to find one that has the lowest hardware cost. The search is performed by scheduling each operation at all of its valid alternatives (FUs $\times$ time slots). In a modulo schedule, each operation can be scheduled in at most II different time slots. Operations are considered in order of least to most available alternatives; the order does not affect the algorithm's optimality, only its runtime. The search space can be represented by a tree as shown in Figure 4.3. Each node represents a partial schedule, or a state in which some operations have been assigned FUs and

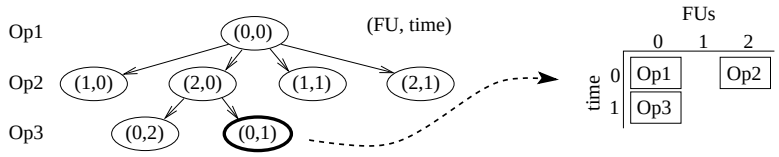


**Figure 4.3:** Branch-and-bound search. The highlighted state corresponds to the partial schedule shown.

time slots. The children of a node are formed by scheduling the next operation at all of its valid alternatives, subject to resource and dependence constraints. Leaf nodes in the tree therefore represent full schedules, and the goal is to locate a leaf node whose schedule requires the minimum amount of hardware.

#### 4.3.2.1 Hardware Modeling

The BNB scheduler uses a hardware model to estimate the cost of a machine supporting a given partial schedule. Three aspects of hardware cost are modeled: FUs, register storage, and interconnect wires. Function unit cost is determined by its capabilities and width. In the loop accelerator synthesis system, FU capabilities are determined during the FU allocation phase described in Section 3.3, prior to scheduling. Therefore, the scheduler has influence only on the width of the FU – if only narrow bitwidth operations are scheduled on an FU, then its cost can be reduced. The FU cost for a given partial schedule can therefore be determined as a function of the maximum bitwidth operation scheduled on each FU.

The register storage cost is determined similarly. Each shift register must be wide enough to accommodate the maximum bitwidth operation scheduled on the corresponding FU, and deep enough to hold the value with the longest lifetime. Also,

interconnect wires must connect specific registers with FU input ports. Given a partial schedule, the known producer-consumer relationships between operations is used to obtain the widths and depths of the shift registers, as well as the number of interconnect wires required.

The BNB algorithm requires a single metric to determine whether a given schedule is better or worse than previously explored schedules. Therefore, when the objective is to decrease overall hardware cost, the combined metric used is the sum of wires, storage bits, and FU cost. The units of FU cost are scaled such that they are equivalent to storage bits in terms of the number of logic gates required for implementation. The wire, storage, and FU metrics may also be used alone, for example, to obtain a schedule with the objective of minimizing only storage cost.

#### 4.3.2.2 Hardware Cost Estimation

An effective bound function is a crucial element in any BNB algorithm in order to prune, as early as possible, search paths that will not yield optimal results. The search is bounded using an estimate of the hardware needed to support operations that have not yet been scheduled. Thus, for any partial schedule, when the cost of hardware required by scheduled operations plus hardware estimated for unscheduled operations exceeds the best solution found so far, that search path is pruned. As long as the estimate is conservative (i.e., never overestimates the actual hardware cost), optimality is preserved as no search paths will be erroneously pruned. Additionally, the more accurate the estimate, the more likely a wrong path will be pruned earlier,

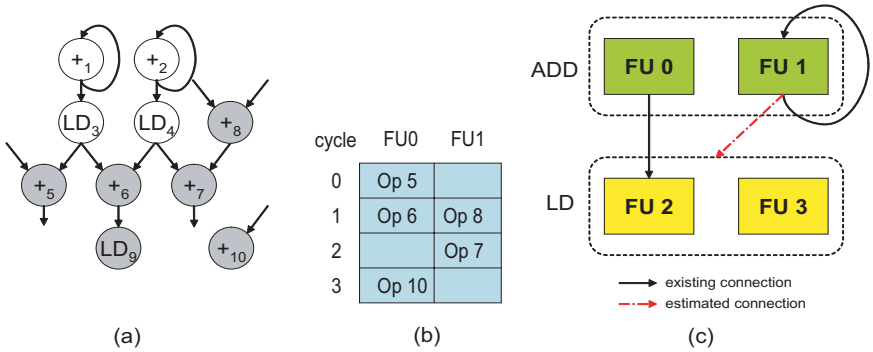


**Figure 4.4:** Wire estimation example: (a) DFG, (b) partial schedule, (c) connection diagram

thus decreasing the run time of the search.

**Wire estimation.** For the wire estimation, FUs are placed into groups based on their functionality. An FU group is the basic unit for estimation, and estimated connections are made between FU groups. For a given pair of FU groups, we collect all compatible edges[1] whose producer or consumer ops are unscheduled. Then, we determine the minimum number of additional connections required to support those unscheduled edges based on the number of available slots on the FUs (each FU has II slots). We optimistically assume that empty slots in the FUs can be occupied by any compatible unscheduled producer, ignoring scheduling constraints. This assumption guarantees that the estimation is a lower bound for the wire cost. It is assumed that existing wires in an FU group with $n$ free slots can be reused by $n$ unscheduled operations with compatible edges. When there are more than $n$ such operations, new estimated connections are made to support the remaining operations.

Figure 4.4 shows how the estimation is performed for the ADD FU group. The processor consists of four FUs, two in the ADD FU group (FU0, FU1) and two in

[1]Multiple dataflow edges whose producer and consumer operations can execute on corresponding FU groups.

the LD FU group (FU2, FU3). Assume the shaded operations are already scheduled and wires are being estimated for the unshaded operations. There are two types of edges originating from ADD operations: ADD→ADD and ADD→LD. As there is already a connection from FU 0 to FU 2 and FU 0 has one available slot, one of the two ADD→LD edges can potentially be scheduled without generating additional connections. This will make FU 0 fully occupied and the producer of the second edge must be placed on FU 1. Therefore, a new estimated connection between FU 1 and the LD FU group is created. Another estimation is performed independently for the ADD→ADD edges. Here, both can potentially be scheduled by placing the producers on FU 1, as it has two available slots. Thus, the ADD→ADD edges will not require any additional connections. As a result, the wire estimation for the ADD FU group is one. Wire estimation for the LD FU group proceeds in a similar manner.

**Storage estimation.** Estimating the incremental storage requirements for unscheduled operations is performed using an analogous method. First, the overall storage requirements for the unscheduled operations is determined; then, based on the number of available execution slots in the FUs and the existing register storage, the number of bits of new storage needed to support the unscheduled operations is estimated.

For each unscheduled operation, an estimate of the number of register bits needed to hold its result is obtained. This value depends on the width and depth of the output register; the width is simply the bitwidth of the operation, while the depth can be estimated from the estart/lstart[2] times of the operation and its consumers.

[2] $estart_{op}$: earliest start time of *op* ignoring resource constraints. $lstart_{op}$: latest start time of *op*

More specifically, for operation *op* with consumers *cons*:

$$depth = \left(\max_{c \in cons} estart_c\right) - lstart_{op} - latency_{op} \tag{4.1}$$

Once register requirements are approximated for the unscheduled operations, it is optimistically assumed that existing shift registers at the outputs of compatible FUs with available execution slots can be reused to satisfy these requirements. Any required register bits that cannot be satisfied by existing registers become part of the incremental storage estimation. Similarly to the wire estimation, this storage estimation does not take dependence constraints into consideration and is therefore conservative.

**Function unit estimation.** FU cost estimation is somewhat simpler than wire or storage estimation, since FU capabilities are fixed prior to scheduling and only the FU width varies depending on the schedule. First, unscheduled operations are grouped by type and their maximum bitwidth is determined. Next, existing FUs with free slots are used to satisfy these FU requirements. Finally, the additional cost of FUs needed to support the remaining operations (either by widening existing FUs or creating new FUs) is calculated.

For a given partial schedule, once the wire, storage, and FU costs have been estimated for the unscheduled operations, the search may be pruned. Once again, a single metric is needed for the hardware cost estimate, and this is obtained by the weighted sum of the wire, storage, and FU cost metrics. Note that these hardware

without delaying exit operations.

estimations are performed at every step of the BNB search. Therefore, they are implemented in a computationally efficient way, using incremental updates to internal data structures in order to minimize their impact on the execution time of the search. Note also that it is worthwhile to spend some computation time obtaining an accurate estimate if it allows the search paths to be pruned earlier, since the number of states eliminated by pruning a node is exponential in the height of the node.

### 4.3.3 Integer Linear Programming Formulation

The third approach to the problem is an integer linear programming (ILP) formulation for achieving modulo schedules optimal with respect to the cost of hardware generated from the schedule. The basic structure of the formulation is identical to the one proposed in [15, 26]. The basic formulation described in these works do not perform FU assignment, but only ensure that a valid assignment is possible. FU assignment is crucial in determining cost of hardware derived from the schedule. In the formulation described in this section, additional variables and constraints to represent FU assignment for operations is added to the basic formulation. An objective function to represent hardware cost is derived from these variables and constraints.

#### 4.3.3.1 Basic Formulation

The body of the loop under consideration is represented by a graph $G = \{V, E\}$, where $V$ represents the set of operations in the loop body and $E$ represents data dependence edges between operations. Each dependence edge has an associated latency

$l_{i,j}$ which specifies the latency of the producer $i$, and a distance $d_{i,j}$, which specifies the difference in iterations between when the value is produced by $i$ and when the value is consumed by $j$.

Consider a loop with $|V| = N$ operations. Let $II$ be the initiation interval. The schedule for this loop is represented by $II \times N$ binary variables $X_{i,s}$. Operation $i \in \{0, N-1\}$ is scheduled in slot $s$, $0 \leq s \leq II - 1$, if $X_{i,s} = 1$. The following constraint enforces a unique slot for every operation $i$.

$$\sum_{s=0}^{II-1} X_{i,s} = 1 \qquad \forall i \in \{0, N-1\} \tag{4.2}$$

$N$ integer variables $k_i, i \in \{0, N-1\}$ are introduced to represent the stage in which each operation is placed. $X_{i,s}$ and $k_i$ uniquely identify the cycle in which an operation $i$ is scheduled. In fact, the schedule time of an operation $i$ is given by

$$t_i = \sum_{s=1}^{II-1} s \times X_{i,s} + II \times k_i \tag{4.3}$$

Note that $t_i$ is used as a shorthand to represent the schedule time of an operation $i$. In a real implementation, there is no need to introduce a new variable to represent the schedule time. Given the $t_i$'s for all operations, the data dependences between operations can be enforced with the following set of constraints.

$$t_j + d_{i,j} \times II - t_i \geq l_{i,j} \qquad \forall (i,j) \in E \tag{4.4}$$

The schedule times should satisfy the resource constraints, i.e., the number of operations scheduled in each slot should not exceed the available number of FUs for each FU type. Suppose $I_f$ are the set of operations that require a FU of type $f$ and $M_f$ are the total number of FUs of type $f$ available. Then, the following constraint enforces the resource constraints.

$$\sum_{i \in I_f} X_{i,s} \leq M_f \qquad s \in \{0, II-1\} \tag{4.5}$$

Note that the above constraint only ensures a valid FU assignment and does not actually perform the assignment.

#### 4.3.3.2 Function Unit Assignment

The FU assignment for operations is represented by a set of binary variables $R_{i,j}$, $i \in \{0, N-1\}$, $j \in \{0, M_f - 1\}$, i.e., there are $M_f$ binary variables for every op $i$, where $M_f$ is the number of compatible FUs to which $i$ can be assigned. The following constraint enforces a unique assignment.

$$\sum_{j=0}^{M_f-1} R_{i,j} = 1 \qquad \forall i \in \{0, N-1\} \tag{4.6}$$

The number of operations assigned to a particular FU cannot exceed $II$. The following constraint enforces this.

$$\sum_{i \in I_j} R_{i,j} \leq II \qquad i \in I_j \text{ can execute on } j \tag{4.7}$$

Even with the above constraint, an FU can be assigned to two operations in the same cycle. To prevent this from happening, the following constraint has to be enforced for every FU.

$$\sum_{i \in I_j} R_{i,j} \times X_{i,s} \leq 1 \qquad \forall s \in \{0, II-1\} \; and \; i \in I_j \text{ can execute on FU } j \tag{4.8}$$

The above equation is a sum of products of two binary variables, and is non-linear; it can be linearized as follows. For every $R_{i,j}$ and $X_{i,s}$ appearing in the above set of equations, an auxiliary binary variable $Z_{i,j,s}$ is introduced and following set constraints are enforced on $Z_{i,j,s}$.

$$\begin{aligned} -R_{i,j} + Z_{i,j,s} &\leq 0 \\ -X_{i,s} + Z_{i,j,s} &\leq 0 \\ R_{i,j} + X_{i,s} - Z_{i,j,s} &\leq 1 \end{aligned} \tag{4.9}$$

Now the product terms in Equation 4.8 can be replaced with the corresponding $Z_{i,j,s}$'s. Solving equations 4.2 through 4.9 would yield a valid schedule and FU assignment for operations in a loop.

#### 4.3.3.3 Cost Minimization

As described in Section 3.2, the hardware schema is a set of FUs writing to independent shift registers. The cost of the hardware includes cost of the FUs and cost of the shift registers and cost of wires used to connect shift registers to the input of FUs. In this section, we describe modeling of costs of FU and shift registers only.

Modeling wire cost is left out due to space considerations.

**Function unit cost.** The cost of the FU depends on the set of operations assigned to it. For example, if 8-bit and 16-bit add operations are assigned to an add FU, then the cost of the add FU is the cost of a 16-bit adder. Suppose $H_i$ is the cost of a FU required to execute operation $i$ only. $H_i$ is a constant and is a (possibly non-linear) function of the bitwidth of operation $i$. Now, the cost of a FU $j$ will be at least $H_i$, if $i$ is assigned to $j$. Since we have binary variables to represent the fact that operation $i$ is assigned to FU $j$, the above fact be represented as follows.

$$C_j \geq R_{i,j} \times H_i \qquad i \text{ can execute on FU } j \textit{ and } C_j \text{ is the cost of FU } j \tag{4.10}$$

The above constraint is introduced into the integer program for every operation $i$ that can be assigned to FU $j$. Thus, $C_j$ automatically gets set to the maximum cost of an FU that can execute any set of operations assigned to it. The total cost of FUs in the hardware can be calculated as follows.

$$\sum_{j \in FUs} C_j \tag{4.11}$$

**Storage cost.** As described in Section 3.2, the FUs write their output to the head of a shift register which shifts the values down every cycle. The shift register should have enough entries to hold the values until the consumer FU reads it in a later cycle. Consider an operation $i_1$ feeding another operation $i_2$. From Equation 4.3 we know that $t_{i_1}$ and $t_{i_2}$ are the schedule times of $i_1$ and $i_2$ respectively. The value

produced by $i_1$ is read by $i_2$ after $t_{i_2} - t_{i_1} + II \times d_{i_1,i_2} - l_{i_1,i_2} + 1$ cycles. $i_1$ could have many consumers and the latest time a value produced by $i_1$ is live is the maximum of $t_{i_2} - t_{i_1} + II \times d_{i_1,i_2} - l_{i_1,i_2} + 1$ with respect to some consumer. A integer variable $LT_i$ is introduced for every producer $i$ operation in the loop body to indicate the maximum lifetime (measured in number of cycles) of the value produced by that operation.

$$LT_i \geq t_{i'} - t_i + II \times d_{i,i'} - l_{i,i'} + 1 \qquad (i, i') \in E \tag{4.12}$$

Note that the lifetime indicates the lifetime in actual number of cycles. This is significantly different from the lifetime measure used in [16, 26] which is just the maximum number of values produced by an operation live at any instant. The maximum lifetimes of values produced by operations is used to calculate the depth $D_j$ of the shift register associated with an FU $j$. A shift register should hold live values from all operations assigned to it. Therefore, $D_j$ is the maximum of lifetimes of any operation assigned to it. This can be represented as follows.

$$D_j \geq R_{i,j} \times LT_i \qquad \forall i \text{ assigned to } j \tag{4.13}$$

The above equation is a product of a binary variable and an integer variable, and is non-linear. However, it can be linearized using an auxiliary variable $TD_j$ as shown below.

$$
\begin{aligned}
TD_j &\geq 0 \\
TD_j &\leq P \times R_{i,j} \\
TD_j &\leq LT_i \\
TD_j &\geq LT_i - (1 - R_{i,j}) \times P \\
D_j &\geq TD_j
\end{aligned}
\tag{4.14}
$$

where $P$ is a suitably large constant. Note that $TD_j$ is 0 when $R_{i,j}$ is 0 and is equal to $LT_i$ when $R_{i,j}$ is 1. $D_j$ thus gets the maximum of $LT_i$ among all operations $i$ assigned to FU $j$.

The cost of the shift register of an FU also depends on the bitwidth of the operations assigned to the FU. In fact, the width of the shift register has to be the maximum of the bitwidths of operations assigned to the FU. The width $W_j$ of the shift register associated with FU $j$ is calculated as follows.

$$
W_j \geq R_{i,j} \times BW_i \qquad \forall\ i \text{ assigned to } j \tag{4.15}
$$

where $BW_i$ is a constant, indicating the bitwidth of the values produced by operation $i$. From $D_j$ and $W_j$, the cost $S_j$ of the shift register associated with FU $j$ can be calculated as follows.

$$
S_j = W_j \times D_j \tag{4.16}
$$

The above equation is non-linear. However, it can be linearized using the observation that $W_j$ can take only a small set of discrete values. Suppose $W_j$ can take values $w_1$,

$w_2$, ... , $w_k$. Then, $W_j$ can be represented as shown below.

$$W_j = \sum_{n=1}^{k} w_n \times b_{j,w_n}, \qquad \sum_{n=1}^{k} b_{j,w_n} = 1 \tag{4.17}$$

where $b_{j,w_1}$, $b_{j,w_2}$, ... , $b_{j,w_k}$ are binary variables. Now $S_j$ can be expressed in linear form as follows.

$$\begin{aligned} S_j &\leq w_{max} \times D_j \\ S_j &\geq w_n \times D_j - (1 - b_{j,w_n}) \times Q \qquad \forall n \in \{1, n\} \end{aligned} \tag{4.18}$$

where $w_{max}$ is the maximum among $w_1$, $w_2$, ... $w_k$ and Q is a suitable large constant.

The objective function for minimizing the cost of data-path of the hardware can now be calculated from equations 4.11 and 4.18.

$$\sum_{j \in FUs} C_j + S_j \tag{4.19}$$

The overall ILP formulation for cost sensitive modulo scheduling can be stated as "minimize Equation 4.19, subject to the constraints expressed in Equations 4.2 through 4.18."

## 4.4 Decomposition Methods

It is necessary to decompose the modulo scheduling problem described in the previous section because the number of possible schedules is too large for realistic loops. There are multiple ways in which the problem can be decomposed. One

approach is to partition the dataflow graph into sets of operations and then schedule the sets one by one. Another approach is to perform scheduling in phases. In this case, all operations are considered at once, but only resource assignment is performed in the first phase, and time assignment is performed in the second phase. Alternatively, the two phases can be performed in reverse order.

### 4.4.1 Operation Partitioning

One natural way of simplifying the scheduling problem is to partition the operations into multiple disjoint sets. The size of each set is bounded (generally to 10-15 operations), and thus the space of possible schedules for the operations in a set can be reasonably explored using the branch-and-bound or ILP techniques described in Section 4.3.

The scheduler considers sets of operations in sequence. Within each set, an optimal assignment of operations to resources and time is obtained which minimizes the cost of the additional hardware required by this set. Once operations from a set are scheduled, their resource and time slot assignments are fixed, and subsequent sets will take these assignments into account when they are scheduled. Thus, for each set of operations, the scheduler attempts to utilize two forms of hardware sharing to minimize cost: intra-set sharing, where operations within a set reuse new hardware, and inter-set sharing, where operations reuse existing hardware from previously scheduled sets.

The partitioning scheduler therefore obtains an optimal solution for each set, and

the combination of these solutions forms the final global schedule. This decomposition method loses some global optimality because only operations within the same set are considered together, and scheduling decisions made in earlier sets cannot be changed when scheduling later sets. However, in general this method is effective in producing low-cost schedules as both resource and time assignments are made jointly, and the decisions account for previously scheduled sets. An elegant tradeoff can be achieved between global optimality and running time of the scheduler. Larger sets are likely to give solutions closer to the globally optimal solution at the cost of increased search time. Smaller sets can be quickly searched to find locally optimal solutions.

Two issues have to be addressed in this scheduling scheme. First, a suitable partitioning method must be devised. Second, a backtracking strategy has to be designed to ensure successful completion of the scheduler.

#### 4.4.1.1 Partitioning Method

A simple way to partition the data flow graph is to consider the height based priority order of operations used in a typical scheduler, and place every $n$ operations into a set (where $n$ is the desired set size). Since the height based priority minimizes the instances where a consumer is scheduled before its producer operation, this partitioning method minimizes backtracking and ensures quick convergence to a schedule. A more sophisticated graph partitioning method could also be employed to form partitions. However, unlike traditional graph partitioning, the goal of partitioning the dataflow graph of the loop body is not to achieve min-cut of the edges. This is because we are

not considering a traditional performance metric like schedule length. Instead, a good partition is one which exposes as many hardware sharing opportunities as possible within a set. Since exhaustive search is performed on each partition, all the sharing opportunities will be exploited and the combined global solution is improved.

A simple heuristic is used to form partitions with high hardware sharing opportunities. First, a similarity metric is calculated between every pair of operations in the dataflow graph. Then, the operations are partitioned into sets by taking operation pairs in order of descending similarity and placing every $n$ operations into a set.

The similarity metric has two components, one based on potential for sharing interconnect wires and one based on potential for sharing register storage. The wire similarity metric is a count of the number of wires (in bits) that can potentially be shared between two operations and their producers/consumers, determined by counting compatible edges. To estimate the storage similarity metric, register requirements are first estimated for each operation using the method from Section 4.3.2.2. Then, the metric is calculated as the number of bits of register storage the two operations have in common. This figure accounts for the dimensions of the register files, so that a wide, shallow register file has little similarity with a narrow, deep file even if the total number of storage bits is similar.

This storage similarity metric can be augmented to account for "register waste," that is, unused bits of storage that would result if the two operations shared storage. This gives preference to combining an operation with small register requirements with another similar operation, rather than one with large register requirements, even if

the bits of common storage would be the same.

Figure 4.4(a) shows an example DFG. Consider the two operations $+_1$ and $+_2$. Both of them have an incoming edge from an add operation and an outgoing edge to a load operation; thus, the wire similarity metric is 64 (assuming 32-bit operations). Similarly, both operations will require the shift registers to hold their results for $II$ cycles as there is an inter-iteration dependence from each add to itself; assuming $II = 4$, this translates to a storage similarity metric of 128. Thus, the overall similarity between the two operations is 192. Assuming these are the most similar operations in the DFG, they will be added to the same operation set and scheduled together.

#### 4.4.1.2 Backtracking

During modulo scheduling, it is possible that a set of operations cannot be scheduled due to conflicts with previously scheduled operations. In such a situation, it is necessary to use backtracking in order to maintain forward progress. When a conflict arises during traditional modulo scheduling, the operation is forcibly scheduled and conflicting operations are unscheduled and placed in the queue to be rescheduled later. The method of backtracking used in this scheduler is similar, but at the granularity of operation sets rather than individual operations. When a set cannot be scheduled, first it is determined which scheduled operation(s) is causing the conflict. Then, all operations in the same set as the conflicting operation are unscheduled. Finally the current set of operations is scheduled, and the unscheduled set is later rescheduled.

In general, backtracking has an adverse effect on the solution quality. This is

because each set is optimally scheduled given the previously scheduled sets. If some of these previous sets are later unscheduled, the current set is no longer optimal. In addition, the sets are effectively scheduled out of priority order, which can potentially decrease the amount of hardware sharing that is achieved.

### 4.4.2 Time and Space Decomposition

The job of a scheduler is to assign both a schedule time and an FU (e.g., "space") to every operation in the loop body. In the context of the schedulers described in Section 4.3, assigning both time and space for an operation in a single pass has a multiplicative effect on the search combinatorics. For example, in the branch-and-bound scheduler, the number of possibilities for an operation to be explored by the scheduler is the product of number of time slots possible for the operation and the number of FUs to which the operation can be assigned. Similarly, in the ILP scheduler, the number of variables introduced by Equation 4.9 is equal to $II$ times number of FUs for every operation in the loop body.

The problem of scheduling can be decomposed into its two constituent phases: (1) assigning a time slot to every operation, (2) fixing the operations in space. Note that the first phase still has to honor resource restrictions, i.e., it cannot assign more operations to a time slot than there are FUs available to execute those operations. The second phase of assigning operations to FUs should ensure that it does not assign two operations scheduled in the same time slot to the same FU. Such an

assignment is always made possible by enforcing the resource restrictions in the first phase. The number of possibilities for every operation is reduced from $O(II \times \#FUs)$ in the combined solution to $O(II) + O(\#FUs)$ in the decomposed solution. The decomposed scheduler phases still have to be cost sensitive. Due to the nature of decomposition, some optimizations may not be possible in a particular phase. In time-space decomposition, optimizing for FU cost and width of the shift register file is not possible in the first phase. This is because the cost of FUs and width of registers depend on the assignment of operations to FUs. However the time assignment phase can optimize the depth of the shift register files.

In the ILP scheduler, assigning valid time slots to operations can be enforced using the constraints given by Equations 4.2 through 4.5. Note that time slot assignment is sufficient to calculate the lifetime of the value produced by an operation $i$, given by Equation 4.12. Since the lifetimes $LT_i$ directly affect the register depth, minimizing lifetimes is important. Therefore, $\sum_{i=0}^{N-1} BW_i \times LT_i$ is used as the objective function in the formulation. Note that the lifetimes of operations are weighted by their bitwidths $BW_i$. This is to ensure that lifetimes of narrow operations are not minimized at the cost of wide operations. Solving the set of constraints described above gives a time slot assignment for all operations in the loop. Now the space (resource) assignment can be performed by forming a new ILP problem which includes all equations described in Section 4.3.3. The objective function remains the same, given by Equation 4.19. However, the values of time slots $X_{i,s}$ and stages $k_i$ obtained from time assignment phase are explicitly specified to the ILP problem formed in the space assignment

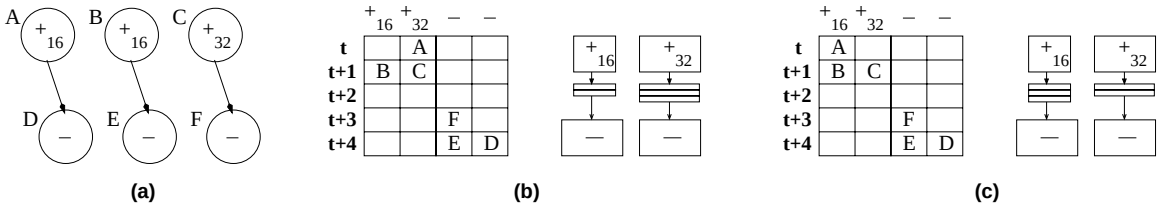


**Figure 4.5:** Effect of space-time decomposition. (a) Dataflow graph, (b) schedule resulting in optimal FU cost but suboptimal overall cost, (c) schedule with same FU cost and improved overall cost.

phase. Thus the second phase problem size is reduced greatly, because only resource assignments have to be computed.

### 4.4.3 Space and Time Decomposition

In this decomposition, the scheduling problem solved in two phases, namely, FU assignment followed by time assignment. This has the effect of optimizing the FU cost and shift registers' width before optimizing the depth of shift register files.

In the ILP scheduler, the formulation for space assignment consists of Equations 4.6 and 4.7. The objective function used in this phase is $\sum_{j=0}^{N-1} W_j$, where the $W_j$'s are given by Equation 4.15. Thus, the FU assignment phase reduces the sum of widths of the FUs. Note that this minimizes both the FU cost and the width of shift register files. Now, the time assignment can be performed by forming an ILP problem that includes all equations described in Section 4.3.3, and explicitly specifying the values for $R_{i,j}$'s obtained from the FU assignment phase.

Figure 4.5 illustrates a negative effect of phase ordering the scheduling problem into FU assignment followed by time assignment. Figure 4.5(a) shows part of the dataflow graph of a loop. There are two 16-bit adds feeding subtract operations and

a 32-bit add feeding a subtract operation. Suppose the machine has a budget for 2 adders and 2 subtractors and let the subtract operations be identical in width. The goal of FU assignment phase is to minimize the FU costs. Figures 4.5(b) and 4.5(c) show two possible assignments which result in the same FU cost of a 32-bit adder, a 16-bit adder and two subtractors. The crucial difference however is that operation $A$ is assigned to the 32-bit adder in Figure 4.5(b) and the 16-bit adder in figure 4.5(c). Note that both these assignments result in the same FU cost, and there is no way for the FU assignment phase to differentiate between these two solutions. Now consider the time assignment phase. Suppose that, due to other data dependencies, the only possible time assignment is as shown in either of the Figures 4.5(b) or (c). The separation of the operations due to the schedule in Figure 4.5(b) results in $16 \times 2 + 32 \times 3 = 128$ register bits. However, the schedule in Figure 4.5(c) results only in $16 \times 3 + 32 \times 2 = 112$ register bits. Thus, phase ordering could result in some sub-optimality.

## 4.5 Experimental Results

Loop kernels from several application domains are used to evaluate cost sensitive modulo scheduling. `Idct`, `dequant` and `dcacrecon` are loops from MPEG-4; `fsed`, `sobel`, and `sharp` are image processing loops; `blowfish` and `sha` are used in encryption applications; `lyapunov` is a mathematical kernel; and `viterbi`, `fft`, `fir`, and `iir` are commonly used in signal processing. The sizes of the loops range from 24 operations for `iir` up to 120 operations for `idct`. In general, loops for these applications can have intra loop code and may not be perfectly nested. For the experiments,

we manually convert the loop kernels to a single perfectly nested `for` loop. Only the innermost loop is considered for modulo scheduling. The numbers reported below correspond to hardware generated for the innermost loop only.

For each benchmark, we use the compiler-directed loop accelerator synthesis system described in Section 3.3. After FU allocation, various cost sensitive scheduling algorithms are evaluated. From the resulting schedules, the hardware datapath and control path is generated and the resulting RTL is synthesized to obtain gate counts. Synthesis is performed with Synopsys Design Compiler in $0.18\mu$ technology. A 200-MHz clock rate is assumed.

The ILP scheduler is used for most experiments; however, the BNB scheduler is used for the experiments that vary the cost objectives or partitioning method. The two schedulers are both exact solutions; thus, we do not compare them with each other. Their use in certain experiments is a software engineering decision as some experiments are more amenable to one formulation or the other.

The first experiment, shown in Figure 4.6, evaluates the effectiveness of the different decomposition methods in reducing hardware cost. The baseline in this experiment is the hardware resulting from the naïve, iterative modulo scheduler [58] followed by a stage scheduling postpass [14] as implemented in the Trimaran compiler framework [67], and is represented by 1.0. This baseline result is shown by the first bar of each benchmark; all bars are divided into three segments, representing the contribution of MUXes, storage, and FUs to the overall cost. The remaining bars are as follows: the second bar shows the greedy algorithm described in Section 4.3.1; the

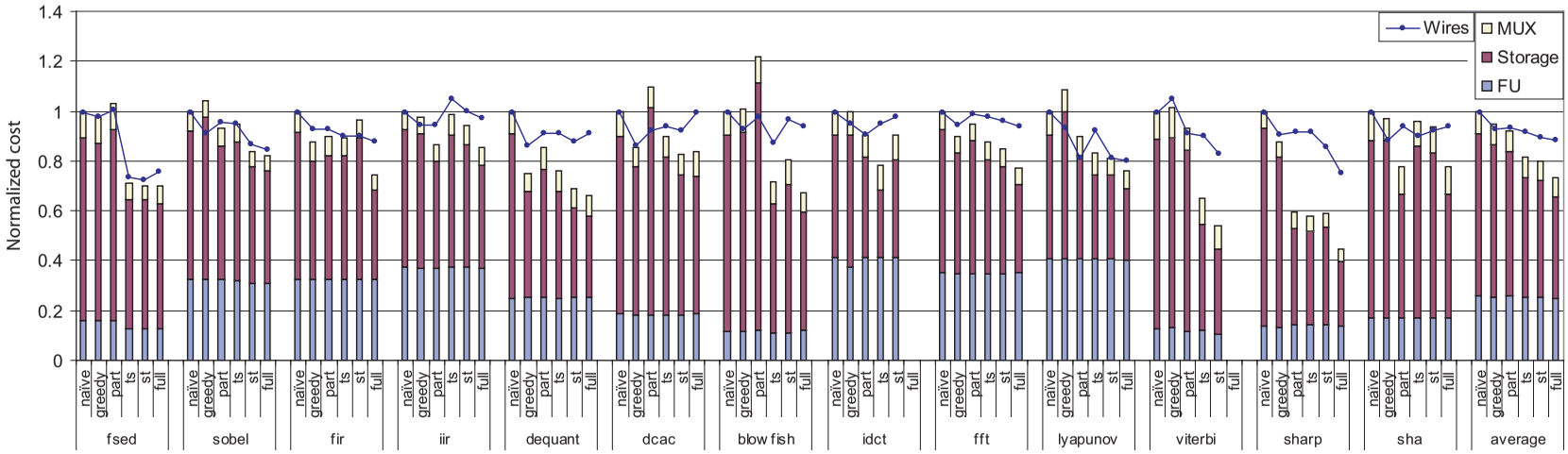


**Figure 4.6:** Hardware cost breakdown of loop accelerators synthesized using various scheduling techniques, relative to naïve scheduler.

third is the partitioned scheduler described in Section 4.4.1, using the priority-based partitioning method with a set size of 16 operations; the fourth is the time-space decomposition described in Section 4.4.2; the fifth is the space-time decomposition described in Section 4.4.3; and the sixth bar shows the optimal solution. Note that for some large benchmarks (`idct` and `viterbi`) this value could not be obtained due to the problem complexity, emphasizing the need for problem decomposition. The number of interconnect wires relative to the naïve scheduler is also shown in this figure as lines superimposed on the bars.

In this graph, FU cost does not differ significantly across schedulers. This is because FU capabilities are fixed prior to scheduling and most schedules result in the same or similar FU cost. The overall gate savings is significant in many benchmarks. The time-space decomposition scheduling achieves gate savings of 42% for `sharp`.

The greedy scheduler achieves only 5% gate savings on average and sometimes performs worse than the naïve scheduler. This is because it considers only one operation at a time and can be trapped in local minima. The average gate savings achieved by the partitioned, time-space and space-time scheduling methods are 8%, 19% and 20% respectively. In general, the time-space and space-time decomposition methods perform well as they are able to consider all operations at once. This is an advantage because the final machine cost is due to the combined effects of all operations rather than individual scheduling decisions. The partitioned cost sensitive scheduler results in slightly more gates than the naïve scheduler for some benchmarks like `fsed`, `dcacrecon`, and `blowfish`. This is due to the locally greedy nature of

the decomposition. Also, for large benchmarks, the fixed-sized operation sets make up smaller fractions of the whole loop and thus the algorithm becomes greedier as it "sees" less of the loop at once.

The optimal scheduler achieves 27% savings over the naïve scheduler. For some benchmarks (`iir`, `sha`) the partitioned scheduler performs near optimal. The time-space and space-time decomposed schedulers are able to achieve near optimal for many benchmarks while only requiring a fraction of the runtime. Both time-space and space-time schedulers produce high quality solutions and can practically handle large problem sizes. Thus, we believe these methods to be the best choices for accelerator synthesis. The two perform differently according to the application characteristics: space-time performs better for loops with more bitwidth variation (`sobel`, `viterbi`) while time-space performs better for loops with more register lifetime variation (`blowfish`, `idct`).

Generally, the number of wires decreases as gate count decreases. On average, the wire savings achieved for the three decomposed scheduling methods are 7%, 8%, and 10% for partitioned, time-space, and space-time, respectively. In many cases, the wire cost of the optimal solution is higher than the wire cost for one of the decomposed solutions; this is because the optimal scheduling formulation does not account for wire cost.

The next experiment shows the effect of changing the hardware cost objective. The objective discussed thus far has been minimizing the sum of logic (storage and FUs) and wires. The weights of these components can be modified; for example, if

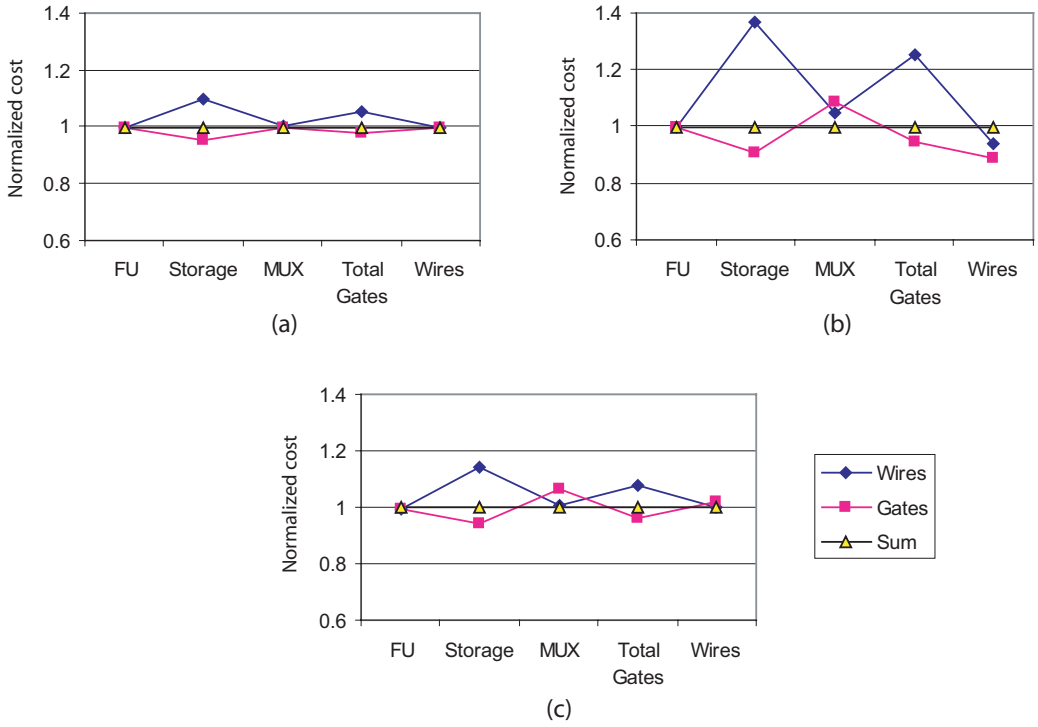


**Figure 4.7:** Effect of different cost objectives on (a) `iir`, (b) `sha`, and (c) average across all benchmarks.

interconnect cost is a dominating factor, the weight of the interconnect wires can be increased as a fraction of overall cost. The BNB scheduler can naturally accommodate these varying cost components. Figure 4.7 shows the breakdown of FU, storage, MUX, and wire costs relative to the baseline which optimizes the sum of these components. Each curve represents the machine resulting from scheduling with a certain cost objective; optimizing wires alone and optimizing logic gates (storage + FU) alone are presented. In Figure 4.7(a), `iir` is shown; when optimizing for wires, the cost of storage increases while wire cost decreases slightly. Conversely, optimizing for gates reduces the storage cost but increases wires slightly. Figure 4.7(b) shows the `sha` benchmark; interestingly, optimizing for gates reduces the number of wires. This is a

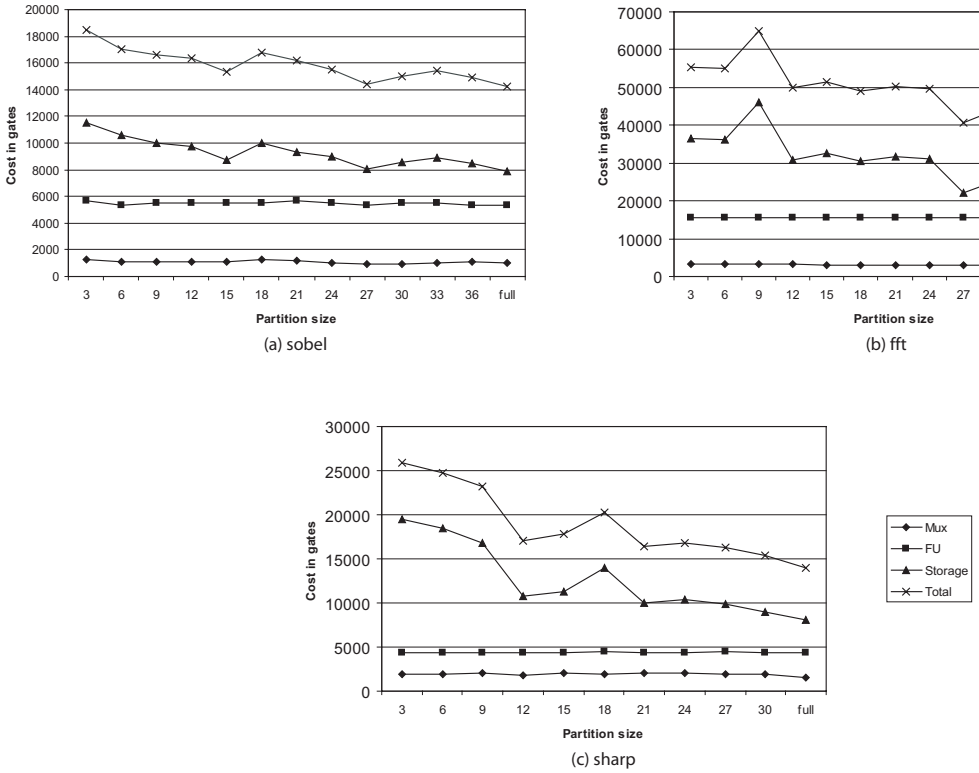


**Figure 4.8:** Effect of partition size on hardware cost.

product of the problem decomposition, which is imperfect – the baseline scheduler is unable to exploit wire sharing even though the gate optimizing scheduler happens to do so successfully. Figure 4.7(c) shows the average across all benchmarks; note that the wire optimizing scheduler did not save wires on average (though the wire count remains low). This is because jointly optimizing both gates and wires naturally results in good wire sharing (as fewer connections are made between fewer logic gates), and optimizing only wires does not improve on this for most benchmarks.

To show the effect of set size on the partitioned scheduler discussed in Section 4.4.1, Figure 4.8 shows the hardware cost of scheduling the `sobel`, `fft`, and `sharp` benchmarks with varying set sizes. The set size is varied from 3 operations per set up to

"full", where all operations are in one set. Each graph shows four lines, representing the FU, MUX, storage, and total gate costs at each set size. First, note that for these benchmarks, FU cost remains largely constant as there is little bitwidth variation among the data values, and no width specialization is performed. Second, the storage cost is where the scheduler is able to take the most advantage of larger set sizes. Third, the hardware cost decreases as set size increases, closely tracking the storage cost decrease. As expected, with larger set sizes, the scheduler is able to exploit hardware sharing across more operations at once. However, the overall cost generally nears optimal before the partition size becomes very large. For example, for `sobel`, a partition size of 15 gives a gate cost within 6% of optimal. Thus, the scheduler is often able to obtain good results while partitioning the operations into small sets.

In the ILP scheduler, CPLEX was used to solve the ILP formulations. A time limit of 6 hours was enforced for the ILP formulations leading to fully optimal solutions. CPLEX reports the best solution seen so far when the time limit expires. Thus the numbers reported for the optimal solution in Figure 4.6 correspond to this best solution. For the time-space and space-time decompositions, CPLEX runtimes were between 30 seconds for the smaller benchmarks like `fir` to 2 hours for the larger benchmarks like `idct` and `viterbi`. The CPLEX runtimes for partitioned ILP formulation were less than a second for smaller partitions sizes and a maximum of 80 minutes for the bigger partition sizes, irrespective of the benchmarks. Note that the time taken by the rest of the compiler phases is non-significant (less than a minute)

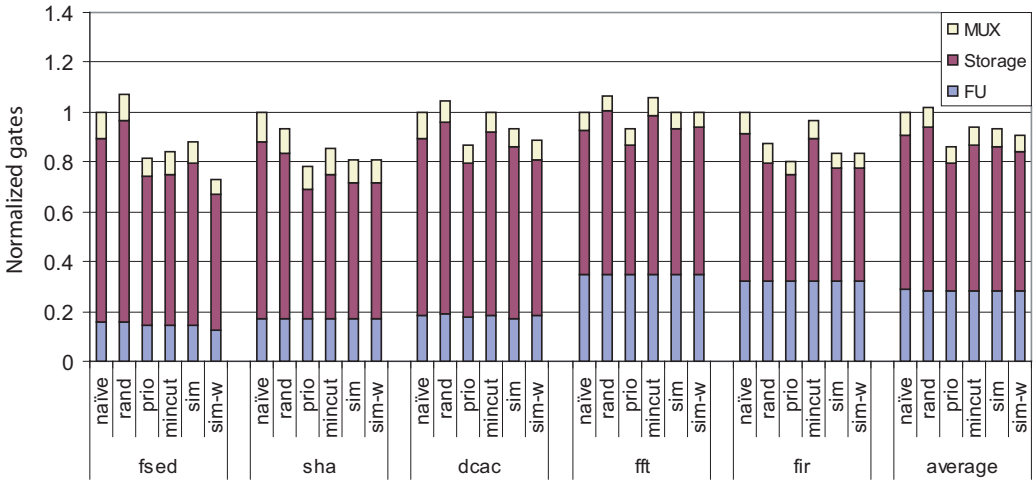


**Figure 4.9:** Cost breakdown for various partitioning methods.

compared to the CPLEX runs.

For the partitioned scheduler, various partitioning strategies are investigated as discussed in Section 4.4.1.1. Figure 4.9 shows the resulting hardware cost of various partitioning methods for select benchmarks and the overall average. A partition size of 8 is used in these experiments. In the graph, *rand* refers to random partitioning and usually performs worse than the naïve, cost unaware scheduler. *Prio* (priority-based) is the best on average, and is the method used in other experiments involving the partitioned scheduler. *Mincut* refers to a standard graph partitioner which attempts to minimize edge cuts; we use the Metis [32] partitioner. This method can perform poorly as it does not account for hardware cost. *Sim* (similarity-based) and *sim-w* (similarity-based with waste accounting) perform better than *mincut* but are hampered by backtracking effects as discussed in Section 4.4.1.2. Note that not all benchmarks could be scheduled using all partition methods (due to backtracking effects), so the cost average includes only benchmarks that could be scheduled using

all methods. Thus, the average cost of *prio* is not the same as in Figure 4.6.

## 4.6 Summary

This chapter proposes cost sensitive modulo scheduling in a loop accelerator synthesis system. Scheduling decisions must be made with the goal of decreasing the cost of hardware that is generated from the final schedule. Traditional modulo schedulers are not suitable in this context as they are unaware of the effect of scheduling decisions on hardware cost. Two exact solutions, branch-and-bound and ILP, are presented to solve this problem. In addition, three methods of decomposing the problem are presented that allow the algorithm to solve realistic problems. The decomposition techniques work either by partitioning the dataflow graph into smaller subgraphs and optimally scheduling the subgraphs, or by splitting the scheduling problem into two phases, time slot and resource assignment. All decomposition methods were successful at making increasing problem sizes tractable, and depending on the application, different decomposition methods performed better than others. Since the final cost depends on the combined effects of all operations, the time-space and space-time methods, which consider all operations together, worked best. Overall, cost sensitive modulo scheduling increases hardware efficiency of automatically synthesized loop accelerators by an average of 8–20%, with individual savings of up to 42% over a naïve scheduler.

# CHAPTER 5

# Multifunction Accelerator Design

## 5.1 Introduction

There is a growing push to increase the functionality of special-purpose hardware. Many applications that run on portable devices, such as wireless networking, do not have one dominant loop nest that requires acceleration. Rather, these applications are composed of a number of compute-intensive algorithms, including filters, transforms, encoders, and decoders. Further, increasing capabilities, such as supporting streaming video or multiple wireless protocols, places a larger burden on the hardware designer to support more functionality. Dedicated accelerators for each critical algorithm could be created and included in a system-on-chip. However, the inability to share hardware between individual accelerators creates an inefficient design. Processor-based solutions are the obvious approach to creating multi-purpose designs due to their inherent programmability. However, such solutions do not offer the performance, cost, and energy efficiency of accelerators as there is an inherent overhead

to instruction-based execution.

The focus of this chapter is on automatic design of multifunction loop accelerators from high-level specifications. The goal is to maintain the efficiency of single-function accelerators while exposing opportunities for hardware sharing across multiple algorithms. By building one accelerator that can run multiple loops, overall hardware cost savings can be realized when the loops are to execute disjointly. In addition, if the loops are not executing disjointly, it is possible to trade off hardware savings with performance. For example, if two loops in a streaming application are both non-critical in terms of the overall application pipeline, it may be beneficial to merge them into one accelerator. Similarly, in a multithreaded application, it may be beneficial to merge two loops from different threads into one accelerator which is time-multiplexed across the threads.

To create multifunction designs, the single-function system is extended using three alternate strategies. The simplest strategy is to create individual accelerators for each algorithm and place them next to each other. This method is referred to as a summed design, and is the baseline for comparison. The second strategy is to again create individual accelerators for each algorithm. The data and control paths for each accelerator are then intelligently unioned together to create a single design capable of all algorithms. Finally, the third strategy is to perform joint cost-aware synthesis of all algorithms. We employ an integer linear programming (ILP) formulation to find a joint solution with optimal estimated cost. A consequence of the joint scheduling strategy is that synthesis time and memory usage may become prohibitive for large

loop bodies or large numbers of loops. Each successive strategy represents a more complex approach and hence has more potential to exploit sharing opportunities.

## 5.2 Synthesizing Multifunction Accelerators

Multifunction design refers to generalizing a loop accelerator to support two or more loop nests. One obvious approach to creating a multifunction accelerator is to separately design accelerators for the individual loops, and then place these loop accelerators side by side in silicon. The area of the final accelerator would be the sum of the areas of the individual accelerators. However, by creating an accelerator with a single datapath that can support multiple loops, more hardware sharing can be achieved while continuing to meet the throughput constraints of both loops.

The cost of a multifunction accelerator is affected by the individual functions in several ways. First, the execution resources required by the multifunction accelerator must be a superset of the resources required for each individual accelerator. Since the multiple functions will not be executing simultaneously, any resources common to the individual accelerators need only be instantiated once in the combined accelerator. Effectively, the multifunction accelerator should have the union of the FUs required by the individual accelerators. Second, the cost of the SRFs is sensitive to how the hardware is shared across functions. Since every FU has an SRF at its output, and the SRF has the bitwidth of its widest member and the depth of its value with the longest lifetime, there is a potential for careless sharing to result in large, underutilized SRFs. Third, one advantage of a customized ASIC is that there are few control signals

**Constraints:**

For each loop $a$:

| | | | |
|---|---|---|---|
| Time slots: | $\sum_{s=0}^{II_a-1} X_{i,s,a} = 1$ | $\forall i \in \{1, N\}$ | (5.1) |
| Resources: | $\sum_{f=1}^{M_f} R_{i,f,a} = 1$ | $\forall i \in \{1, N\}$ | (5.2) |
| | $\sum_{i \in I_f} R_{i,f,a} \times X_{i,s,a} \leq 1$ | | (5.3) |
| Dependences: | $t_{j,a} + d_{i,j,a} \times II_a - t_{i,a} \geq l_{i,j,a}$ | $\forall (i,j,a) \in E_a$ | (5.4) |
| SRF Depth: | $LT_{i,a} \geq t_{i',a} - t_{i,a} + II_a \times d_{i,i',a} - l_{i,i',a} + 1$ | $(i,i',a) \in E_a$ | (5.5) |
| | $D_f \geq R_{i,f,a} \times LT_{i,a}$ | $\forall i$ assigned to $f$ | (5.6) |
| FU/SRF Width: | $W_f \geq R_{i,f,a} \times BW_{i,a}$ | $\forall i$ assigned to $f$ | (5.7) |

**Objective:**

Cost: $$Cost = \sum_f D_f \times W_f + fu\_cost_f \times W_f \quad (5.8)$$

**Definitions:**

$t_{i,a} = \sum_{s=1}^{II_a-1} s \times X_{i,s,a} + II_a \times k_{i,a}$ $\quad$ $l_{i,j,a}$ = latency on edge $(i,j)$

$d_{i,j,a}$ = iteration distance on edge $(i,j)$ $\quad$ $M_f$ = number of FUs of type $f$

**Figure 5.1:** ILP formulation for joint scheduling.

that need to be distributed across the chip, since the datapath is hard-wired for a specific loop. When multiple loops come into play, not only must the datapath be able to support the computation and communication requirements of each loop, but the control path must be capable of directing the datapath according to which loop is being executed.

Two techniques are presented to increase hardware sharing: joint scheduling and the union of individually designed accelerators.

### 5.2.1 Joint Scheduling

Since the cost of the multifunction datapath depends on the combined schedules of all loops, an ideal scheduler should look at all loops simultaneously and schedule them to minimize the total hardware cost (while meeting their individual II constraints). This is referred to as *joint scheduling*; the scheduler is aware that all loops will execute on the same hardware, and is therefore able to make scheduling decisions that maximize hardware sharing across loops.

An ILP formulation for joint scheduling is used. This formulation is similar to the modulo scheduling formulation with extensions to minimize accelerator cost as described in Section 4.3.3. These formulations are extended to consider multiple loops simultaneously. For each loop $a$ under consideration, integer variables to represent time and FU assignment are introduced. For every operation $i$ in loop $a$, $II_a$ mutually exclusive binary variables $X_{i,s,a}$ represent the time slot $s$ in the modulo reservation table (MRT) that the operation is scheduled. The integer variables $k_{i,a}$ represent the stage in which operation $i$ is scheduled. Binary variables $R_{i,f,a}$ represent the assignment of operation $i$ in loop $a$ to the FU $f$. The set of variables $X_{i,s,a}$, $k_{i,a}$, and $R_{i,f,a}$ represent complete modulo schedules for the loops. Other auxiliary variables are introduced to represent the cost of the hardware.

The full ILP formulation for joint scheduling is shown in Figure 5.1. The formulation consists of basic constraints (Equations 5.1 through 5.4) that ensure a valid schedule, and auxiliary constraints (Equations 5.5 through 5.8) that are used to compute the cost of the resulting hardware. Note that Equations 5.3, 5.6, 5.7, and 5.8

have non-linear components; these may be linearized using standard techniques as was done in Section 4.3.3.

The schedule validity constraints for individual loops are totally independent and represented using disjoint variables. However, there is only one set of variables that represent the hardware cost. For example, the cost of an FU is represented by a single variable, but depends on FU assignment of operations in all loops. Similarly, SRF costs are modeled using a single set of variables.

### 5.2.2 Union of Accelerators

The joint scheduler considers the effects on hardware cost of the scheduling alternatives for operations in all loops, and selects combinations of alternatives to minimize cost. This is computationally complex, because the number of possible schedules grows exponentially as the number of loops increases (since the scheduling alternatives of operations in different loops are independent). As a result, joint scheduling with ILP is impractical for large loop bodies or high numbers of loops.

Instead, the multi-loop scheduling problem may be divided into two phases to reduce its complexity. First, loops are scheduled individually and a single-function accelerator is designed for each loop; then, the accelerator datapaths are unioned into one multifunction datapath that supports all loops. This phase ordering can result in high quality designs, as the single-function accelerator costs are first minimized, and then hardware sharing across loops is exploited during the accelerator union. Synthesis runtimes are reduced significantly as it is no longer necessary to consider

all schedules simultaneously.

The union phase is accomplished by selecting an FU and its corresponding SRF from each single-function accelerator and combining them into a single FU and SRF in the resultant accelerator. The new FU has the bitwidth and functionality to execute all operations supported by the individual FUs being combined. Similarly, the new SRF has sufficient width and depth to meet the storage requirements of any of the SRFs being combined. This process is repeated for the remaining FUs and SRFs until all of them have been combined. At this point, the resulting accelerator supports all of the functionality of the individual accelerators.

The cost of the multifunction accelerator is affected by the specific FUs and SRFs that are combined. For FUs, the ideal case occurs when FUs with identical functionality and bitwidth from $k$ individual accelerators are combined into a single FU. This FU in the multifunction accelerator represents a cost savings (by a factor of $k$) over the single-function accelerators due to hardware sharing. When FUs with differing functionality are combined, no cost savings is achieved in the FUs, but this may enable cost savings in the corresponding SRFs. In the case of SRFs, maximal sharing occurs when two or more SRFs with similar bitwidths and numbers of registers are combined; in this case, only a single SRF is required in the multifunction accelerator where several were needed by the single-function accelerators.

#### 5.2.2.1 Positional Union

The most straightforward union method is a *positional union*, where the FUs in each accelerator are ordered by functionality (multiple FUs with the same functionality have no particular order), and FUs and SRFs in corresponding positions are selected for combination. The first FU and SRF in accelerator 1 are combined with the first FU and SRF in accelerator 2 to form the first FU and SRF in the multi-function accelerator, and so on. This union method yields good hardware sharing in the FUs, as FUs with identical functionality are combined, and the number of unique FUs in the resultant accelerator is therefore minimized. However, it does not account for FU width, nor does it attempt to improve hardware sharing in the SRFs. Sharing in the SRFs occurs by chance, if the dimensions of the SRFs being combined happen to be similar.

In Figure 5.2, an example of positional union is shown on the left. Here, each single-function accelerator has two ADD FUs and an AND FU. The FUs and SRFs have varying widths and depths, and thus varying costs, as shown to the right of each FU and SRF. The FUs of the two accelerators are combined according to functionality, and the resulting accelerator is shown on the lower left of the figure. Each FU and SRF in the unioned accelerator is sized to accomodate the corresponding FUs and SRFs from the single-function accelerators directly above them.

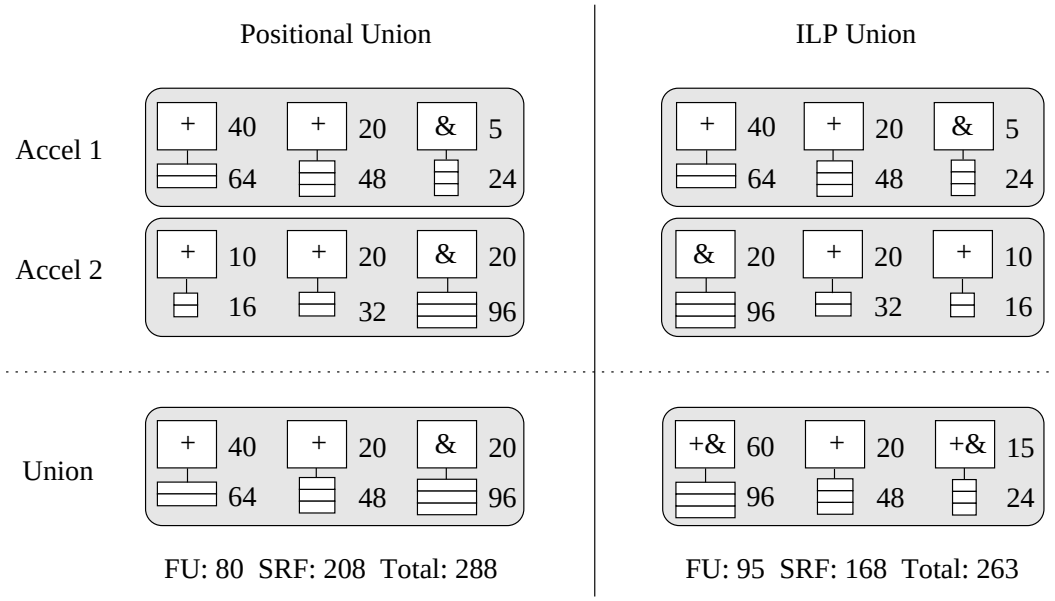


**Figure 5.2:** Example of union techniques. Two single-function accelerators, each with three FUs, are combined using positional (left) and ILP (right) methods. The cost of each FU and SRF is shown on its right.

#### 5.2.2.2 ILP Union of Accelerators

An improved union method to increase hardware sharing should consider all permutations of FUs (and corresponding SRFs) from the different loops, and select the permutation that results in minimal cost, considering both FU and SRF costs. This can be formulated as an ILP problem where binary variables are used to represent the possible pairings of FUs and SRFs from different loops. In this section, the combination of two loops to form a multifunction accelerator will be examined. Unions of more than two loops will be considered in the next section.

Assume that both single-function accelerators have $N$ FUs. (If one accelerator has fewer FUs than the other, zero-width FUs may be added to make the number of FUs equal.) Then, $N^2$ binary variables $x_{ij}$ may be used to represent the combination of FU $i$ from the first loop with FU $j$ from the second loop (along with their corresponding

SRFs). For example, if $x_{11} = 1$, the first FUs in both accelerators will be combined in the multifunction accelerator. In addition, the following equations ensure that each FU is selected exactly once for combination with another FU:

$$\sum_{1 \le j \le N} x_{ij} = 1 \quad \forall i, \qquad \sum_{1 \le i \le N} x_{ij} = 1 \quad \forall j \tag{5.9}$$

Next, the objective function is defined so that the overall cost of the multifunction accelerator is minimized. This cost consists of two components: FU cost and SRF cost. Define variables $F_{ij}$ as the cost of the FU resulting from the combination of FU $i$ from loop 1 and FU $j$ from loop 2. Depending on the functionality and bitwidth of these FUs, this cost can vary from the maximum cost of the two FUs up to their sum. Also, define variables $R_{ij}$ as the cost of the SRF resulting from the combination of the SRFs corresponding to these two FUs. Then, the objective function is the minimization of the following:

$$Cost = \sum_{\forall i,j} (F_{ij} + R_{ij}) \times x_{ij} \tag{5.10}$$

By minimizing (5.10) subject to the constraints (5.9), a combination of the FUs and SRFs of two loops is chosen that minimizes the cost of the multifunction accelerator.

The right side of Figure 5.2 shows an example of the ILP union. The single-function accelerators contain the same FUs and SRFs as in the positional union case, but they are combined differently. The resulting FU cost is higher than the FU

cost from the positional union, because dissimilar FUs were combined and thus less hardware sharing in the FUs is achieved. However, the overall cost is lower as the SRF hardware is shared more intelligently.

#### 5.2.2.3 Union of Multiple Accelerators

In the case where more than two loops are being combined, two strategies may be applied to extend the union technique. The first strategy is referred to as *pairwise union* and consists of first combining two accelerators to form a (temporary) multifunction accelerator. This temporary accelerator is then combined with the third single-function accelerator to form a new multifunction accelerator that supports all three loops. This process is continued, combining the new temporary accelerator with remaining single-function accelerators, until all desired loops have been combined into one multifunction accelerator.

The second method is referred to as *full union* and extends the ILP formulation given in the previous section. Given $k$ loops, there are $N^k$ binary variables $x_{i_1 \ldots i_k}$ that represent the combination of FUs $i_1$, ..., $i_k$ from accelerators 1, ..., $k$, respectively. Constraints (5.9) and objective (5.10) are extended to reflect the additional loops. The solution consists of the $N$ variables set to 1 which represent the specific combinations of FUs and SRFs which minimize the final hardware cost.

The advantage of full union is that it simultaneously considers all single-function accelerators together, and determines the best permutation of FUs to minimize the overall FU and SRF cost. However, the downside is that the number of variables

is exponential in the number of loops. Therefore, the full union quickly becomes infeasible for higher numbers of loops. Conversely, the pairwise union method may become trapped in local minima as it only considers two accelerators at a time during combining. We find experimentally that the pairwise union performs nearly as well as the full union in terms of final hardware cost, and its runtime is significantly faster due to its lower complexity.

## 5.3 Experimental Results

Kernels from four different application domains are used to evaluate the loop accelerator designs. `Sharp`, `sobel`, and `fsed` are image processing algorithms. `Idct`, `dequant` and `dcacrecon` are computationally intensive loops extracted from MPEG-4. `Bffir` and `bfform` are loops from the beamformer benchmark of the StreamIt suite [66]. `Viterbi`, `fft`, `convolve`, `fmdemodulator`, `fmfilter`, and `fir` are loops from the signal processing domain. To evaluate multifunction designs, loops from within the same application domain are combined, as they would likely be part of the same larger application accelerator.

For each machine configuration, we use the synthesis system described in this dissertation to design loop accelerators and generate RTL. The resulting Verilog is synthesized using the Synopsys design compiler in 0.18$\mu$ technology. All designs were synthesized with a 200-MHz clock rate. For all experiments, performance is held constant and is specified by the II value. A typical II is selected for each benchmark (for example, II=4 for `sobel` and II=8 for `idct`), and multifunction hardware is

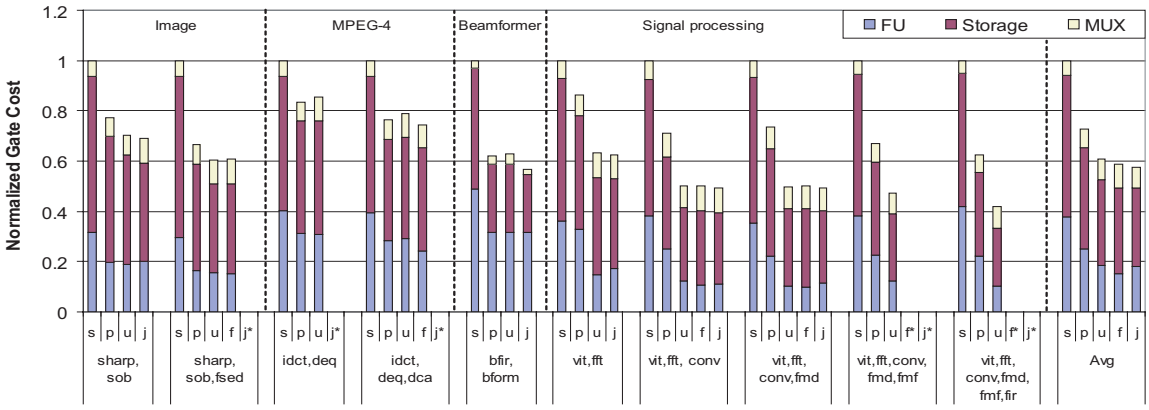


**Figure 5.3:** Gate cost of multifunction accelerators designed using sum (s), positional union (p), pairwise union (u), full union (f) (not shown for 2-loop combinations), and joint scheduling (j). * indicates the synthesis did not complete due to problem complexity.

synthesized for combinations of benchmarks within the same domain. Gate counts are used to measure the cost of each accelerator configuration.

Figure 5.3 shows the cost in gates of multifunction loop accelerators designed using various scheduling methods. Each group of bars represents a benchmark combination, showing, from left to right, the sum of individual accelerators (s), the positional union of individual accelerators (p), the pairwise union (u), the full union (f), and the joint solution (j). When only two accelerators are combined, the full union is not shown as it is identical to the pairwise union. The bars are normalized to the sum of the cost of individual accelerators for that benchmark group. In addition, each bar is divided vertically into three segments, representing the contribution of FUs, storage, and MUXes to the overall cost. Since the joint solution relies on an ILP formulation with a large number of variables and constraints, it did not complete for some benchmark groups (labeled j$*$). Also, for groups containing more than four benchmarks, the full

union becomes infeasible (labeled f∗).

The first bar of each set represents current state-of-the-art multifunction accelerator design methodologies, i.e., creating single-function accelerators for each loop. Each single-function accelerator is designed using a cost-aware scheduler to minimize cost [19]. Thus, the difference between this bar and the other bars in each group represents the savings obtained by hardware sharing in multifunction designs. Since II is fixed for each benchmark, all multifunction designs in a group have the same performance, and hardware savings is essentially free. (However, note that additional multiplexers may increase critical path delay; this is discussed later in this section.) As the graph shows, the hardware savings is significant and increases with the number of loops. Up to 58% savings is achieved for the signal processing benchmark group, and 43% savings is achieved on average across all groups. Some groups (e.g. `idct` and `dequant`) exhibit less multifunction savings because the sizes of the two loops differ significantly, decreasing the amount of potential sharing.

On average, the pairwise and full union methods yield significantly lower-cost hardware than the positional union and are very close to the cost obtained with joint scheduling. However, in a few cases (most notably the benchmark groups containing `idct`), the positional union yields a lower cost than the more intelligent unions. This is due to two factors: first, MUX cost is not considered during the union phase and can affect the final cost; and second, the FU costs being minimized in the union phase are estimates, and actual FU costs may differ slightly when the design is synthesized into gates. In most benchmark groups, the pairwise union yields hardware that is

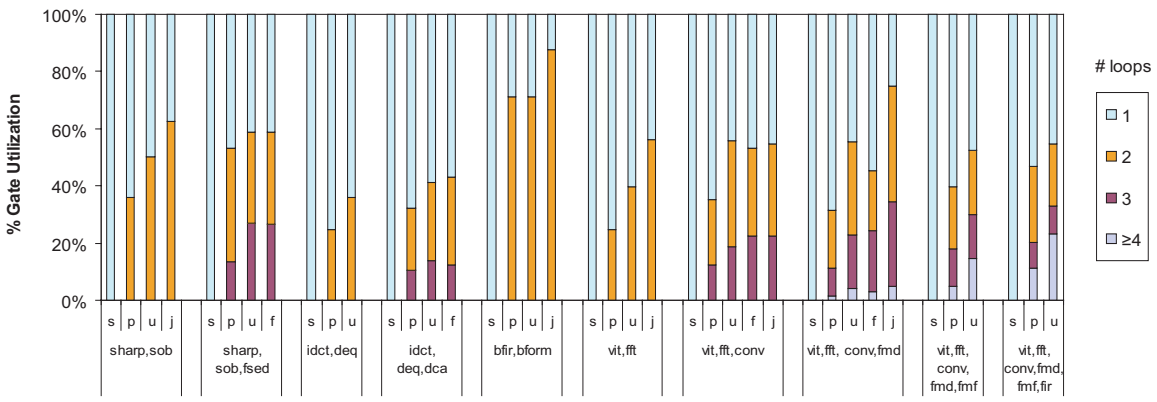


**Figure 5.4:** Degree of sharing of multifunction accelerator gates across loops.

equivalent in cost to the full union. Thus, pairwise union is an effective and tractable method of combining accelerators.

An area in which the multifunction accelerator does not improve on the individual accelerators is in the MUX cost. Although the multifunction accelerator has fewer FUs (and thus fewer MUXes) than the sum of individual accelerators, each MUX must potentially select from more inputs, as more operations execute on each FU.

Figure 5.4 shows the amount of hardware sharing in each of the multifunction accelerators synthesized in Figure 5.3. Each accelerator is represented by a bar which is divided vertically to show the fraction of gates used by 1 loop, 2 loops, etc. In general, lower cost accelerators have a higher fraction of gates used by multiple loops. Some interesting points to note are when sharing across loops increases, but the corresponding hardware cost does not decrease much (e.g. `vit-fft` when moving from union to joint). This occurs because, even though the joint scheduler is better able to share hardware across loops, the union method often has better hardware sharing within each loop (since the single-function accelerators are designed separately).

Thus, hardware sharing still occurs in the union case, and the cost remains low.

Overall, runtimes for the synthesis system ranged from 20 minutes up to several hours on Pentium 4 class machines. The runtimes were dominated by the first step, generation of cost-efficient single-function accelerators; the runtime of the union phase was negligible for positional and pairwise unions, and up to 1 hour for the full union. The joint scheduler was allowed to run for several days; the bars missing from Figure 5.3 took longer than 5 days to run.

A side effect of multifunction designs is that additional interconnect is necessary to accomplish sharing in the datapath. The additional interconnect consists mostly of wider MUXes at the inputs of FUs. This can affect critical paths through the accelerator datapath and hence the maximal clock rate of the design. On average, the critical path delay in multifunction designs increased by 4% over the single-function designs. The largest critical path increase occurred in the signal processing group due to the increased resource sharing among the six loops. In this group, the length of the critical path increased by 12% over that of the single-function accelerator. All of the multifunction designs were able to meet the target clock rate of 200 MHz.

## 5.4 Summary

This chapter extends the synthesis system to create accelerators that support multiple loops. Cost savings is achieved by sharing hardware across loops while meeting the performance requirements of each loop. Union methods are presented to reduce the complexity of the scheduling problem. It is shown that intelligently

unioning single-function accelerators yields multifunction accelerators that are nearly optimal in cost. By evaluating accelerators designed for various application domains, average hardware savings of 43% are realized due to sharing of execution resources and storage between loops, with individual savings of up to 58%.

# CHAPTER 6

# Programmable Loop Accelerator Design

## 6.1 Introduction

As shown in previous chapters, the high performance and low power demands of emerging applications can often be met using hardwired solutions, e.g., ASICs such as loop accelerators. Most modern embedded systems employ ASICs for the most compute-intensive tasks. However, this is in direct conflict with an increasingly important characteristic: post-programmability. A programmable solution offers several key advantages. First, software implementations allow the application to evolve in a natural way after the chip has been manufactured due to changes in the specification, bug fixes, or the addition of new features. Second, multi-mode operation is enabled by running multiple different applications or variants of applications on the same hardware. Third, time-to-market of new devices is lower because the hardware can be re-used and hardware may be developed in parallel with software. And finally, chip volumes are higher as the same chip can support multiple products in the same

family.

The tradeoffs between performance, power, and programmability are at the heart of the hardware implementation choice that designers are forced to make. ASICs provide the highest performance and lowest energy solutions for specific problems. However, they offer little in the areas of programmability and hardware re-use due to the hardwired nature of the design. At the other end of the spectrum are processors and DSPs. Processors offer full programmability and thus the ability to execute a wide range of applications. But, processors offer poor energy efficiency and often cannot meet application performance requirements. ASICs typically offer 100-1000x more energy-efficiency for specific applications than processors. Middle-ground solutions offer the promise of high efficiency together with full programmability. However, they often fall short of these goals. For instance, FPGAs achieve extremely high performance for bit-level parallel computation. But, the overhead of gate-level reconfigurability often causes them to fall short in applications that have limited parallelism or rely on more expensive computations, such as multiplies.

A key question that this chapter investigates is: *How much programmability is really required in a design?* Programmability is generally thought of as a binary issue - either a design is programmable or not. Programmable designs support a wide range of applications while hardwired designs support a single algorithm implementation. An important insight is that semi-programmable solutions may be enough for many embedded designs. For example, video coding standards are typically developed years ahead of time by industrial consortiums [31]. These standards go through many

rounds of development and adjustment, but the core algorithm kernels often evolve at a relatively slow rate. At the same time, domain-specific hardware is often essential to achieve the necessary performance and energy efficiency. And, this customized hardware is neither appropriate nor efficient for applications outside the domain. Therefore, providing universal programmability may have little practical value.

Our approach is to push programmability into a highly customized hardware substrate to retain the high performance and energy efficiency of an ASIC, while offering a limited degree of post-programmability. The starting point is a stylized loop accelerator (LA) that is customized for a single application loop nest, as discussed in Chapter 3. The structure of the base LA template is generalized to create a semi-programmable solution, termed a programmable LA or PLA. However, the PLA datapath is still highly specialized with point-to-point interconnect, fixed-capability function units, and limited storage to retain its inherent efficiency characteristics. Such a platform cannot execute an arbitrary loop. Rather, the programmability objective is to map loops with similar computation structure onto a common hardware platform, such as two loops from the same application domain or a single loop that has undergone small to modest changes in composition.

This chapter contains the following contributions:

- An analysis of the evolution of several media applications to understand the programmability needs of customized hardware.
- A parameterized template for a PLA is developed. The template offers high degrees of customization to the target loop, while providing programmability

for a range of loops with similar computation structure.

- To automatically map loops onto PLAs, a constraint-driven modulo scheduling formulation is presented.
- The performance, area, and power efficiency of the PLAs are evaluated and compared to single-function LAs and the OR-1200 embedded processor for a range of compute-intensive loops.
- The programmability of the PLAs is evaluated across a range of loops and synthetically generated variations of these loops.

## 6.2 Motivation

### 6.2.1 Architecture Style vs. Efficiency

A wide range of architectures have been designed before to address the problem of providing high performance computation efficiently. These solutions maintain or sacrifice programmability to various degrees depending on the domain they target. This section describes some of these solutions and motivates the need for semi-programmable LAs.

Figure 6.1 shows the peak performance achievable by different architecture styles and their power efficiency. The x-axis in Figure 6.1 indicates programmability of different solutions. General purpose processors (GPPs), which fall on the lower right corner of the figure, are highly programmable solutions, but are limited in terms of

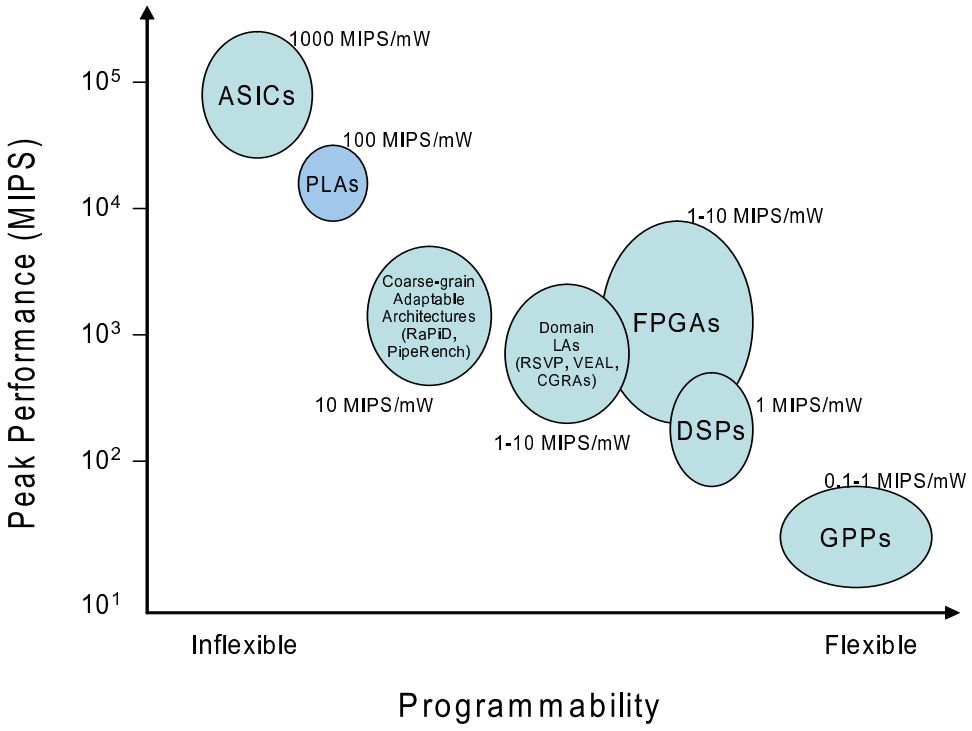


**Figure 6.1:** Peak performance and power efficiency of different architecture styles.

the peak performance they can achieve. Also, structures like instruction decoders and caches that are needed support programmability consume energy, resulting in a low computation efficiency of about 1 MIPS/mW for the Pentium M. On the other extreme of the spectrum are ASICs. ASICs are custom designed for a particular problem, without extraneous hardware structures. Thus, ASICs have high computational density with hard-wired control, resulting in high computation efficiency up to 1000 to 10000 times more than GPPs. The space between these two extremes is populated by different solutions that have varying degrees of programmability.

Digital signal processors (DSPs) [47, 64, 65] increase the computation efficiency by providing specialized features that optimize execution of signal processing algorithms. These features include special arithmetic operations like multiply-accumulate and bit

manipulation operations, hardware modulo addressing, and memory architectures optimized for streaming data. A wide range of DSP algorithms can be executed efficiently on these processors efficiently. DSPs typically offer an order of magnitude increase in power efficiency.

Domain loop accelerators are designed to execute computation intensive loops present in media and signal processing domains. Their design is close to a VLIW processor, but with a much higher number of FUs, and thus higher peak performance. Very long instruction words present in a control memory direct all FUs every cycle. However, the domain LAs have lesser flexibility compared to GPPs because only highly computationally intensive loops map well to them. Arbitrary control intensive code yields low computation efficiency on these architectures. Some examples of architectures in this design space are VEAL [9], RSVP [8], CGRAs [45, 53], and the Perception Processor [44].

FPGAs have fine grain logic blocks that can be reconfigured to perform various bit level logic and arithmetic functions. The fine grain reconfigurability allows FPGAs to be very flexible. Bit parallel computations present in domains like encryption can be performed very efficiently on FPGAs. However, complex integer and floating point operations do not map well on to FPGAs. Thus, for a set of domains, FPGAs are very flexible and highly efficient.

Coarse-grain adaptable architectures have coarser grain building blocks compared to FPGAs, but still maintain bit-level reconfigurability. The coarser reconfiguration granularity improves the computation efficiency of these solutions. However, non-

standard tools are needed to map computations onto them and their success has been limited to the multimedia domain. PipeRench [25] and RaPiD [13] are some examples of coarse-grain adaptable architectures.

The programmable solutions shown in Figure 6.1 are all "universally" programmable, allowing any loop to be mapped on to them, although at varying degrees of efficiency. There is a wide gap between the efficiency that can be achieved by ASICs and the efficiency that can be achieved by these programmable solutions. Section 6.2.2 shows that there are instances where there is a narrow requirement of flexibility. Using any of the above solutions is overkill for these instances as these solutions sacrifice too much efficiency for the needed flexibility. The PLAs proposed in this dissertation are positioned in the design space where a small but non-trivial amount of programmability as well as the high efficiency of ASICs are both required.

### 6.2.2 Programmability Case Study

As applications evolve over time, code changes are inevitable. Whether due to changing requirements, changing standards, bug fixes, or new features, software is constantly in flux. With hardwired solutions, every time the code in an accelerated loop changes, new hardware must be synthesized even if the changes are small and the dataflow between operations within the loop is substantially similar. By adding some programmability, the hardware can be made robust in the face of such changes. By looking at some loops from real applications, we can get a feel for what kinds of changes typically occur.

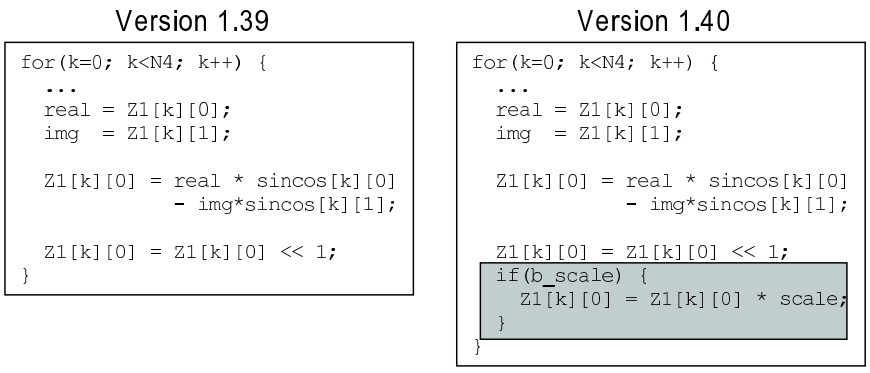


**Figure 6.2:** Feature Addition to `mdct.c` in `faad2`.

Figure 6.2 shows a loop from the `faad2` application, which is a commonly used free audio decoder for the Advanced Audio Coding (AAC) standard. The figure shows that between revisions 1.39 and 1.40 of the software, the loop has been modified with the addition of an if-clause, while the rest of the loop remains the same. This represents the addition of a new feature that requires certain new code in the loop to be guarded under a flag. To implement the if-clause, the hardware must have function units capable of performing load, multiply, and store. As these operations are already present elsewhere in the loop, the new code should ideally be executable on the same hardware, although the level of performance may be lower because the same hardware resources are being used to execute more operations. The additional control flow should not present a problem because the loop can be if-converted, and a compare operation is not required inside the loop because the if-condition is live-in.

Figure 6.3 shows another loop from the same application. In this case, the code changes from version 1.33 to 1.34 consist of sign changes on the right hand side of some assignment statements, as might occur in a bug fix. These sign changes correspond to dataflow changes in the loop, as some values now must go through a subtractor,

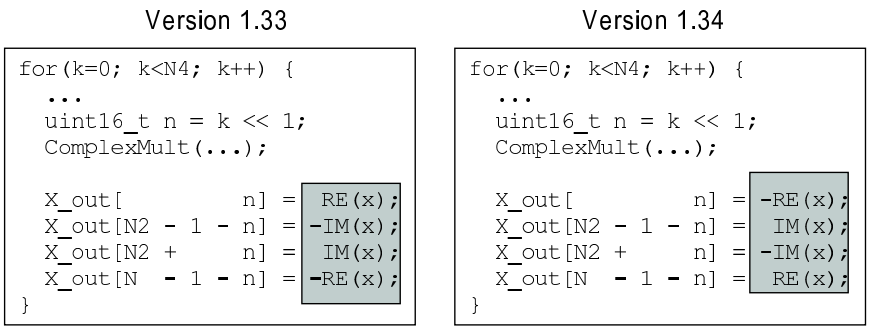


**Figure 6.3:** Bug-fix to `mdct.c` in `faad2`.

while other values should no longer go through a subtractor. (Alternatively, the dataflow changes could occur post-negation, with the same values being stored to different addresses.) In this case, the number of operations does not change, but the communication between operations changes, and the hardware should be flexible enough to accommodate this.

It can be seen that loops in real applications undergo minor changes over time. Since the changes do not alter the loops significantly, it is possible to design an efficient LA that remains usable after these changes are made.

## 6.3 From Single-function LA to Programmable LA

### 6.3.1 Single-function Accelerator

A single-function LA is used as a baseline. This accelerator is designed to execute a specific loop at a given performance level, and is not programmable. Then, starting from the single-function baseline, the datapath is generalized to create a more programmable design. The goal is to remove or relax the most restrictive parts of

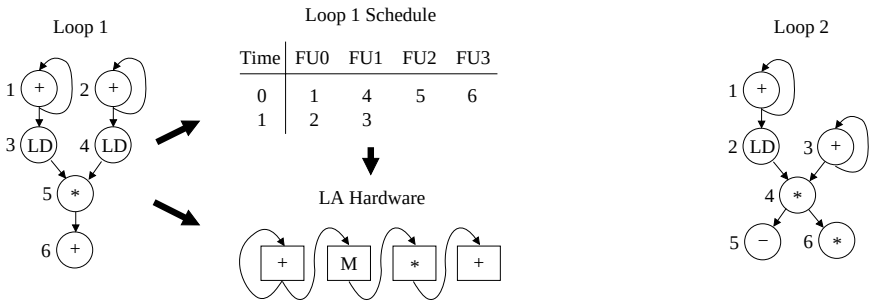


**Figure 6.4:** LA scheduling and synthesis example.

the architecture that limit programmability, while retaining the efficiency available through customization. This section describes the datapath generalizations used to create a PLA.

The left side of Figure 6.4 shows a portion of the loop from the FIR filter application. Assuming the given II is 2, the abstract architecture will have two adders, one memory unit, and one multiplier. When the operations in the loop are scheduled as shown in Figure 6.4, the resulting single-function LA hardware will be as depicted (registers are omitted from the figure for clarity). The connectivity within the LA is limited because only those connections required to support this schedule are created. For example, the input of the multiplier can only come from the memory unit.

Now, assume that a second loop (shown on the right) is to be mapped to the same LA. This second loop is somewhat similar to the first, in that it also contains adds, loads, and multiplies. However, the functionality is different, and the communication patterns between operations are different as well. For example, Loop 2 contains a subtract operation which did not exist in Loop 1, and also contains a dataflow edge from ADD to MUL, which also did not exist in Loop 1. The next subsection will

discuss the changes to be made to the LA datapath to make it programmable and support the execution of the second loop.

### 6.3.2 Programmable Loop Accelerator

To build a programmable loop accelerator (PLA), the datapath features of the single-function LA that are least flexible should be generalized in a power and area efficient manner. The next sections discuss these datapath characteristics.

#### 6.3.2.1 Functionality

The LA is limited by the opcode repertoire of the FUs. For example, if a new loop contains a subtract operation, but no FU is capable of performing subtraction, it will not be possible to map the new loop onto the LA. FUs can be generalized with low additional cost by adding functionality that is complementary to existing functionality. For example, any adder can be generalized to support both addition and subtraction with low additional cost. Other generalizations include broadening the opcode repertoire of logical, memory, comparison, and shift FUs to include all variants of those respective opcodes (e.g. all shift FUs are expanded such that they are capable of left and right arithmetic and logical shifts). The costs of FU generalization include increased hardware area and power consumption, as well as increased encoding requirements for the larger number of supported opcodes. For the example second loop of Figure 6.4, there is a subtract operation that is not supported by the single-function LA. By generalizing the adders to adder-subtractors, the functionality of the

second loop will be supported.

#### 6.3.2.2 Point-to-point Connectivity

A major area where the LA achieves efficiency wins is the point-to-point connectivity scheme. Only those connections that are needed to sustain the producer-consumer communications in the modulo schedule exist in the single-function LA. This means that not all FUs are able to communicate directly with other FUs, making it difficult to map new applications onto the hardware. Two techniques are used to relax this constraint. First, all FUs are given the ability to perform a MOV; that is, copy one of its inputs to its output. This allows values to be transferred from a source FU to a destination FU via intermediate FUs. Second, a low-bandwidth bus is created that connects all FUs in the accelerator.[1] This allows a single value transfer from any FU to any other FU each cycle. The bus is scheduled by the compiler and thus is not arbitrated. Such a global bus can be viewed as a fallback communication path, ensuring that communication from any FU to any other FU is possible. Thus, the programmability of a given accelerator (in terms of the number of different loops that can be mapped onto it) increases significantly; however, since the bus is low bandwidth, if a loop requires a large number of bus transfers, it will not be possible to achieve a schedule with low II (high performance).

The global bus incurs additional hardware cost as each register file contains a new read port which can place a value onto the bus, and each MUX contains a new input which allows the FU to read the value from the bus in addition to the existing

[1] This bus may be pipelined or organized in a hierarchical manner for larger accelerators.

point-to-point connections.

In Loop 2 of Figure 6.4, two of the communication paths are not supported by the single-function LA. Specifically, the edge from operation 3 to 4 cannot be mapped onto the LA because there is no wire from an adder to a multiplier, and the edge from operation 4 to 6 cannot be mapped because there is no wire from a multiplier to a multiplier. The $3 \rightarrow 4$ communication can be handled by inserting a MOV to pass the value from the adder through the memory unit to the multiplier. The $4 \rightarrow 6$ communication can be handled by passing the value on the global bus.

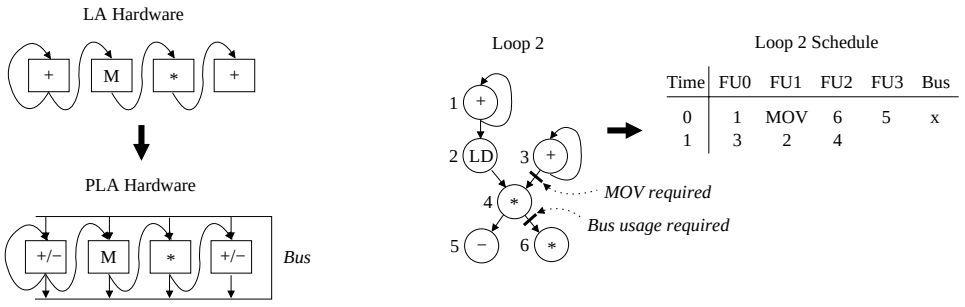


**Figure 6.5:** PLA generalization and scheduling example.

Figure 6.5 shows the results of the datapath generalization so far: FUs have been generalized, MOVs are supported, and a global bus has been added. Loop 2 is now able to execute on the LA originally designed for Loop 1, using the II=2 schedule shown. The remainder of this section discusses additional datapath restrictions that are not shown by this simple example.

#### 6.3.2.3 Shift Register Files

A limiting aspect of the single-function hardware is the nature of the SRFs – because they have a fixed number of entries, any value produced by the corresponding FU must be consumed within a certain number of cycles, or it will "fall off" the end of the SRF. In addition, specific SRF entries are connected to consuming FUs, so the values can only be read at certain times. Both of these issues can be addressed by replacing SRFs with rotating register files (RRFs) [12]. RRFs are similar to standard register files with the modification that the physical register address is a function of the input address and a base register which is decremented once per iteration. RRFs are well suited for modulo scheduled loops because this renaming mechanism overcomes cross-iteration register overwrites.

The replacement of SRFs by RRFs introduces some additional hardware, namely base registers, adders, and decoders for the read and write ports. In addition, the sizes of the RRFs are rounded up to the next power of two to facilitate efficient implementation of register rotation. However, the RRFs remain small (thus the width of base registers and adders is only a few bits per register file) and distributed.

An additional cost of replacing SRFs with RRFs is in the control path: each read and write port now requires an address, whereas the hardwired SRFs required no addressing at all. In addition, the valid bits that were associated with SRFs are no longer required for RRFs; thus, one less bit is required per register. However, multiple-producer single-consumer relationships must now be handled through the use of SELECT operations in software.

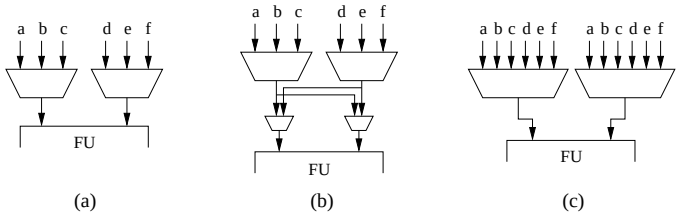


**Figure 6.6:** Generalizing port-specific connections: (a) baseline, (b) allowing swaps, (c) generalized.

#### 6.3.2.4 Port-specific Connectivity

In the single-function LA, each input port of an FU has its own connections to specific register files. To schedule another operation onto that FU, both of the operation's source operands must be routable from where they are produced to the corresponding input ports of the FU. Scheduling can fail if either routing is not possible. If the operation is commutative, then swapping the sources of the operation may result in a successful schedule; however, to relax this constraint more generally, the actual physical connectivity within the datapath should be increased. Figure 6.6 illustrates two methods for accomplishing this. The first is to introduce an additional level of MUXing at the FU input ports such that the ports can swap values, as shown in Figure 6.6(b). In the figure, this allows port 1 to read value $d$ and port 2 to read value $a$, for example. However, modeling this two-level MUX is challenging for the compiler, as it must ensure during scheduling that invalid combinations (e.g. port 1 reading $d$ and port 2 reading $e$) do not occur. Thus, a more general strategy is to widen the input MUXes to allow each input port to read its operand from any connection originally made to either port, as shown in Figure 6.6(c).

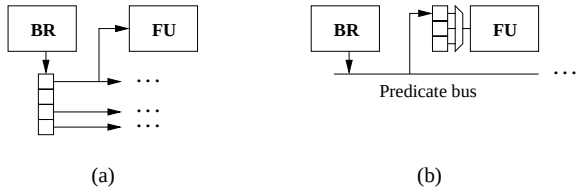


**Figure 6.7:** Generalizing staging predicate: (a) direct hardwired connections, (b) generalized.

#### 6.3.2.5 Staging Predicate

The LA is a hardware implementation of a modulo scheduled loop; as such, operations in the loop kernel are scheduled in various stages, and must be controlled by guarding predicates as the software pipeline fills and drains. This guarding predicate is produced by the branch unit and consumed by all other FUs. In the single-function hardware, specific connections are made between registers in the branch unit's output SRF and the other FUs. This effectively restricts the stage in which operations on a given FU may be scheduled. To generalize this aspect of the hardware, staging predicates are broadcast over a bus to all FUs, significantly increasing scheduling flexibility. The additional cost is low because each predicate is a single bit, and the number of predicates required is just the number of stages in the schedule.

#### 6.3.2.6 Hardwired Control

In the single-function LA, the datapath is directed by hardwired control signals generated by a finite state machine. To allow programmability, the datapath should instead be directed by signals from a control memory. The size of the control memory depends on the number of FUs, MUXes, and registers in the design as well as on the

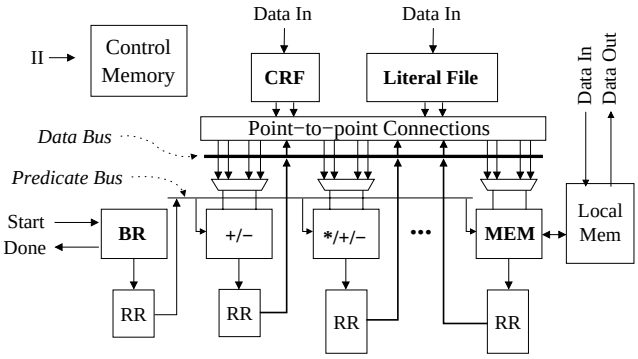


**Figure 6.8:** Template for programmable loop accelerator.

maximum allowed II. In addition, in the single-function LA, literal operands are hardwired. Clearly, this does not allow a loop with different literals to be mapped to the hardware. By placing literals into a central literal file, different literal values may be used for different loops.

#### 6.3.2.7 PLA Architecture

Figure 6.8 shows the template for the PLA, generalized from the datapath shown in Figure 3.2. The accelerator is designed for a specific loop at a specific throughput, but contains a more general datapath than the single-function LA to allow different loops to be mapped onto the hardware. FUs have been generalized to support more functionality; a low-bandwidth bus connects all FUs; the staging predicate is broadcast over a bus; shift register files are replaced with small, distributed RRFs; and the FU input MUXes are widened. The area and power overheads of these changes will be discussed in Section 5.3.

The augmented design flow for PLAs is shown in Figure 6.9. During the creation of the hardware, the datapath is customized for a given loop (labeled Loop 1) but

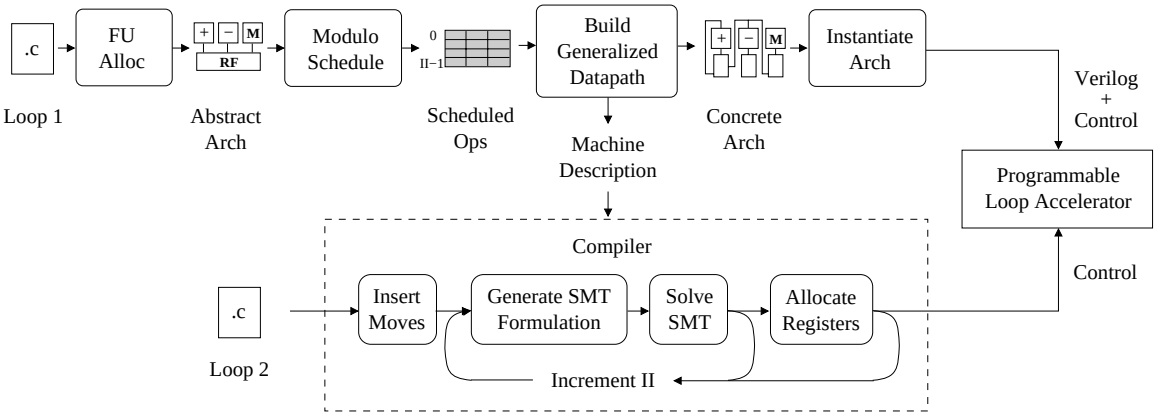


**Figure 6.9:** Design and compilation flow for programmable loop accelerator.

is also generalized using the techniques described above. Additional control logic is generated to support the programmable features of the LA. A scheduler-oriented description of the hardware is then generated, containing both information about the datapath as well as the control signals required to direct the datapath. This machine description can then be used by the compiler (shown by the dotted box and described in the next section) to map a new loop onto the same hardware.

## 6.4 Constraint-driven Scheduling

### 6.4.1 Scheduling Overview

The objectives of scheduling a loop onto an existing accelerator are significantly different from those of scheduling to design the accelerator. When designing the accelerator, the scheduler targets an abstract, fully-connected VLIW machine, and attempts to minimize the final cost of the accelerator at a given II. However, when

targeting the existing accelerator, the cost is fixed and the goal is to map the loop onto the hardware with the lowest II possible.

Conventional modulo schedulers assume a machine with a datapath that is largely homogeneous. For example, FUs are typically ALUs capable of all integer operations, and a centralized register file allows data transfers from any producer FU to any consumer FU. Multicluster VLIWs and CGRAs have more distributed resources, but these architectures are still regular. Conversely, the loop accelerator datapath contains a significant amount of heterogeneity. FUs have a subset of functionality that is tailored for the loop being accelerated, and connections between FUs are point-to-point and highly irregular. A scheduler targeting an accelerator must accommodate this heterogeneity. In terms of FU functionality, the scheduler must restrict the valid resource assignments of each operation to those FUs that are compatible with the operation. In terms of limited connectivity, if an operation produces a value on some FU and this value cannot be directly accessed by the FU where the consumer is scheduled, then either MOV operations must be scheduled to route the value through other FUs, or a global bus must be used to transfer the value.

The proposed constraint-driven modulo scheduler maps a new loop onto an existing PLA by first inserting any potentially required MOVs into the loop's dataflow graph, and then formulating the assignment of operations to FUs and time slots as a satisfiability problem as described in the next subsection. As in conventional modulo scheduling, allocation of rotating registers is performed after assignment of operations to FUs and time slots. If the loop cannot be scheduled at a given II, or if rotating

register allocation fails, the II is increased and another scheduling attempt is made. The dotted box in Figure 6.9 shows the compiler flow.

### 6.4.2 SMT-based Scheduling

The scheduling problem is formulated as a Satisfiability Modulo Theory (SMT) problem. SMT is a general form of satisfiability (SAT) that allows the use of predicates over non-binary variables (for example, integers) in addition to conventional boolean expressions. The problem input is a dataflow graph, a desired II, and a PLA; the output is a modulo schedule where each operation in the dataflow graph has been assigned an FU and a time slot, if such a schedule is feasible.

The body of the loop being scheduled is represented as a dataflow graph $G = (V, E)$, where $V$ represents the set of operations in the loop and $E$ represents the data dependence edges between operations. Each edge has an associated latency $l_{i,j}$ that specifies the latency of the producer operation $i$, and a distance $d_{i,j}$ that specifies the iteration distance between when the value is produced by operation $i$ and consumed by operation $j$.

The schedule for the loop is represented by the $|V| \times |F| \times II$ boolean variables $X_{i,f,t}$, where $F$ is the set of FUs in the machine and $II$ is the initiation interval. Thus, operation $i \in V$ is scheduled on FU $f \in F$ in time slot $t \in \{0, II-1\}$ if $X_{i,f,t}$ is true. Variables representing the assignment of operations to incompatible FUs are omitted from the formulation. In addition, a set of $|V|$ integer variables $S_i$ represent the stage assignment for each operation $i$ in the modulo schedule.

To ensure that each operation is assigned to exactly one FU and time slot, the following constraints are asserted:

$$\bigvee_{\forall f \in F} \bigvee_{t=0}^{II-1} X_{i,f,t} = true \qquad \forall i \in V \tag{6.1}$$

$$X_{i,f_1,t_1} \wedge X_{i,f_2,t_2} = false \qquad \forall i \in V, f_1 \neq f_2, t_1 \neq t_2 \tag{6.2}$$

Next, to ensure that each FU has at most one operation assigned to it in each time slot, the following set of constraints are asserted:

$$X_{i_1,f,t} \wedge X_{i_2,f,t} = false \qquad \forall f \in F, t \in \{0, II-1\}, i_1 \neq i_2 \tag{6.3}$$

It is assumed that any multi-cycle FUs are fully pipelined and able to begin executing a new operation each cycle.

Next, constraints must be asserted to ensure that no data dependence violations occur. In other words, given producer operation $i$ and consumer operation $j$, the unrolled schedule time of $j$ must be at least $l_{i,j} - (d_{i,j} \times II)$ cycles after that of $i$. In other words:

$$ust(j) \geq ust(i) + l_{i,j} - (d_{i,j} \times II)$$

where $ust(i)$ is the unrolled schedule time of $i$. Since $ust(i)$ is a function of both the stage $S_i$ and the time slot $t_i$, this can be expressed as:

$$(S_j \times II) + t_j \geq (S_i \times II) + t_i + l_{i,j} - (d_{i,j} \times II)$$

In the SMT formulation, $t$ is not a true variable; rather, it is a constant with respect to some boolean variable $X_{i,f,t}$. Thus, the above can be expressed in terms of variables $X$ and $S$, and constants $t$, $l$, $d$, and $II$:

$$\neg X_{i,f_i,t_i} \vee \neg X_{j,f_j,t_j} \vee (S_j \times II) + t_j \geq (S_i \times II) + t_i + l_{i,j} - (d_{i,j} \times II) \tag{6.4}$$

Constraint (6.4) is asserted for all values of $t_i$ and $t_j$ between 0 and $II - 1$, and for all FUs $f_i$ and $f_j$ compatible with operations $i$ and $j$, respectively. This set of constraints is repeated for all pairs of operations that have data dependence edges between them.

Note that all of the above constraints merely ensure that a valid schedule can be achieved given a fully connected architecture; none of the constraints presented thus far consider the limited connectivity of the loop accelerator datapath. Not all FUs are able to communicate directly with each other; thus, the satisfaction of constraints (6.4) may still result in an invalid schedule. To resolve this, the constraints should be modified slightly. When the producer FU $f_i$ and the consumer FU $f_j$ are directly connected (there is a wire from the register file at the output of $f_i$ to the input of $f_j$), constraints (6.4) may be asserted as before. However, when there is no such connection, the following constraints are asserted instead, which prohibit operations $i$ and $j$ from being scheduled on $f_i$ and $f_j$:

$$\neg \left( \bigvee_{t_i=0}^{II-1} X_{i,f_i,t_i} \right) \vee \neg \left( \bigvee_{t_j=0}^{II-1} X_{j,f_j,t_j} \right) \tag{6.5}$$

Another feature of the PLA is the presence of a low-bandwidth global bus for transferring values between any pair of FUs. The bus is modeled as a counted resource, with a limited number of transfers available per clock cycle. In the SMT formulation, additional boolean variables $B_{i,t}$ are introduced, representing the use of a global bus resource by operation $i$ in time slot $t$. When a producer FU and a consumer FU are directly connected, the bus is not needed because the value can be transferred through the standard point-to-point connections. However, when two FUs $f_i$ and $f_j$ are not directly connected, constraint (6.5) may be modified to allow use of the global bus:

$$\neg \left( \bigvee_{t_i=0}^{II-1} X_{i,f_i,t_i} \right) \vee \left( \bigwedge_{t_j=0}^{II-1} \neg X_{j,f_j,t_j} \vee B_{j,t_j} \right) \tag{6.6}$$

It then remains to limit the number of global bus users in each cycle:

$$B_{i_1,t} \wedge B_{i_2,t} = false \qquad \forall t \in \{0, II-1\}, i_1 \neq i_2 \tag{6.7}$$

The above assumes that one global bus resource is available per cycle. To model more than one bus resource, either additional boolean variables should be introduced to represent each additional resource, or the boolean variables may be replaced by integer variables whose sum is constrained to be less than or equal to the number of bus transfers available per cycle.

Solving for boolean variables $X_{i,f,t}$ and $B_{i,t}$ and integer variables $S_i$ under the constraints given by Equations (6.1) through (6.7) gives a legal modulo schedule with initiation interval $II$ for the graph $G$ on a given PLA datapath.

## 6.5 Graph Perturbation

A goal of this work is to quantify the similarity required between two loops in order for one loop to be mappable onto a PLA designed for another loop. Towards this end, it is useful to systematically generate a series of loops with varying degrees of similarity. We propose a graph perturbation method that takes an existing dataflow graph for a loop and introduces small changes, producing new loops that are increasingly different from the original loop.

In a dataflow graph, changes to nodes and edges represent modifications to the original source code of the loop. For example, a new node can represent a new C statement; changing an edge can represent changing the operands of a statement. Most operations in the loop have two source operands; therefore, when a node in the dataflow graph has fewer than two incoming edges, one or more of these source operands are either live-in (defined by operations outside of the loop) or literal values. Thus, when perturbing the graph, adding or removing an incoming edge of a node corresponds to changing a live-in or literal operand to a register operand or vice versa. During the graph perturbation, it is assumed that nodes in the graph can have up to two incoming edges (excluding the guarding predicate input, which exists for all operations), although in reality, exceptions exist for operations, such as store-with-displacement and operations with multiply-defined source operands.

In the graph perturbation module, four basic transformations are used:

- **Adding an edge between existing nodes.** A random node is selected as the producer node; a random node with fewer than two incoming edges is selected

as the consumer node. A new edge is inserted from producer node to consumer node. The latency of the edge is set to the latency of the producing operation. The iteration distance of the edge is set depending on the order of producer and consumer in a topological sort of the graph: if the producer appears later than the consumer, then the distance is set to 1. Otherwise, it is set to 0.

- **Adding an edge with a new producer.** A random node with fewer than two incoming edges is selected as the consumer; a new node with a random opcode is generated to create a new producer node. A new edge is inserted from producer to consumer with the latency of the producing operation and a distance of 0.

- **Adding an edge with a new consumer.** A random node is selected as a producer, and a new node with a random opcode is generated to create a new consumer node. A new edge is inserted from producer to consumer with the latency of the producing operation and a distance of 0.

- **Removing an edge.** A random edge is deleted from the graph. Edges originating from producers with only one consumer are excluded, as removing such edges would render the producing operation useless. On the other hand, removing an edge from a consumer with only one producer is permitted, as this corresponds to replacing the operation's register operand with a literal or live-in operand.

The graph perturbation process is iterative. Beginning with the original graph,

a random transformation is chosen from among the four in the above list. Then, random nodes or edges are selected as needed depending on the transformation, and the transformation is applied. This process is repeated as many times as desired. With each iteration, the graph becomes successively more dissimilar from the original graph. As nodes and edges are perturbed, the communication patterns within the graph change and it becomes less likely that the graph can be mapped onto hardware designed for the original loop.

Figure 6.10(a) shows the dataflow graph for `heat`, a loop kernel from a scientific application that models heat diffusion. After 5 perturbations, the graph is as shown in Figure 6.10(b). Four new edges (and two new nodes) have been added, and one edge (from node 13 to itself) has been removed. At this point, the graph still resembles the original. In Figure 6.10(c), 10 perturbations have been performed in total, most of which happen to be new edges. By this point, the graph looks fairly different from the original, yet in this case it is still possible to map it onto the PLA designed for the original loop.

One limiting factor in mapping a loop to an accelerator is the functionality of the FUs. If the loop contains an operation that is not supported by any FUs in the hardware, mapping is guaranteed to fail. Graphs produced by introducing such operations to the original graph will fail trivially; thus, we disallow such perturbations in this study to preserve functional compatibility. Note that, as mentioned in Section 6.3.2, the PLA datapath already contains FUs that have been generalized to some degree; thus, it is possible for loops containing operations that do not exist in the original

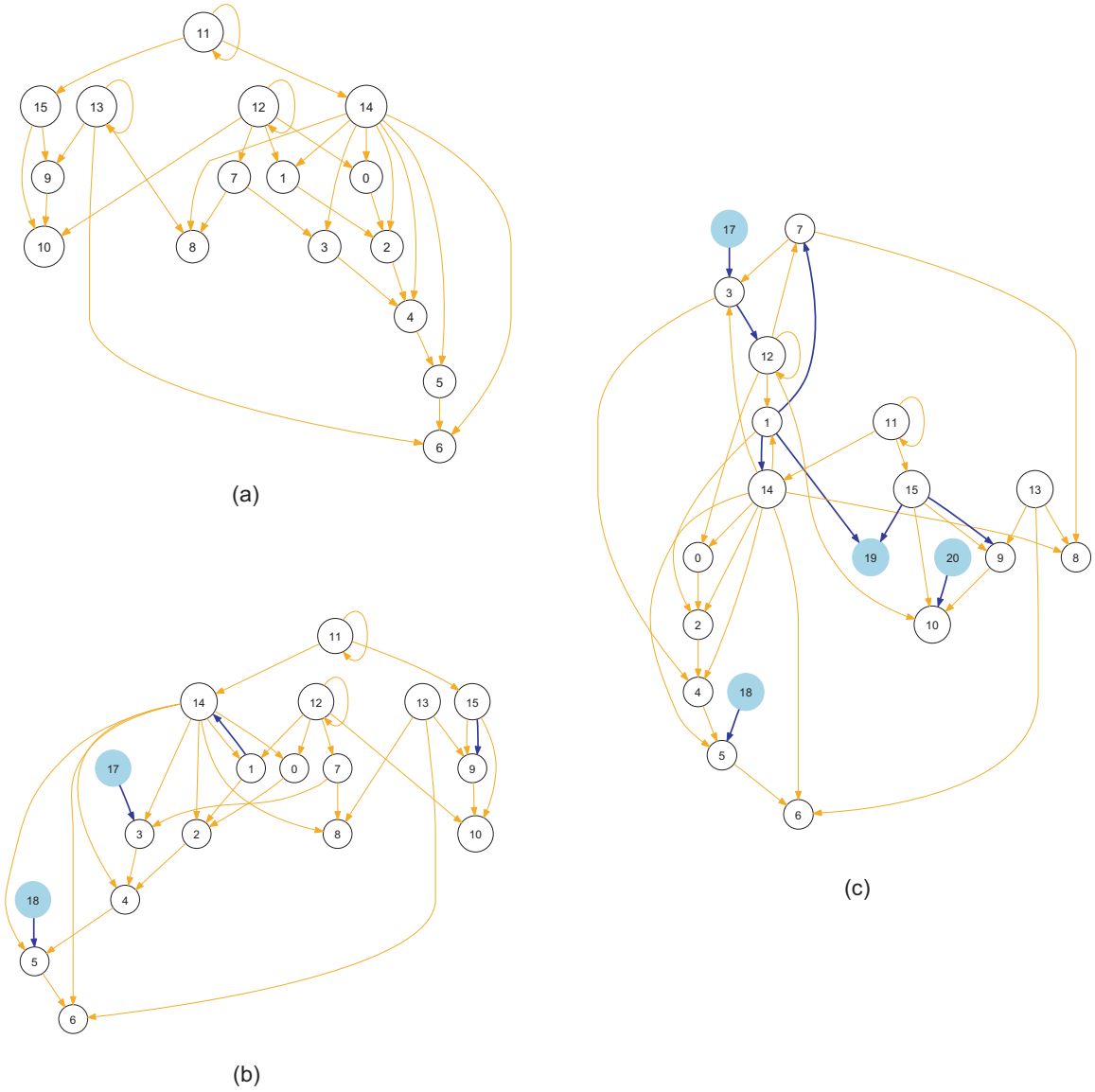


**Figure 6.10:** Graph perturbation example from the `heat` benchmark: (a) original loop, (b) after 5 perturbations, (c) after 10 perturbations.

loop to be successfully mapped.

## 6.6 Experimental Results

### 6.6.1 Overview

Loop kernels from various DSP (`fir`, `fft`, `fmradio`, `bfform`), media (`dcac`, `dequant`, `fsed`, `sobel`), and linear algebra (`heat`, `lu`) applications are used to evaluate the efficiency and programmability of the PLA architecture. The loops range in size from 17 operations up to 60 operations. For each loop, the synthesis system is used to generate Verilog corresponding to both single-function and programmable LAs. Synthesis and placement are performed on the Verilog using Synopsys Design Compiler and Physical Compiler and a 0.13$\mu$ standard cell library. Power analysis is performed using PrimeTime PX after the design has been back-annotated with information about parasitics and switching activity. Three experiments are shown: first, the PLA is compared with single-function LAs as well as with the OR-1200 RISC processor [51], which is a simple, single-issue core with a 5-stage in-order pipeline. This experiment examines the tradeoffs in power efficiency when moving from single-function to semi-programmable to fully programmable hardware. The second experiment shows the costs of the various PLA datapath generalizations described in Section 6.3.2. The third experiment measures the programmability of the PLA by attempting to map different loops onto an accelerator.

### 6.6.2 Area and Power Comparison

In the first experiment, the LAs are compared with the OR-1200 processor, which is synthesized in the same technology ($0.13\mu$) as the accelerators. The loops are compiled for the processor using a version of the GNU compiler toolchain which has been ported to the OR-1200; optimization level -O2 is used. PrimeTime PX is used to measure the power consumption of the processor given switching activity information obtained during loop execution. Both the local memories in the loop accelerators and the caches in the OR-1200 are included in the power measurements. Figure 6.11 shows the relative power consumption of the single-function LA, the PLA, and the OR-1200 for each loop, on a logarithmic scale. The power consumption of the single-function LA is 1.0; for each loop, the first bar shows the power consumption of the PLA, and the second bar shows that of the OR-1200. In addition, there is a third bar for each benchmark, representing the amount of power the OR-1200 would consume if it ran at a frequency yielding the same performance as the corresponding LA.[2] It is important to note that though the power consumption of the PLAs and the OR-1200 is comparable, the PLAs are 6x to 33x faster than the processor, and this difference in power efficiency is reflected in the performance-equivalent bar.

As the graph shows, the PLAs consume about 2x to 9x more power than the corresponding single-function LAs (which have the same performance). This increased power consumption is due to several factors. First, the power consumed by the RRFs makes up a significant fraction of the overall PLA datapath. When the SRFs in the

[2]Note that no voltage scaling is done, so the power consumption of the OR-1200 is an underestimate.

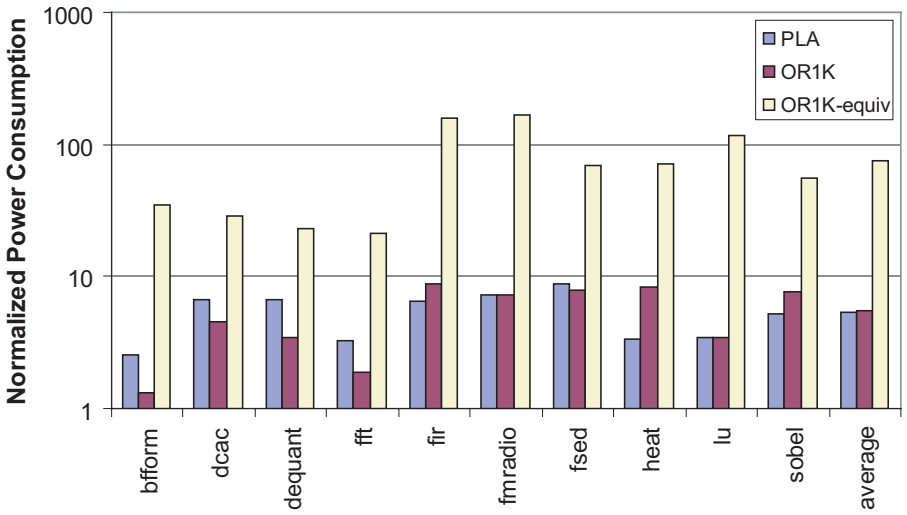


**Figure 6.11:** Power consumption of PLA and OR-1200 relative to single-function LA.

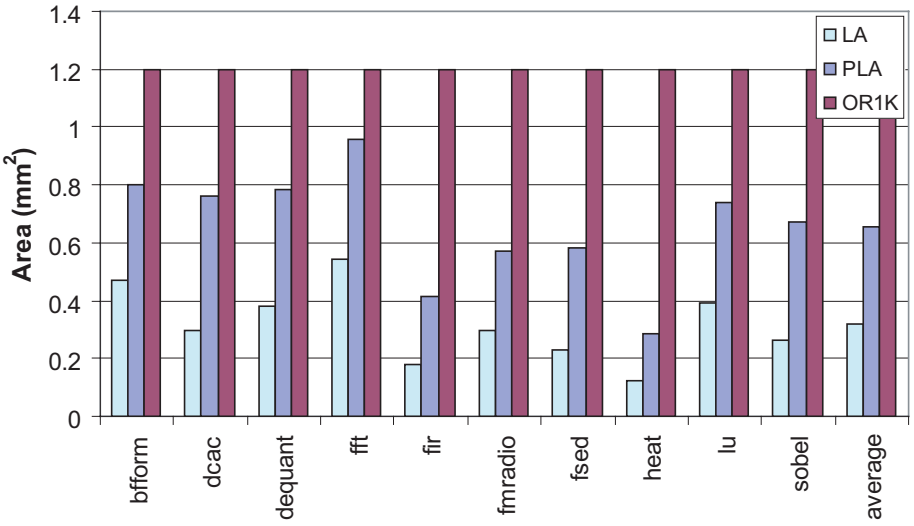


**Figure 6.12:** Area of loop accelerators and OR-1200.

single-function LA are replaced with RRFs, their sizes must be increased to the next power of two, and additional logic must be added in the form of decoders, adders, and base registers. Also, in the current implementation, the RRFs are synthesized from behavioral descriptions rather than being created by a RF generator, thus missing out on typical RF area and power optimizations such as master latch sharing. The PLA also has other datapath generalizations as described in Section 6.3.2, such as wider MUXes, which consume additional power. Finally, since the PLA datapath is more complex than the single-function LA, when synthesizing both LAs with the same target clock frequency, the gates in the PLA will be sized larger to meet timing constraints, thus consuming more power.

Comparing the PLA with the OR-1200 at the same performance level, the OR-1200 consumes from 4x to 34x more power. Since the OR-1200 performs general instruction-based execution, it suffers increased power consumption due to factors such as instruction fetch and decode, a centralized register file, caches, and the data forwarding network. Conversely, the PLA is a customized architecture with distributed datapath elements and local memories, and thus is able to achieve high throughput with significantly less power.

Figure 6.12 shows a comparison of the areas of the single-function LA, PLA, and OR-1200. The generalized datapath of the PLA causes its area to increase roughly 2x compared to the single-function LA. Overall, all three hardware implementation styles take up relatively little area, with single-function LAs averaging $0.3mm^2$, PLAs averaging $0.65mm^2$, and the OR-1200 occupying $1.2mm^2$. In terms of area efficiency

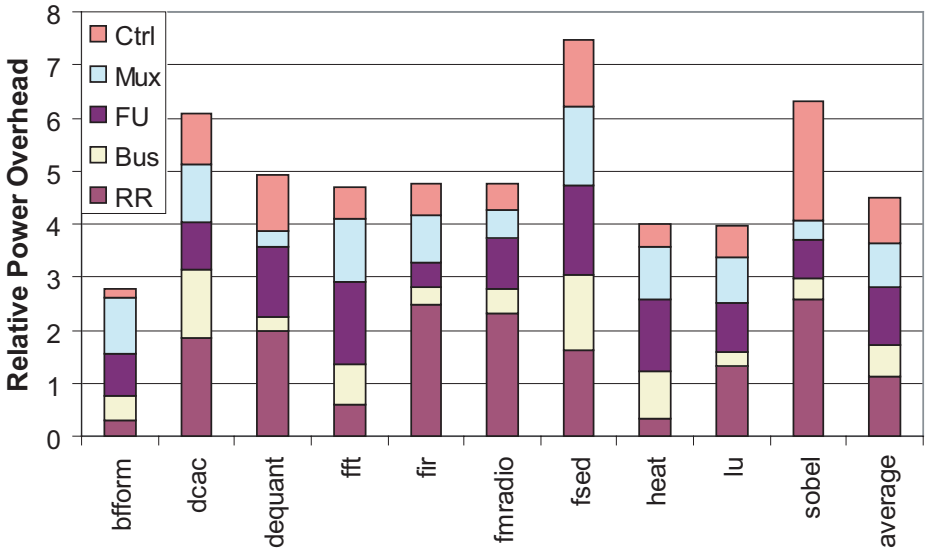


**Figure 6.13:** Power consumption breakdown of PLA generalizations.

(performance per area), the PLA is roughly 30x more efficient than the OR-1200 on average for these loops.

### 6.6.3 Datapath Generalizations

Figure 6.13 shows the power overheads of the major datapath generalizations in the PLA. For each loop, a stacked bar shows the breakdown of the amount of power consumption contributed by each datapath element. The power contribution of some datapath elements (such as the global bus) are difficult to isolate when looking at the overall PLA hardware; to measure these contributions, an LA was created which had (for example) a global bus and no other generalizations, and the overall power consumption was compared with that of the original single-function LA. In general, the datapath components contributing the highest amount of overhead are the RRFs

and the FU generalizations.

### 6.6.4 Programmability

For each loop, a PLA is synthesized using our system; Table 6.1 shows the characteristics of each loop and its corresponding PLA. The Yices SMT solver [11] is then used to modulo schedule loops onto PLAs using the formulation described in Section 6.4.2. Two types of experiments are presented: first, we perturb the loops and attempt to map them onto the PLAs designed for the corresponding unperturbed loops. These experiments study the relationship between loop similarity and mappability, and represent reuse of existing hardware after source modifications are made to a loop. Next, we attempt to map (unperturbed) loops onto PLAs designed for other loops. This cross-compilation experiment examines the ability to reuse existing hardware for different loops with similar computation structure.

For the perturbation study, we run a set of experiments wherein each loop is randomly perturbed a number of times as described in Section 6.5. For each number of perturbations, the SMT scheduler is used to map the perturbed loop onto the PLA. Initially, the perturbed loop is scheduled at the same II as the original loop; if this fails, the II is incremented until the scheduler succeeds or a threshold is reached. The less the II needs to be increased, the more easily the hardware can be reused. Note that typically, the II can continue to be increased until there is sufficient scheduling flexibility to route all data transfers and the scheduler succeeds. Conversely, it is generally not possible for a perturbed loop to be scheduled at a lower II than the

| Loop | #Ops | RecMII | Base II | #FUs |
|---|---|---|---|---|
| dcac | 44 | 2 | 4 | 13 |
| dequant | 63 | 3 | 8 | 12 |
| fft | 54 | 1 | 7 | 13 |
| fir | 26 | 1 | 2 | 13 |
| fmradio | 18 | 1 | 4 | 6 |
| fsed | 40 | 1 | 4 | 11 |
| heat | 17 | 6 | 6 | 5 |
| lu | 41 | 9 | 9 | 9 |
| sobel | 49 | 1 | 4 | 16 |
| turbo | 17 | 1 | 4 | 6 |

**Table 6.1:** Loop kernels from DSP and media applications.

original loop, because after each perturbation, the number of operations either increases or remains the same. Thus the resources (which were allocated to support the throughput of the original II) are insufficient to support a higher throughput.

Figure 6.14 shows the results of the perturbation study. The y-axis shows the number of perturbations from the original loop. The bar for each benchmark is segmented to indicate the amount that the II needed to be increased in order to achieve a successful schedule. Multiple runs are performed, perturbing the graph using different random seeds, and the II increases are averaged across these runs. The performance decrease that the II increase corresponds to is dependent on the original II shown in Table 6.1 under the "Base II" column.

As the graph shows, the programmability of the PLA depends on the original loop. Factors such as more opcodes and more heterogeneous communication patterns in a loop will lead to more programmable hardware. For example, `fir` is a small loop which has simple, repeated communication patterns. Thus, there are fewer unique point-to-point connections in the datapath. On the other hand, `heat` is also a small

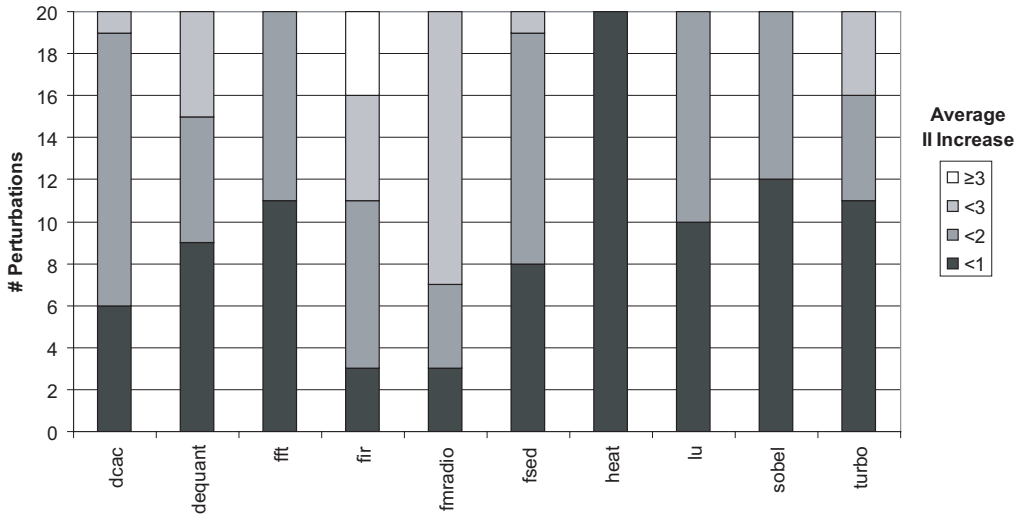


**Figure 6.14:** II increase necessary to schedule loops with perturbations.

loop, but its PLA contains more heterogeneous connections.

Figure 6.15 looks at three of the benchmarks in more detail. Each graph represents one loop kernel, with the number of perturbations shown on the x-axis. The two lines represent the relative II increase required to schedule the perturbed loop (averaged across multiple runs with different random seeds) as well as a measure of how similar the perturbed loop is to the original. The similarity metric is based on degree distribution [48], which is a histogram describing how many operations in the dataflow graph have a given degree (number of connections with other operations). We make the modification that nodes are differentiated by class of operation (arithmetic vs. memory) when calculating the distribution. The degree distributions of two dataflow graphs are then normalized to range from 0 to 1 and compared using the sum of absolute differences. Thus, the value can range from 0 (very similar) to 2 (very different). As can be seen in the graphs, the II increase generally tracks the

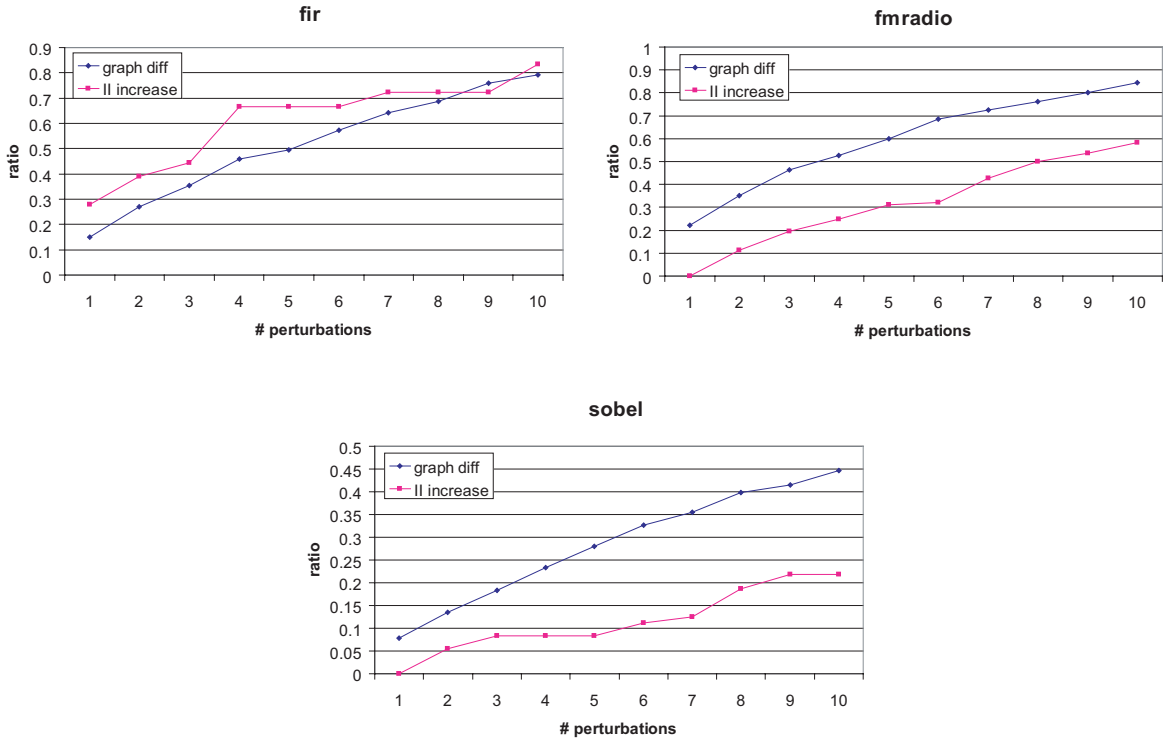


**Figure 6.15:** Relative II increase and graph difference vs. perturbations for `fir`, `fmradio`, and `sobel`.

increase in difference between perturbed loops and original loops. In several cases, the II "levels off" before increasing again; this happens when increasing the II gives enough scheduling flexibility that multiple additional perturbations can be scheduled without further II increases. Also, notice that in the larger loop (`sobel`), the graph of II increase is flatter, as each II increase corresponds to more scheduling slots becoming available.

In order to study the programmability effects of the architecture generalizations described in Section 6.3.2, the same perturbation study is run with more restrictive PLA hardware. Figure 6.16 shows the results of scheduling the `fir` benchmark to

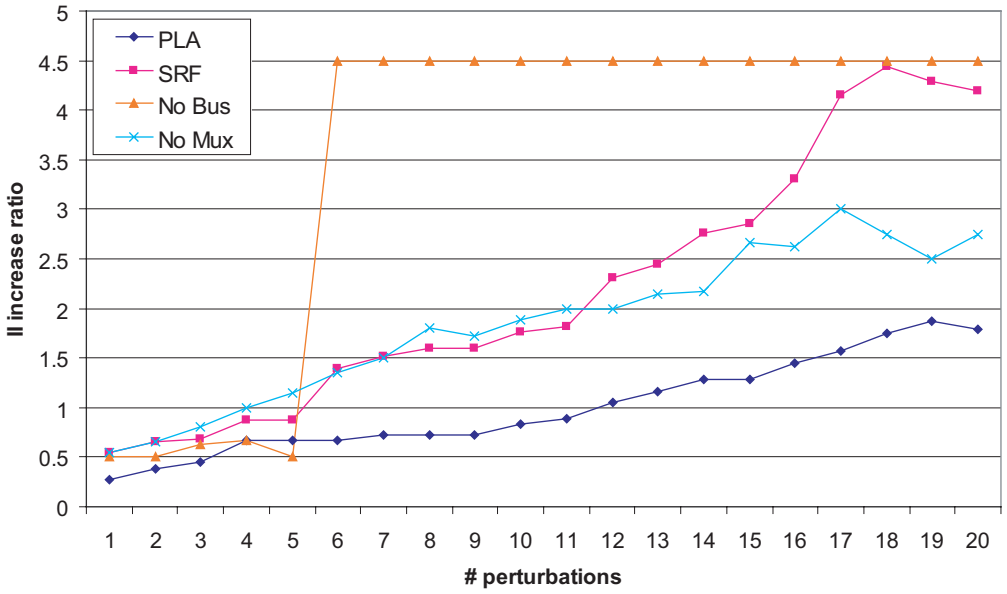


**Figure 6.16:** Perturbation studies with more restrictive PLAs.

various more restrictive hardware configurations. The "SRF" configuration replaces the rotating register files with SRFs; it is assumed that any entry can be read out of the SRF, but entries may "fall off" the end, so consumers are forced to be scheduled closer in time to producers. The "No Bus" configuration contains no global bus. This limits connectivity significantly because the remaining interconnect is highly customized to the original loop. In the "No Mux" configuration, the MUX inputs are not allowed to be swapped; thus, connections are port-specific and more restrictive. In each configuration, the reduced flexibility in the datapath means that higher IIs are required to map perturbed loops onto PLAs. In the case of "No Bus", mapping failed outright beyond 6 perturbations due to insufficient interconnect; thus, the maximum II increase that was attempted (4.5x) is shown for greater than 6 perturbations. It can be seen that with all of the hardware generalizations in place ("PLA" line), the

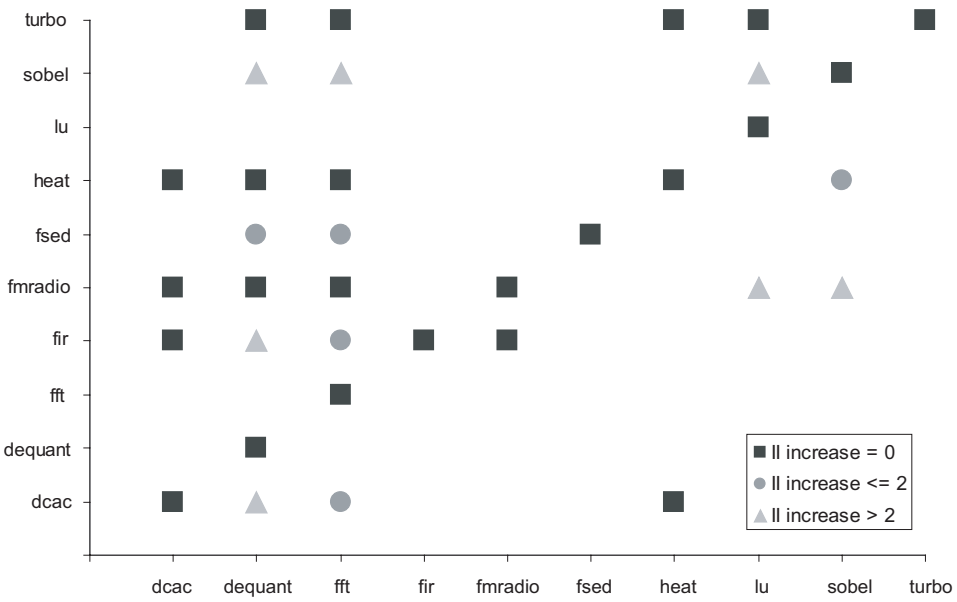


**Figure 6.17:** Cross compilation results. PLAs are designed for loops along the x-axis at II values listed in Table 6.1. Loops along the y-axis are then mapped onto them.

achievable II is significantly lower as the number of perturbations increases.

Figure 6.17 shows the results from the cross-compilation study. PLAs are designed for the loops listed across the x-axis, and the loops listed on the y-axis are mapped onto them. The presence of a symbol indicates that the loop was successfully mapped onto the hardware. A dark square indicates that the mapping was accomplished with no II increase over the ResMII; as expected, dark squares appear on the diagonal where loops are mapped onto their own hardware. Other symbols represent successful mapping with some II increase. The lack of a symbol at a particular coordinate indicates that mapping failed for that combination of loop and hardware; typically this occurred because of incompatible functionality (that is, the loop contained an operation that could not be executed on any FU).

| | dcac | deq | fft | fir | fmr | fsed | heat | lu | sob |
|---|---|---|---|---|---|---|---|---|---|
| **turbo** | 1.41 | 1.27 | 1.50 | 1.76 | 1.67 | 1.65 | 1.53 | 1.35 | 1.44 |
| **sobel** | 1.26 | 1.09 | 0.67 | 1.10 | 0.74 | 0.76 | 1.25 | 1.03 | |
| **lu** | 1.06 | 0.95 | 1.30 | 1.63 | 1.26 | 1.42 | 1.35 | | |
| **heat** | 0.90 | 1.20 | 0.97 | 0.97 | 0.98 | 1.23 | | | |
| **fsed** | 1.21 | 1.08 | 0.64 | 0.92 | 0.80 | | | | |
| **fmradio** | 1.24 | 1.13 | 0.82 | 0.81 | | | | | |
| **fir** | 1.24 | 1.41 | 0.96 | | | | | | |
| **fft** | 1.25 | 0.97 | | | | | | | |
| **dequant** | 0.89 | | | | | | | | |

**Table 6.2:** Similarity of loop kernels; a lower number means the two loops are more similar to each other.

The success of cross-compilation primarily depends on two factors, loop size and loop similarity. With respect to loop size, it is easier for smaller loops to map onto larger hardware, as more scheduling flexibility is available. Note that two columns, those of `dequant` and `fft`, are heavily populated, indicating that most other loops were able to successfully map to these PLAs. These are the two largest loops, and the resulting PLAs have more functionality and interconnectivity as a result. Similarly, rows corresponding to smaller loops are well-populated. With respect to loop similarity, loops are often able to map onto the hardware of other similar loops. Table 6.2 shows the degree-based similarity metric described earlier in this section for the loops in this cross-compilation study. The `dcac` loop is most similar to `heat` and `dequant`, and is successfully mapped onto hardware designed for these other two loops even though the `heat` loop is significantly smaller. However, in general the loop similarity is not an ideal predictor of schedulability, as similarity is an estimated aggregate measure that does not account for the specific resource usage requirements of the

loops.

The runtime of the SMT scheduler ranged from a few seconds up to half an hour, depending on the size of the loop (the largest loop had 63 operations).

## 6.7 Accelerator Efficiency Analysis

Three general classes of loop accelerators have been presented in this dissertation: single-function LAs, multifunction LAs, and programmable LAs. Figure 6.18 plots the performance vs. power consumption of these LAs as well as the OR-1200. On this plot, points on the same slope have equivalent power efficiency in terms of MIPS/mW, with points towards the upper left having greater power efficiency. For each type of hardware, the average efficiency is plotted as a line; for the designs studied, the single-function LAs achieve 105 MIPS/mW, multifunction LAs achieve 36 MIPS/mW, PLAs achieve 24 MIPS/mW, and the OR-1200 achieves 2 MIPS/mW on average.

As can be seen from the plot, the loop accelerators are able to achieve order-of-magnitude improvements in efficiency over the OR-1200 via customization. The PLAs allow hardware reuse in the presence of source code changes, giving up some efficiency to the non-programmable LAs but maintaining large efficiency gains over general purpose hardware. Four commercially available hardware implementations are also shown in the plot: the Tensilica Diamond Core [63], a processor with ASIP-style instruction set extensions optimized for embedded designs; the Texas Instruments C6x digital signal processor [65]; the ARM11 embedded general purpose processor [3]; and the Intel Itanium 2 [29], a general purpose processor targeted for enterprise servers.

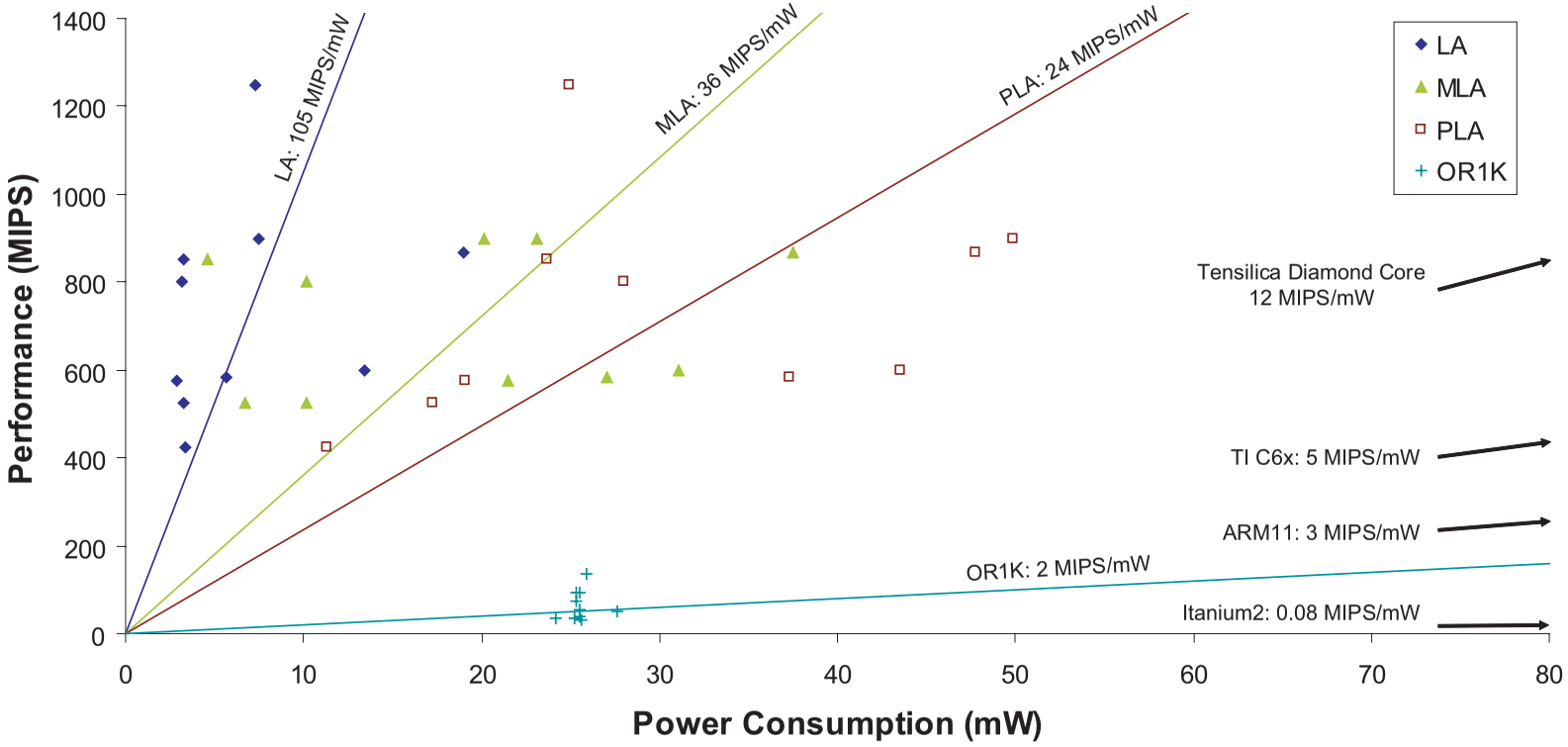


**Figure 6.18:** Performance/power of loop accelerators, OR-1200, and commercial architectures.

As can be observed, the efficiency decreases significantly as the hardware becomes more general and less tailored for embedded applications.

## 6.8 Related Work

Related work in terms of hardware design and reconfigurable datapaths was largely discussed in Section 2.4. Thus, this section will discuss work related to compilation for irregular datapaths. Such prior work can best classified by the target architecture: CGRAs, multicluster VLIWs, and DSPs.

**CGRA scheduling.** Several modulo scheduling techniques for CGRAs have been proposed. [45] proposes a modulo scheduling algorithm for ADRES architecture based on simulated annealing. It begins with a random placement of operations on the FUs of a CGRA, which may not be a valid modulo schedule. Operations are then randomly moved between FUs until a valid schedule is achieved. Modulo graph embedding is a modulo scheduling technique that leverages graph embedding commonly used in graph layout and visualization [53]. The scheduling problem is reduced to drawing a guest graph (the loop body) onto a three dimensional host graph (the CGRA). The three dimensions consist of the 2-D function unit array and the time slots.

Other CGRA scheduling techniques do not focus on software pipelining loops. Lee et al. propose a compilation approach for a generic CGRA [35]. This approach generates pipeline schedules for innermost loop bodies so that iterations can be issued successively. The main focus of their work is to enable memory sharing between operations of different iterations placed on the same processing element. [69] proposes an

acyclic scheduling technique that decouples resource allocation and time assignment for CGRAs. A graph is constructed where nodes are operations and edges are inserted between nodes that have direct data dependences or common consumers. This graph is then partitioned into cliques and resource allocation is performed by assigning operations in each clique to the same resource. Time slots for operations are later assigned in scheduling phase. Last, convergent scheduling is proposed as a generic framework for instruction scheduling on spatial architectures [37]. Their framework comprises a series of acyclic scheduling heuristics that address independent concerns like load balancing, communication minimization, etc.

**Multicluster VLIW scheduling.** A large body of work has been done on compiling acyclic and loop code for clustered VLIWs [1, 7, 18, 30, 50, 52, 59]. The clustered nature of the datapath can either be taken into account in a prepass before scheduling, such as the Bottom-Up Greedy algorithm in the Bulldog compiler [18], or during scheduling, such as the Unified Assign and Schedule algorithm used in the Lego compiler [52]. Multicluster scheduling is generally an easier problem than CGRA scheduling because it does not consider the issue of routing values through a sparse interconnection network.

**DSP compilation.** A common characteristic of DSPs is non-uniform interconnect between multiple function units and function units to register files. Template-based code generation is typically used to map applications onto such datapaths [38, 39, 43]. However, this is generally done in the context of a single-issue architecture, thus there is no significant scheduling component. A related area is scheduling to pro-

cessors with partial register bypass networks [33, 53]. Partial bypass introduces the problem of variable latencies on dataflow edges depending on function units chosen for a producer/consumer pair.

The primary difference that sets our work apart from these techniques is the irregularity of the target architecture. CGRAs and multicluster VLIWs generally have a regular datapath with uniform interconnectivity, though not all connections are direct. These designs are typically not customized to a particular application, but rather are either general-purpose or possibly domain specific. Conversely, the architectures that we investigate are highly customized LAs with several generalizations. Programmability and thus the opportunities for a scheduler are limited to applications that have similar computation structure to that which the original LA was designed. As a result, previous scheduling approaches cannot readily be adapted to PLA architectures.

## 6.9 Summary

Customized loop accelerators are able to provide significant performance and power efficiency gains over general purpose processors. By building semi-programmable accelerators, it is possible to achieve these efficiency gains while allowing hardware to be reused as the software evolves. The loop accelerator datapath is generalized in an efficient way such that loops that are similar to the original loop may be mapped onto the accelerator. Such programmable loop accelerators provide hardware reusability along with order-of-magnitude improvements in power and area efficiency

over simple low power general purpose processors. In addition, a constraint-driven modulo scheduler is presented which maps loops onto the PLA. The programmability of the PLA architecture and effectiveness of the constraint-driven scheduler are evaluated using a graph perturbation method which allows for systematic exploration of the relationship between loop similarity and hardware reusability. For the loops studied, the PLA was able to achieve 4x-34x better power efficiency and about 30x better area efficiency than a general purpose processor, while losing 2x-9x in power and 2x in area to a custom non-programmable LA.

# CHAPTER 7

# Conclusion

## 7.1 Summary

Loop accelerators provide order-of-magnitude gains in computation efficiency over general purpose processors for highly-executed loop kernels. The difficulty with using custom accelerators is that designing customized hardware is expensive and time consuming. Given the speed at which the computing industry advances, time-to-market is of paramount importance, particularly in the quickly evolving embedded domain. Using an automated system to build accelerators from high-level specifications allows designers to create hardware with a significantly shorter time-to-market. Although design quality of automatically generated hardware may not be as high as full custom hand-designed hardware, it can actually be significantly better when taking into account market advancement during the long time period required for manual design.

In this dissertation, a synthesis system was presented that designs application-centric architectures. The system allows different types of efficient loop accelerators

to be automatically generated from C code. The high level of computation efficiency is achieved through the use of a hardware template that is customized for each loop, using techniques to exploit fine-grained and coarse-grained hardware sharing.

The key component of the application-centric architecture synthesis system is the modulo scheduler, which exposes a high degree of instruction-level parallelism by overlapping iterations of the loop. The scheduler attempts to exploit fine-grained hardware sharing to reduce the cost and increase the efficiency of the accelerator. The accelerator datapath is then derived from the schedule; the final loop accelerator is essentially a hardware implementation of the modulo scheduled loop. The cost sensitive modulo scheduler increases the hardware efficiency (in terms of performance per area) of synthesized accelerators by 20% on average.

By combining the functionality of several accelerators in the form of a multifunction accelerator, significant hardware savings can be achieved. Instead of requiring separate accelerators for each computationally intensive loop, several loops may share a single datapath, exploiting coarse-grained hardware sharing when permitted by the application or applications. By using the datapath union technique, significant hardware savings can be achieved, averaging 43% over the cost of separate accelerators.

A primary downside of building customized hardware is the lack of flexibility: if the software changes, the hardware must be redesigned. By making the hardware semi-programmable, small changes to the software can be accommodated without having to build new hardware. By keeping the amount of programmability small, the hardware can retain its high degree of computational efficiency. Thus, the loop accel-

erator was generalized to support post-programmability, and the synthesis toolchain was augmented to be able to map new loops onto existing hardware. The programmable accelerator achieved up to 34x better power efficiency and 30x better area efficiency than a simple general purpose processor, while losing 2x-9x in power and 2x in area to a non-programmable accelerator.

## 7.2 Future Directions

The research presented here can be extended in several directions. First, there is the issue of integration of accelerators into a complete system. In order for the accelerator to be useful, the performance overheads of transferring control and data to the accelerator must be amortized over its execution. The choice of how accelerators interface with the rest of the system will affect the amount of these overheads, thus impacting the overall performance of the system. Hardware interface questions include how the accelerator is connected to the rest of the system, whether or not a shared memory model is used, etc. Other system integration issues relate to the software side: what support is needed in the host operating system or runtime libraries to efficiently make use of accelerators? The use of language extensions can also be investigated, as they can make it easier for developers to use accelerator hardware.

Another related issue is how to partition work across a heterogeneous system consisting of general purpose processors, loop accelerators, and other types of computation hardware such as domain accelerators and coarse-grained reconfigurable architectures. Different portions of applications should be matched to different computing

substrates depending on the varying requirements for performance, efficiency, and programmability. This can be done statically or dynamically; potentially, a run-time system could be used to dynamically map parts of applications to the available hardware that is best suited for their execution.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] A. Aletà, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Dec. 2001.

[2] E. R. Altman and G. A. Gao. Optimal modulo scheduling through enumeration. *International Journal of Parallel Programming*, 26(3):313–344, 1998.

[3] ARM. Arm11. http://www.arm.com/products/CPUs/families/ARM11Family.html.

[4] S. Bakshi and D. Gajski. Components selection for high performance pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):182–194, June 1996.

[5] K. Bondalapati et al. DEFACTO: A design environment for adaptive computing technology. In *Proc. of the Reconfigurable Architectures Workshop*, pages 570–578, Apr. 1999.

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

[7] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[8] S. Ciricescu et al. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 141–150, 2003.

[9] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, page To appear, June 2008.

[10] H. Corporaal and H. J. Mulder. MOVE: A framework for high-performance processor design. In *Proc. of Supercomputing '91*, pages 692–701, Nov. 1991.

[11] L. de Moura and B. Dutertre. Yices 1.0: An efficient SMT solver. In *The Satisfiability Modulo Theories Competition (SMT-COMP)*, Aug. 2006.

[12] J. Dehnert and R. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1):181–227, May 1993.

[13] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[14] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, pages 338–349, Nov. 1995.

[15] A. E. Eichenberger and E. S. Davidson. Efficient formulation for optimal modulo schedulers. In *Proc. of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 194–205, June 1997.

[16] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, Nov. 1994.

[17] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proc. of the 1995 International Conference on Supercomputing*, pages 31–40, July 1995.

[18] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[19] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, Nov. 2005.

[20] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Increasing hardware efficiency with multifunction loop accelerators. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 276–281, Oct. 2006.

[21] K. Fan, H. Park, M. Kudlur, and S. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*, pages 124–133, Apr. 2008.

[22] D. D. Gajski et al. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[23] M. Gokhale and B. Schott. Data-parallel C on a reconfigurable logic array. *Journal of Supercomputing*, 9(3):291–313, Sept. 1995.

[24] M. Gokhale and J. Stone. NAPA C: Compiler for a hybrid RISC/FPGA architecture. In *Proc. of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 126–137, Apr. 1998.

[25] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[26] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, Nov. 1994.

[27] Z. Huang, S. Malik, N. Moreano, and G. Araujo. The design of dynamically reconfigurable datapath coprocessors. *ACM Transactions on Embedded Computing Systems*, 3(2):361–384, 2004.

[28] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[29] Intel Corporation, Santa Clara, CA. *Intel IA-64 Software Developer's Manual*, 2002.

[30] K. Kailas, K. Ebcioğlu, and A. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 133–142, Feb. 2001.

[31] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006.

[32] G. Karypis and V. Kumar. *Metis: A Software Package for Paritioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparce Matrices.* University of Minnesota, Sept. 1998.

[33] M. Kudlur, K. Fan, M. Chu, R. Ravindran, N. Clark, and S. Mahlke. FLASH: Foresighted latency-aware scheduling heuristic for processors with customized datapaths. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 201–212, Mar. 2004.

[34] M. Kudlur, K. Fan, and S. Mahlke. Streamroller: Automatic synthesis of prescribed throughput accelerator pipelines. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 270–275, Oct. 2006.

[35] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Journal of Design & Test of Computers*, 20(1):26–33, Jan. 2003.

[36] J. Lee, Y. Hsu, and Y. Lin. A new integer linear programming formulation for the scheduling problem in data-path synthesis. In *Proc. of the 1989 International Conference on Computer Aided Design*, pages 20–23, 1989.

[37] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 111–122, 2002.

[38] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, MA, 1997.

[39] R. Leupers. *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers, Boston, MA, 2000.

[40] J. Llosa et al. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. of the 5th International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[41] J. Llosa and S. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 99–110, 2002.

[42] S. Mahlke et al. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, Nov. 2001.

[43] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[44] B. Mathew and A. Davis. A loop accelerator for low power embedded VLIW processors. In *Proc. of the 2004 International Conference on on Hardware/Software Co-design and System Synthesis*, pages 6–11, 2004.

[45] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003.

[46] S. Memik et al. Global resource sharing for synthesis of control data flow graphs on FPGAs. In *Proc. of the 40th Design Automation Conference*, pages 604–609, June 2003.

[47] Motorola. *CPU12 Reference Manual*, June 2003. http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf.

[48] M. E. J. Newman. The structure and function of complex networks. *Society for Industrial and Applied Mathematics Review*, 45(2):167–256, 2003.

[49] S. Note, W. Geurts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proc. of the 28th Design Automation Conference*, pages 597–602, June 1991.

[50] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1998.

[51] OpenCores. OpenRISC 1200, 2006. http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200.

[52] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 308–315, Dec. 1998.

[53] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006.

[54] I. Park and C. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Proc. of the 28th Design Automation Conference*, pages 680–685, 1991.

[55] N. Park and F. Kurdahi. Module assignment and interconnect sharing in register-transfer synthesis of pipelined data paths. In *Proc. of the 1989 International Conference on Computer Aided Design*, pages 16–19, Nov. 1989.

[56] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):356–370, Mar. 1988.

[57] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavorial synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.

[58] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[59] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 124–133, Dec. 2000.

[60] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[61] SEMATECH. International technology roadmap for semiconductors, 1999. http://www.itrs.net/.

[62] G. Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proc. of the 10th ACM Symposium on Field Programmable Gate Arrays*, pages 177–186, 2002.

[63] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf.

[64] Texas Instruments. *TMS320C54X DSP Reference Set*, Mar. 2001. http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.

[65] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, July 2006. http://focus.ti.com/lit/ug/spru189g/spru189g.pdf.

[66] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[67] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[68] C. Tseng and D. P. Siewiorek. FACET: A procedure for automated synthesis of digital systems. In *Proc. of the 20th Design Automation Conference*, pages 566–572, June 1983.

[69] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 116–125, 2001.

[70] M. Wazlowski et al. PRISM-II compiler and architecture. In *Proc. of the 1st IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, Apr. 1993.

[71] M. Woh et al. The next generation challenge for software defined radio. In *Proc. of the $7^{th}$ International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 343–354, July 2007.