Modulo Scheduling for Highly Customized Datapaths to Increase Hardware Reusability

Kevin Fan Hyunchul Park Manjunath Kudlur Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{fank, parkhc, kvman, mahlke}@umich.edu

ABSTRACT

In the embedded domain, custom hardware in the form of ASICs is often used to implement critical parts of applications when performance and energy efficiency goals cannot be met with software implementations on a general purpose processor or DSP. The downsides of using ASICs include high non-recurring engineering costs, inability to accommodate changes in the application after production, and inability to reuse hardware for new applications. However, by allowing a degree of post-programmability, the hardware can retain high performance and energy efficiency while increasing flexibility and reusability. The difficulty with programmable custom hardware lies in mapping new applications onto an existing datapath that is both sparse and irregular. This paper proposes a constraint-driven modulo scheduler that maps software-pipelineable loops onto programmable loop accelerator hardware. The scheduler is able to target accelerators with widely varying levels of datapath functional capability and connectivity, and thus, varying degrees of programmability. The paper investigates the ability of the scheduler to map new loops onto existing hardware, which depends on both the degree of programmability of the hardware as well as the similarity of the new loop to the original loop for which the hardware was designed.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Retargetable compilers*

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

The markets for wireless handsets, portable digital assistants, digital cameras, and other special-purpose devices continue to grow explosively, fueled by demand for new functionality, added capabilities, and higher bandwidth. For example, the projected data rates for 4G wireless communication are expected to increase 50 times over current 3G wireless standards. These new devices will require higher performing and more energy-efficient computer systems to meet application performance requirements, while still operating for long periods of time in an untethered environment. These performance and energy demands are in conflict with an increasingly important characteristic, post-programmability. Applications and standards evolve over time making it essential to be able to modify a design during its lifetime. Further, re-use of hardware across platforms and enabling multiple applications to run on the same hardware greatly enhance the value of embedded computer systems. One of the most difficult challenges for designers going forward is achieving high performance and efficiency in conjunction with post programmability.

There are a variety of hardware choices available for embedded computing systems that offer varying levels of performance, energy efficiency, and programmability. Application-specific integrated circuits, or ASICs, are popular in many domains. ASICs are generally hardwired accelerators that implement a single algorithm. ASICs offer designers the highest performance and lowest energy solutions for specific problems. As a result, ASICs are used to do the heavy lifting for performance-critical portions of a workload. The downsides of ASICs are programmability and hardware re-use, where they have little to offer due to the hardwired nature of the design. At the other end of the spectrum are microprocessors and DSPs. Processors offer full programmability and thus the ability to execute a wide range of applications, enabling designers to seamlessly re-use hardware. The downsides of processors are performance and efficiency. Processors, even the highest end DSPs, are often incapable of sustaining the needed computation rates for compute-intensive kernels. Further, the energy efficiency of processors is typically 100-1000x worse than an ASIC. The overhead of instruction execution and the use of regular datapaths in processors are two of the largest reasons for this loss of efficiency.

Several middle ground solutions have emerged that promise to deliver both post-programmability and high performance/efficiency. FPGAs provide a fully configurable hardware substrate for implementing any design, thus they provide full programmability. However, the performance and energy efficiency of FPGAs may not be high due to the inherent overhead of gate-level reconfigurability. An alternative design style that sacrifices gate-level reconfigurability is the coarse-grained reconfigurable architecture (CGRA) [5, 9, 16, 27, 19]. CGRAs consist of an array of function units (FUs) interconnected by an interconnection network. Register files are distributed throughout the CGRAs to hold temporary values and are accessible only by a subset of FUs. CGRAs typically have shorter reconfiguration times, lower delay characteristics, and lower power consumption than FPGAs. A final middle-ground solution are pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.
Copyright 2008 ACM 978-1-59593-978-4/08/04 ...$5.00.

grammable loop accelerators. The Reconfigurable Streaming Vector Processor (RSVP) is a coprocessor architecture that accelerates streaming data operations [3]. Mathew proposes a loop accelerator for low power embedded VLIW processors [18]. These accelerators are generally targeted at vector-style loops.

The goal of this paper is to push programmability into a hardware substrate that is as close to an ASIC as possible to retain the desirable performance and efficiency characteristics. The target hardware platform is a stylized loop accelerator (LA) that is customized for a single application [26, 7, 8]. The LA is a direct hardware realization of a modulo scheduled loop [24]. Each LA has a specialized datapath, including FUs, register files, and interconnect, and a simple controller driven by the initiation interval of the schedule. We generalize the structure of the base LA template to facilitate a small degree of post-programmability. However, the datapath is still highly specialized with point-to-point interconnect, fixed-capability FUs, and limited storage capacity. Such a platform cannot execute any modulo scheduled 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.

The central challenge with this approach is the compiler, specifically modulo scheduling a loop onto a highly irregular datapath. Traditional modulo schedulers rely on datapath regularity, such as centralized register files and uniform interconnect between FUs. Multicluster VLIW designs introduce differential communication latencies into the scheduler [21, 25]. CGRA scheduling has the most similarities with this work [19, 12, 23]. Here, loops are scheduled onto a highly decentralized architecture. Explicit routing of data values must be performed due to the distributed organization of the CGRA. However, interconnect and functionality are highly regular, enabling almost any loop to be mapped onto the hardware. Our problem is different because the hardware is not regular, but rather is highly customized to a particular application.

In this paper, we propose constraint-driven modulo scheduling for mapping applications onto irregular datapaths. The technique leverages satisfiability modulo theory (SMT) commonly used in the computer-aided design community. Mapping of operations onto the irregular LA datapath is modeled as a set of constraints within a SMT problem that attempts to find time and space assignments for the operations subject to the limitations of the datapath.

One of the most difficult aspects of this work is quantifying success. From the outset, we know that the programmability of the underlying LA hardware is limited, thus it is unlikely arbitrary loops will successfully map onto a particular LA. In fact, that is not the objective. We introduce a loop perturbation module that takes a dataflow graph for a target loop and iteratively introduces random modifications to the graph, such as creating new operations or edges, removing operations or edges, and changing existing nodes or edges. In essence, these perturbations synthetically create an arbitrary number of loops that progressively become more dissimilar to the base loop. These synthetic loops serve as approximations for either source modifications to an original loop or loops with similar computation structure. We conduct a set of experiments to map the perturbed loops onto the LA to evaluate the effectiveness of the scheduler and demonstrate the programmability of the LA. We also present a more conventional experiment of mapping multiple loops onto LAs designed for one application.

The contributions of this paper are as follows:

- A parameterized programmable loop accelerator (PLA) template that offers programmability for a range of applications with similar computation structure.

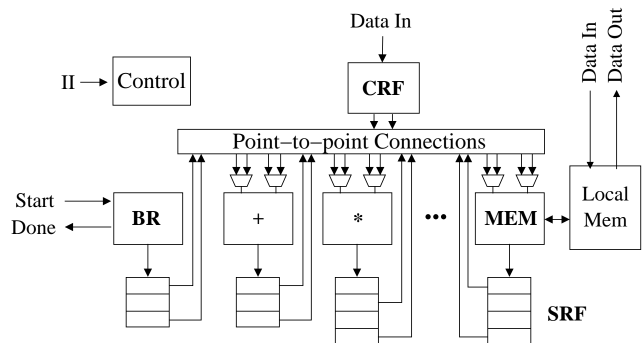

Figure 1: Template for fixed-function loop accelerator.

- An SMT formulation of modulo scheduling for an innermost loop onto a PLA.
- An evaluation of PLA programmability across a set of media processing loops and synthetically generated variations of these loops.

2. LOOP ACCELERATOR ARCHITECTURE

A fixed-function loop accelerator is used as the baseline in this paper. This accelerator is designed to execute a specific loop at a given performance level, and is not programmable. Then, starting from the fixed-function baseline, the datapath is generalized to create a more programmable design. The goal is to remove or relax the most restrictive parts of the architecture that limit programmability, while retaining the efficiency available through customization. In this section, the architecture and synthesis flow for the fixed-function accelerator is described first. Then, the datapath generalizations are described.

2.1 Fixed-function Accelerator

Figure 1 shows the hardware schema used in this paper for the fixed-function accelerator [26, 7]. The accelerator is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU performs a specific set of functions that is tailored for the particular loop. Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register. The entries in a SRF therefore contain the values produced by the corresponding FU in the order they were computed. Wires from the registers back to the FU inputs allow data transfer from producers to consumers. 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. In addition, a central register file (CRF) holds static live-in register values which cannot be stored in the SRFs.

The design flow for the fixed-function accelerator is shown in Figure 2 [7]. The first step in the loop accelerator synthesis process is the creation of an abstract VLIW architecture to which the application is mapped. 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 (subject to recurrence constraints), and a library of hardware cell capabilities and costs, the *FU allocation* stage generates a mix of FUs for the abstract architecture that minimizes FU cost while providing sufficient resources to meet the performance constraint.

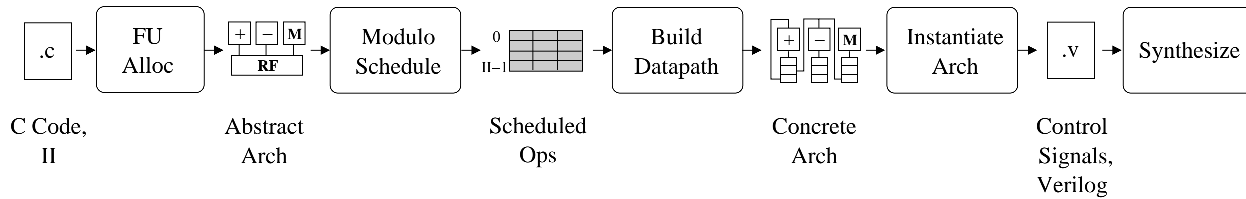

Figure 2: Fixed-function accelerator design flow.

Next, the loop is modulo scheduled to the abstract architecture. Modulo scheduling is a method of overlapping iterations of a loop to achieve high throughput [24]. 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. The scheduler assigns the operations in the loop to FUs and time slots, satisfying all inter-operation dependences and meeting the II requirement. After scheduling, the accelerator datapath is derived from the producer-consumer relationships in the schedule. This includes setting the widths of the FUs and the widths and depths of the SRFs, and connecting specific SRF entries with the appropriate FU inputs.

Based on the datapath, the control path for the accelerator is generated. This consists of a modulo-II counter which directs FU execution (for FUs capable of multiple operations) and controls the MUXes at the FU inputs.

Finally, a Verilog realization of the accelerator is generated by emitting modules with pre-defined behavioral Verilog descriptions that correspond to the datapath elements. Gate-level synthesis and placement/routing are then performed on the Verilog output.

2.2 Programmable Accelerator

To build a programmable loop accelerator (PLA), the datapath features of the fixed-function accelerator that are least flexible should be generalized in an efficient manner. Key accelerator datapath characteristics include:

Functionality. The accelerator 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 accelerator. FUs can be generalized for a low additional cost by adding functionality which is complementary to existing functionality. For example, any adder can be generalized to support both addition and subtraction with low additional cost. Also, any logic unit can be extended to support all logical operations, as they are relatively low cost.

Shift register files. Another limiting aspect of the fixed-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. This introduces some additional hardware, namely base registers, adders, and decoders for the read and write ports. However, the rotating register files remain small (thus the width of base registers and adders is often 2 bits or less) and are highly distributed, minimizing their cost impact.

Point-to-point connectivity. A major area in which the accelerator 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 fixed-function accelerator. This means 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.

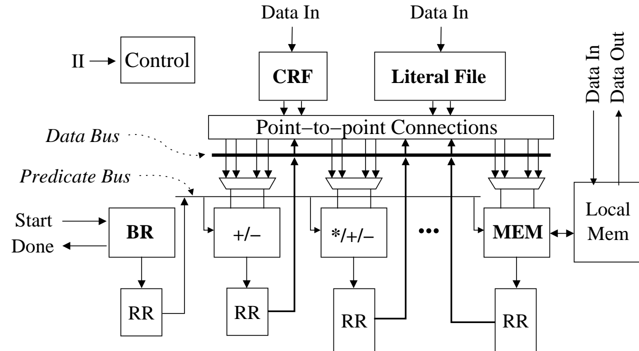

Figure 3: Template for programmable loop accelerator.

Port-specific connectivity. In the fixed-function accelerator, 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 can sometimes result in a successful schedule; however, to relax this constraint more generally, the input MUXes in the PLA are widened to allow each input port to read its operand from any connection made to either port.

Staging predicate. The accelerator 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 fixed-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.

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

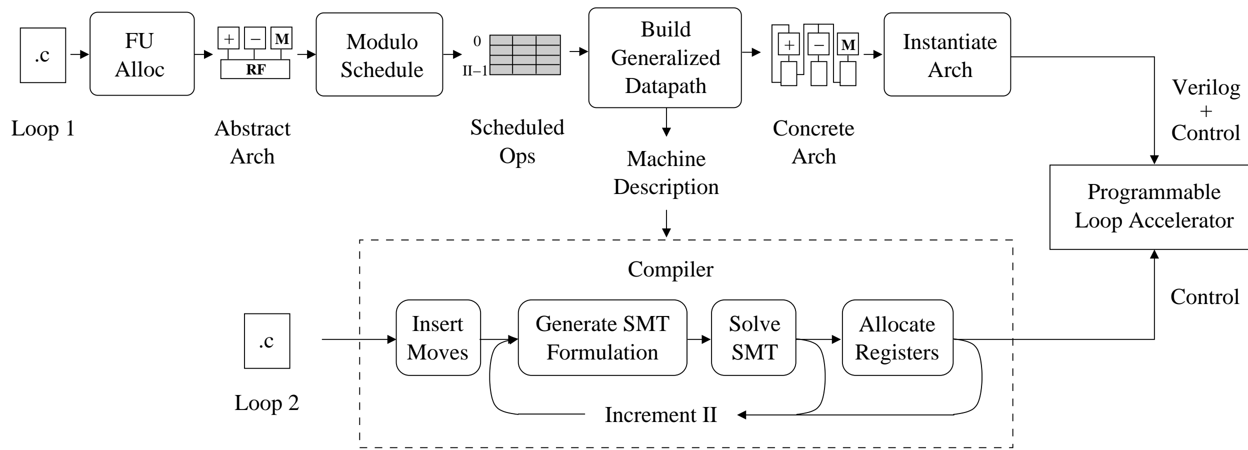

Figure 5: Design and compilation flow for programmable loop accelerator.

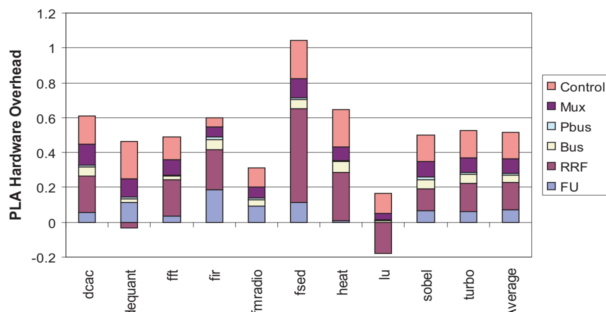

Figure 4: Hardware cost breakdown of PLA generalizations.

Hardwired control. In the single-function accelerator, 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 maximum allowed II. In addition, in the fixed-function accelerator, 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 register file, different literal values may be used for different loops.

Programmable accelerator architecture. Figure 3 shows the template for the PLA, generalized from the datapath shown in Figure 1. The accelerator is designed for a specific loop at a specific throughput, but contains a more general datapath than the fixed-function accelerator 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 rotating register files; and the FU input MUXes are widened.

Figure 4 shows estimates for the hardware cost of each of these datapath generalizations for PLAs designed for loops from various DSP and media applications. The most costly components are the rotating register files (RRFs) and the control, at 15% each. One source of RRF cost is that the number of registers in each file is rounded up to the nearest power of 2. Note that in two cases, the use of RRFs actually results in a cost savings over SRFs due to better packing of lifetimes. On average, the total overhead is about 51%; this is acceptable considering that the total area of a loop accelerator is quite small (all designs are less than 0.6 mm^2 when synthesized in 0.13μ technology). A more detailed analysis of the hardware tradeoffs for PLAs is beyond the scope of this paper. Rather, the PLA design serves as a target for the compiler.

The augmented design flow for PLAs is shown in Figure 5. During the creation of the hardware, the datapath is customized for a given loop (labeled Loop 1) but is also generalized using the techniques described above. Additional control logic is generated to support the programmable features of the accelerator. 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) to map a new loop (labeled Loop 2) onto the same hardware.

3. CONSTRAINT-DRIVEN SCHEDULING

3.1 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

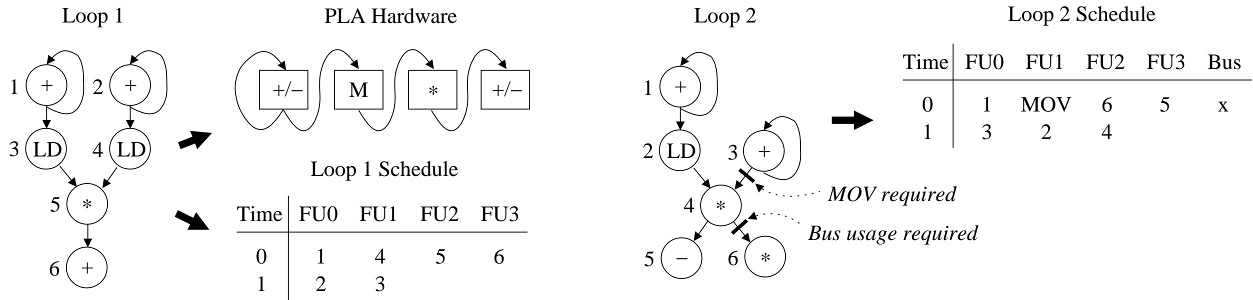

Figure 6: PLA synthesis and scheduling example.

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 5 shows the compiler flow.

3.2 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:

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

$$X_{i,f_1,t_1} \wedge X_{i,f_2,t_2} = false \qquad \forall i \in V, f_1 \neq f_2, t_1 \neq t_2 \tag{2}$$

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:

$$X_{i_1,f,t} \wedge X_{i_2,f,t} = false \qquad \forall f \in F, t \in \{0, II-1\}, i_1 \neq i_2 \tag{3}$$

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:

$$(S_j \times II) + t_j \geq (S_i \times II) + t_i + l_{i,j} - (d_{i,j} \times II) \tag{4}$$

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:

$$\neg X_{i,f_i,t_i} \vee \neg X_{j,f_j,t_j} \vee expression(4) \tag{5}$$

Constraint (5) 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 (5) 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 (5) 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:

$$\neg \left(\bigvee_{t_i=0}^{II-1} X_{i,f_i,t_i} \right) \vee \neg \left(\bigvee_{t_j=0}^{II-1} X_{j,f_j,t_j} \right) \tag{6}$$

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

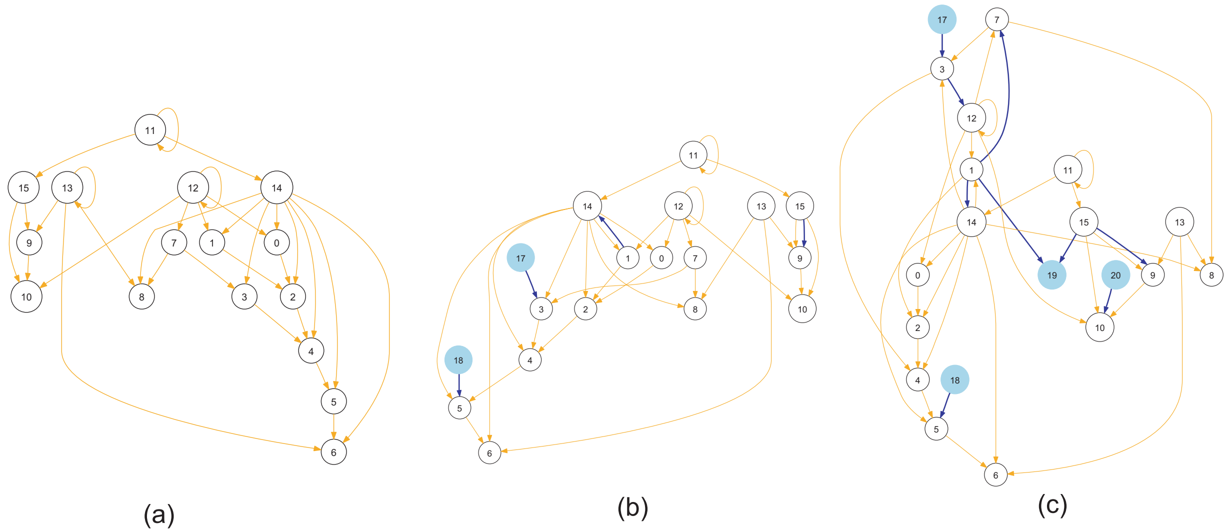

Figure 7: Graph perturbation example from "heat" benchmark: (a) original loop, (b) after 5 perturbations, (c) after 10 perturbations.

not directly connected, constraint (6) may be modified to allow use of the global bus:

$$\neg \left(\bigvee_{t_i=0}^{II-1} X_{i,f_i,t_i} \right) \vee \left(\bigwedge_{t_j=0}^{II-1} \neg X_{j,f_j,t_j} \vee B_{j,t_j} \right) \tag{7}$$

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

$$B_{i_1,t} \wedge B_{i_2,t} = false \qquad \forall t \in \{0, II-1\}, i_1 \neq i_2 \tag{8}$$

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 (1) through (8) gives a legal modulo schedule with initiation interval II for the graph G on a given PLA datapath.

3.3 Example

The left side of Figure 6 shows a portion of the loop from the fir filter application. The given II is 2, and the resulting PLA hardware contains two adders which have been generalized to adder-subtracters, one memory unit, and one multiplier, as shown. Registers are omitted from the figure for clarity. On the right side of the figure, another loop is scheduled onto the same PLA. The second loop is somewhat similar, but the data transfer from operation 3 to operation 4 requires a connection from an adder to a multiplier, which does not exist in the hardware. Thus, a MOV is required to transfer the result of operation 3 through the memory unit to the multiplier. Also, the result of operation 4 feeds operation 6, which requires another connection that does not exist in the hardware. A transfer is scheduled on the bus to send the result of operation 4 back to the multiplier. Thus, the second loop can be scheduled at its ResMII of 2 on the PLA designed for the first loop.

4. 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 7(a) shows the dataflow graph for "heat". After 5 perturbations, the graph is as shown in Figure 7(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 7(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 2.2, 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 loop to be successfully mapped.

5. EXPERIMENTAL RESULTS

Loop kernels from various DSP and media applications are used to evaluate the programmability of the PLA and the ability of the constraint-based scheduler to map loops onto PLAs. For each loop, a PLA is synthesized using the system described in Section 2.2. Table 1 shows the characteristics of each loop and its corresponding PLA. The Yices SMT solver [4] is used to modulo schedule loops onto PLAs using the formulation described in Section 3.2. 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.

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 1: Loop kernels from DSP and media applications.

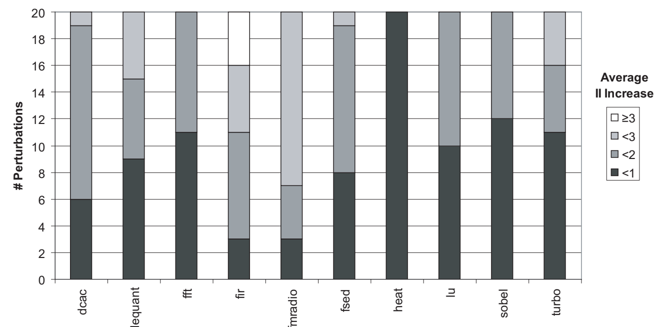

Figure 8: II increase necessary to schedule loops with perturbations.

For the perturbation study, we run a set of experiments wherein each loop is randomly perturbed a number of times as described in Section 4. 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 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 8 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 1 under the "Base II" column.

Some loops, such as the "fmradio" loop, are no longer able to achieve the original II after just 3 perturbations. Others are able to achieve the original II even after a large number of perturbations, particularly the "heat" kernel. Most of the loops can still be mapped onto a PLA after 20 perturbations with small increases in II.

Figure 9 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

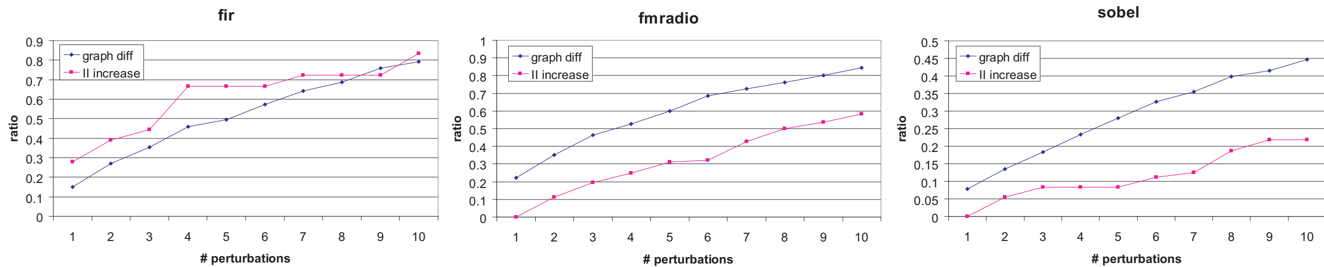

Figure 9: Relative II increase and graph difference vs. perturbations for fir, fmradio, and sobel.

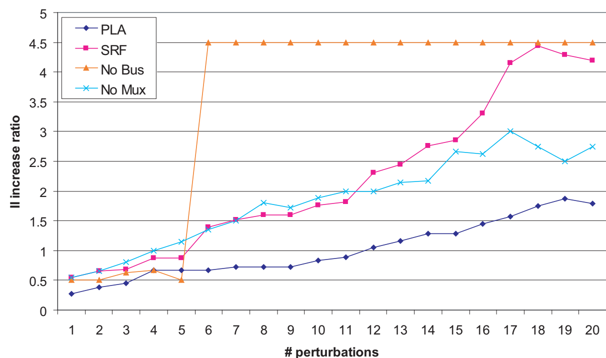

Figure 10: Perturbation studies with more restrictive PLAs.

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 [20], 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 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 utility of the architecture generalizations described in Section 2.2, the same perturbation study is run with more restrictive PLA hardware. Figure 10 shows the results of scheduling the "fir" benchmark to 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 achievable II is significantly lower as the number of perturbations increases.

Figure 11 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).

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 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. RELATED WORK

Prior work in compilation for irregular datapaths can best classified by the target architecture: CGRAs, multicluster VLIWs, and DSPs.

CGRA scheduling. Several modulo scheduling techniques for

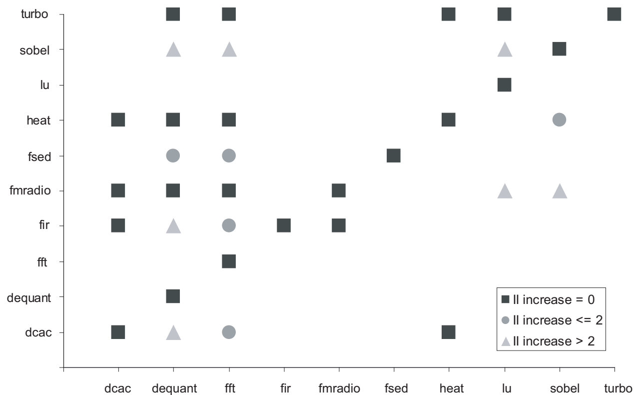

Figure 11: Cross compilation results.

	dcac	deq	fft	fir	fmr	fsed	heat	lu	sob
turb	1.41	1.27	1.50	1.76	1.67	1.65	1.53	1.35	1.44
sob	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				
fmr	1.24	1.13	0.82	0.81					
fir	1.24	1.41	0.96						
fft	1.25	0.97							
deq	0.89								

Table 2: Similarity of loop kernels; a lower number means the two loops are more similar to each other.

CGRAs have been proposed. [19] 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 [23]. 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 [12]. 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. [28] 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 [13]. 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 [6, 21, 22, 25, 1, 10, 2]. 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 [6], or during scheduling, such as the Unified Assign and Schedule algorithm used in the Lego compiler [22]. 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 [14, 15, 17]. 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 processors with partial register bypass networks [11, 23]. 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.

7. CONCLUSION

This paper presents a constraint-driven modulo scheduler which maps software-pipelineable loops onto a generalized loop accelerator architecture. The loop accelerator is customized for a specific loop, giving significant area and energy efficiency gains over general-purpose hardware, but contains features that enable other similar loops to be mapped onto the same hardware. The architecture template and constraint-driven scheduler are evaluated using a graph perturbation method which allows for systematic exploration of the relationship between loop similarity and hardware reusability.

8. ACKNOWLEDGMENTS

We thank Yuanyuan Tian for her help with quantifying graph similarity, as well as the anonymous referees who provided excellent feedback. This research was supported by the National Science Foundation grants CNS-0615261 CCF-0347411, ARM Ltd, and Samsung Advanced Institute of Technology.

9. REFERENCES

[1] ALETÀ, A., CODINA, J., SÁNCHEZ, J., AND GONZÁLEZ, A. Graph-partitioning based instruction scheduling for clustered processors. In *Proc. of the 34th Annual International Symposium on Microarchitecture* (Dec. 2001), pp. 150–159.

[2] CHU, M., FAN, K., AND MAHLKE, S. Region-based hierarchical operation partitioning for multicluster processors. In *Proc. of the SIGPLAN '03 Conference on*

Programming Language Design and Implementation (June 2003), pp. 300–311.

[3] CIRICESCU, S., ESSICK, R., LUCAS, B., MAY, P., MOAT, K., NORRIS, J., SCHUETTE, M., AND SAIDI, A. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture* (2003), pp. 141–150.

[4] DE MOURA, L., AND DUTERTRE, B. Yices 1.0: An efficient SMT solver. In *The Satisfiability Modulo Theories Competition (SMT-COMP)* (Aug. 2006).

[5] EBELING, C., ET AL. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines* (Apr. 1997), pp. 106–115.

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

[7] FAN, K., KUDLUR, M., PARK, H., AND MAHLKE, S. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture* (Nov. 2005), pp. 219–230.

[8] FAN, K., KUDLUR, M., PARK, H., AND MAHLKE, S. Increasing hardware efficiency with multifunction loop accelerators. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis* (Oct. 2006), pp. 276–281.

[9] GOLDSTEIN, S., ET AL. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture* (June 1999), pp. 28–39.

[10] KAILAS, K., EBCIOĞLU, K., AND AGRAWALA, A. CARS: A new code generation framework for clustered ILP processors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture* (Feb. 2001), pp. 133–142.

[11] KUDLUR, M., FAN, K., CHU, M., RAVINDRAN, R., CLARK, N., AND MAHLKE, S. FLASH: Foresighted latency-aware scheduling heuristic for processors with customized datapaths. In *Proc. of the 2004 International Symposium on Code Generation and Optimization* (Mar. 2004), pp. 201–212.

[12] LEE, J., CHOI, K., AND DUTT, N. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Journal of Design & Test of Computers 20*, 1 (Jan. 2003), 26–33.

[13] LEE, W., PUPPIN, D., SWENSON, S., AND AMARASINGHE, S. Convergent scheduling. In *Proc. of the 35th Annual International Symposium on Microarchitecture* (2002), pp. 111–122.

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

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

[16] LU, G., SINGH, H., LEE, M.-H., BAGHERZADEH, N., KURDAHI, F. J., AND FILHO, E. M. C. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference* (1999), pp. 727–734.

[17] MARWEDEL, P., AND GOOSSENS, G. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[18] MATHEW, B., AND DAVIS, A. 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* (2004), pp. 6–11.

[19] MEI, B., 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* (Mar. 2003), pp. 296–301.

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

[21] NYSTROM, E., AND EICHENBERGER, A. E. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture* (Dec. 1998), pp. 103–114.

[22] ÖZER, E., BANERJIA, S., AND CONTE, T. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proc. of the 31st Annual International Symposium on Microarchitecture* (Dec. 1998), pp. 308–315.

[23] PARK, H., FAN, K., KUDLUR, M., AND MAHLKE, S. 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* (Oct. 2006), pp. 136–146.

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

[25] SÁNCHEZ, J., AND GONZÁLEZ, A. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proc. of the 33rd Annual International Symposium on Microarchitecture* (Dec. 2000), pp. 124–133.

[26] SCHREIBER, R., ET AL. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing 31*, 2 (2002), 127–142.

[27] TAYLOR, M. B., ET AL. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro 22*, 2 (2002), 25–35.

[28] VENKATARAMANI, G., NAJJAR, W., KURDAHI, F., BAGHERZADEH, N., AND BOHM, W. 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* (2001), pp. 116–125.