POLYMORPHIC PIPELINE ARRAY: A FLEXIBLE MULTICORE ACCELERATOR FOR MOBILE MULTIMEDIA APPLICATIONS

by

Hyunchul Park

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
2009

Doctoral Committee:
Associate Professor Scott A. Mahlke, Chair
Professor Trevor N. Mudge
Professor James S. Freudenberg
Assistant Professor Satish Narayanasamy
Dr. Hong-seok Kim, Bain & Company

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.

I would also like to thank my thesis committee, Professors Trevor Mudge, James Freudenberg, Satish Narayanasamy, and Dr. Hong-Seok Kim. They donated their time, providing valuable comments and suggestions that helped me refine my thesis.

The research presented in this dissertation is 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. In particular, Kevin Fan gave me valuable help in virtually every aspect of my graduate school life: debugging codes, writing papers, and even fixing my long-suffering 240sx. Manjunath Kudlur also contributed significantly, helping me write my first publication. More recently, Yongjun Park has been unfailingly supportive in performing hardware experiments for the token network and the PPA.

I would also like to thank people at Samsung Advanced Institute of Technology:

Sukjin Kim, Heeseok Kim and Taewook Oh. They helped me set up the compiler environment for my research and provided numerous application benchmarks that were extensively used in my dissertation.

As much as those who provided technical expertise, those who offered engaging conversation and moral support were crucial to my graduate school experience, namely: Amin Ansari, Jay Blome, Hyounkyu Cho, Mike Chu, Nate Clark, Ganesh Dasika, Shuguang Feng, Shantanu Gupta, Jeff Hao, Amir Hormati, Po-Chun Hsu, Steve Lieberman, Yuan Lin, Mojtaba Mehrara, Rob Mullenix, Rajiv Ravindran, Mark Woh, 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. My parents and my brother Sungchul provided their unconditional love and support throughout this whole process. Finally, I am grateful to You-sun Chung for her love and support. The occasional carrot and/or stick she would offer presumably to ward off procrastination and keep me on track seemed to yield uneven results at best. That is, until now.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

POLYMORPHIC PIPELINE ARRAY: A FLEXIBLE MULTICORE ACCELERATOR FOR MOBILE MULTIMEDIA APPLICATIONS

by

Hyunchul Park

Chair: Scott A. Mahlke

Mobile computing in the form of smart phones, netbooks, and PDAs has become an integral part of our everyday lives. Moving ahead to the next generation of mobile devices, we believe that multimedia will become a more critical and product-differentiating feature. High definition audio and video as well as 3D graphics provide richer interfaces and compelling capabilities. However, these algorithms also bring different computational challenges than wireless signal processing. Multimedia algorithms are more complex featuring more control flow and variable computational requirements where execution time is not dominated by innermost vector loops. Further, data access is more complex where media applications typically operate on multi-dimensional vectors of data rather than single-dimensional vectors with sim-

ple strides. Thus, the design of current mobile platforms requires re-examination to account for these new application domains.

In this dissertation, we focus on the design of a programmable, low-power accelerator for multimedia algorithms referred to as a *Polymorphic Pipeline Array* (PPA). The PPA design is inspired by coarse-grain reconfigurable architectures (CGRAs) that consist of an array of function units interconnected by a mesh style interconnect. The PPA improves upon CGRAs by attacking two major limitations: scalability and acceleration limited to innermost loops. The large number of resources are fully utilized by exploiting both fine-grain instruction-level and coarse-grain pipeline parallelism, and the acceleration is extended beyond innermost loops to encompass the whole region of applications.

Various compiler and architectural optimizations are presented for CGRAs that form the basic building blocks of PPA. Two compiler techniques are presented that systematically construct the schedule with intelligent heuristics. Modulo graph embedding leverages graph embedding technique for scheduling in CGRAs and edge-centric modulo scheduling provides a communication-oriented way to address the scheduling problem. For architectural improvement, a novel control path design is presented that leverages the token network of dataflow machines to reduce the instruction memory power.

The PPA is designed with flexibility and programmability as first-order requirements to enable the hardware to be dynamically customizable to the application. A PPA exploit pipeline parallelism found in streaming applications to create a coarse-

grain hardware pipeline to execute streaming media applications. PPA resources are allocated to each stage depending on its size and ability to exploit fine-grain parallelism. For dynamic partitioning of resources, Virtualized modulo scheduling generates a unified schedule that can be easily converted to target different number of resources at run-time.

CHAPTER 1

Introduction

Mobile computing has become a ubiquitous part of society. More than half the world's population now owns on a cell phone, and in some countries, the number of active cell phone contracts out numbers the population. The embedded computer systems that power mobile devices demand high performance and energy efficiency to operate in an untethered environment. Traditionally, hardwired accelerators have done the heavy lifting in terms of computation. Mobile platforms are designed as heterogeneous systems-on-a-chip consisting of multiple processors (general-purpose and/or digital signal processors) and special purpose accelerators constructed for the most compute-intensive tasks. The performance/energy point achieved by these designs is impressive - performing tens of giga-operations per second at sub-Watt power levels.

Moving forward, there is a need to create more programmable mobile computing platforms. Programmable solutions offer several key advantages:

- *Multi-mode operation* is enabled by running multiple application standards

(e.g., two video codecs) or even multiple applications on the same hardware. Accelerator-based solutions require a union of hardware blocks to accomplish all desired applications.

- *Time to market* of an implementation is lower because the hardware can be re-used across multiple platforms. More importantly, hardware integration and software development can progress in parallel.

- *Prototyping and software bug fixes* are enabled on existing silicon with a software change. On-going evolution of specifications are supported in a natural way by allowing software changes after the chipset and even the device have been manufactured.

- *Chip volumes* are higher as the same chip can support multiple standards without requiring hardware changes.

Traditionally, the design of programmable mobile computing platforms has focused on software defined radio [4, 3, 17, 35, 58]. These systems are geared towards wireless signal processing that contain vast amounts of vector parallelism. As a result, single-instruction multiple-data (SIMD) hardware is recognized as an effective strategy to achieve both high-performance and programmability. SIMD provides high efficiency because of its regular structure, ability to scale lanes, and low control cost. However, mobile computing systems are not limited to wireless signal processing. High-definition video, audio, 3-D graphics, and other forms of media processing are high value applications for mobile terminals. In fact, many believe the quality and

types of media support will be the key differentiating factors of future mobile terminals.

Media applications in a mobile environment offer a number of different challenges than wireless signal processing. First, the complexity of media processing algorithms is typically higher than signal processing. Computation is no longer dominated by simple vectorizable innermost loops. Instead, loop bodies are larger with significant amounts of control flow to handle the different operating modes and inherent complexity of media coding. This results in differential dynamic computational requirements. Further, significant time is spent in outer loops and acyclic code regions. As a result, SIMD parallelism is less prevalent and less efficient to exploit in media algorithms [37]. Second, the data access complexity in media processing is higher. Signal processing algorithms typically operate on single dimension vectors, whereas video algorithms operate on two or three dimensional blocks of data where the block size is variable. Thus, video and other forms of media processing push designs to have higher bandwidth and more flexible memory systems. Finally, the power budget is generally more constrained for media processing than wireless signal processing because of higher usage times.

To address these challenges, this work focuses on the design of a flexible media accelerator for mobile computing referred to as a *polymorphic pipeline array* or PPA. Our design does not exploit SIMD parallelism, but rather relies on two forms of pipeline parallelism: coarse-grain pipeline parallelism found in streaming applications [19, 20, 27] and fine-grain parallelism exploited through modulo scheduling

of innermost loops [48]. The PPA consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network and a shared memory. Groups of four PEs form cores that are driven by a single instruction stream. These cores can execute tasks (filters in a streaming application) independently or neighboring cores can be coalesced to execute loops with high degrees of parallelism. The use of a regular interconnection fabric allows the core boundaries to be blurred, thereby allowing the hardware to be customized differently for each application.

The PPA design is inspired by coarse-grain reconfigurable architectures (CGRAs) that consist of an array of function units interconnected by a mesh style interconnect [38, 39]. In CGRAs, small register files are distributed throughout the array to hold temporary values and are accessible only by a small subset of function units. Example commercial CGRA systems that target mobile devices are ADRES [39], MorphoSys [36], and Silicon Hive [47]. Tiled architectures, such as Raw, are closely related to CGRAs though are not intended for mobile computing [52]. The abundance of resources in CGRAs offer large raw computation capabilities while the distributed nature of hardware and simple control provide low energy efficiency.

In this dissertation, we first attack the major challenges for deploying CGRAs in embedded environments. Two compilation techniques for efficient mapping of applications onto the highly distributed architectures like CGRAs are proposed. Also, a novel design of control path in CGRAs is proposed for reducing instruction read power. These optimizations can be directly applied to the PPA since it builds on the basic building blocks of CGRAs. Then, we provide the application analysis that

motivates the PPA design paradigm, the hardware description of PPA, and a new compilation technique to maximize the utilization of the proposed PPA.

1.1 Optimizations for CGRAs

1.1.1 Compiler Support

The most difficult challenge in deploying CGRAs arises in the compiler support. An effective compiler is essential for exploiting the abundance of computing resources available on a CGRA. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of a compiler. Traditional schedulers that just assign an FU and time slot to each operation are insufficient because they do not take routing into consideration. Scalar operand values must be explicitly routed between producing and consuming operations. Further, dedicated routing resources are not provided. Rather, an FU can serve either as a compute resource or as a routing resource at a given time. A compiler scheduler must thus manage the computation and flow of operands across the array to effectively map applications onto CGRAs. Chapter 2 provides the overview of CGRAs and major scheduling challenges.

This dissertation proposes novel scheduling techniques that collectively synthesize an efficient mapping of an application to the CGRA with a reasonable compile time. Specifically, the new techniques are developed in the context of modulo scheduling. Modulo scheduling is a software pipelining technique that overlaps the execution of

loop iterations to provide the opportunity to exploit both loop-level and instruction-level parallelism. Modulo scheduling is especially effective for CGRAs since they provide a large number of resources for exploiting potential parallelism.

In Chapters 3 and 4, we propose two modulo scheduling approaches that differ in their primary objective of scheduling and we categorize them as node-centric and edge-centric approaches. The first approach in Chapter 3, referred as *modulo graph embedding*, is a modulo scheduling technique for CGRAs that leverages graph embedding commonly used in graph layout and visualization. This technique is node-centric in that the focus of scheduling is assigning operations (nodes in dataflow graphs) to FUs, just as in traditional schedulers.

The second approach in Chapter 4, on the other hand, considers routing operands between operations (edges in dataflow graphs) as its primary objective. Operation assignment to FUs can be viewed as a by-product of a successful route, thus no successive placement step is required. In essence, by getting an operand between two points, the necessary operations can be performed along the way for free. We refer to this technique as *edge-centric modulo scheduling*, or EMS.

1.1.2 Control Path Optimization

A major bottleneck for deploying CGRAs into a wider domain of embedded devices lies in the control path. The appealing features in the datapath of CGRAs ironically come back as a major overhead in the control path. The distributed interconnect and register files require a large number of configuration bits to route values across

the network. The abundance of computation resources simply adds up the list for configurations to the control path. As a result, the total number of control bits to configure the whole array can reach nearly 1000 bits each cycle, and the control path takes up to 43% of the total power consumption in existing CGRA designs [25, 5]. In Chapter 5, we propose a novel control path design in CGRAs that leverages a token network in dataflow machines to improve the code efficiency.

1.2 Polymorphic Pipeline Array

The major weakness of CGRAs is their lack of ability to accelerate an entire application. Chapter 6 propose a flexible multicore accelerator that can accelerate multiple regions of a target application. The PPA shares the inherent hardware efficiency of CGRAs: fully distributed register files, nearest neighbor interconnect and simple control. An application study is performed to discover the available parallelism in today's mobile multimedia applications. The result of this study motivates the design of PPA that can support multiple levels of parallelism. Also, a novel scheduling technique is proposed that enables virtualized execution of loops.

CHAPTER 2

Background and Motivation

2.1 CGRA Overview

A CGRA consists of an array of compute nodes, each of which executes word-level operations, communicating through an interconnection network. In general, CGRA designs can be described by four characteristics: size, node functionality, network configuration, and register file sharing. The *size* refers to the number of nodes; commonly this can vary from 4 nodes arranged in a row up to 64 nodes arranged in an 8×8 grid. The *functionality* of each node can vary from a single FU (e.g. adder or subtracter), to an ALU, to a full-blown processor. In addition, the functionality of nodes may be homogeneous or heterogeneous. For example, only a subset of nodes may access data memory.

There are a large number of potential *network configurations*, such as connections between each node and its four (or eight diagonal) nearest neighbors, buses connecting each node to (possibly to a subset of) other nodes in the same row or column,

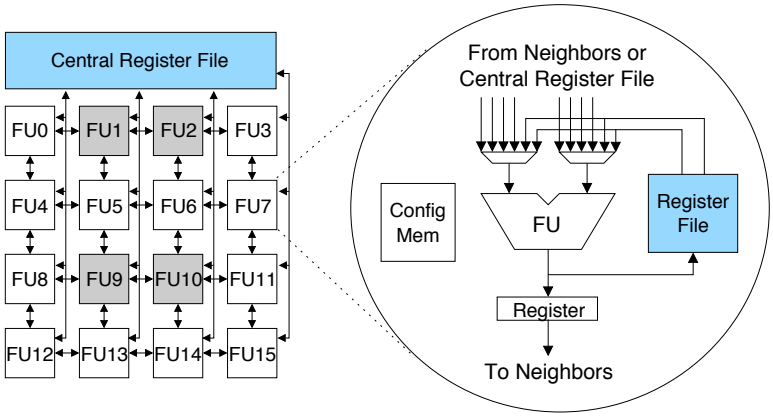


Figure 2.1: Example CGRA design

hierarchical connection schemes, and so on. Finally, the degree of *register file sharing* ranges from small, individual register files at each node, to multiple register files each shared by a small number of nodes, to a single central register file accessible by some or all nodes.

Figure 2.1 shows a CGRA design that contains 16 nodes arranged in a 4×4 mesh; each node can communicate with its four nearest neighbors. In addition, column buses connect each node to a central register file. Each node consists of an FU that can read inputs from neighbors or the central register file and write to a single output register; a small, dedicated register file; and a configuration memory to supply control signals to the MUXes, FU, and register file. Certain operations, such as loads and stores, can only be executed on a subset of FUs (shaded). Note that a node can either perform a computation or route data each cycle, but not both, as routing is accomplished by passing data through the FU. This architecture is only one possible CGRA design; many other variations are possible.

2.2 Modulo Scheduling Challenges

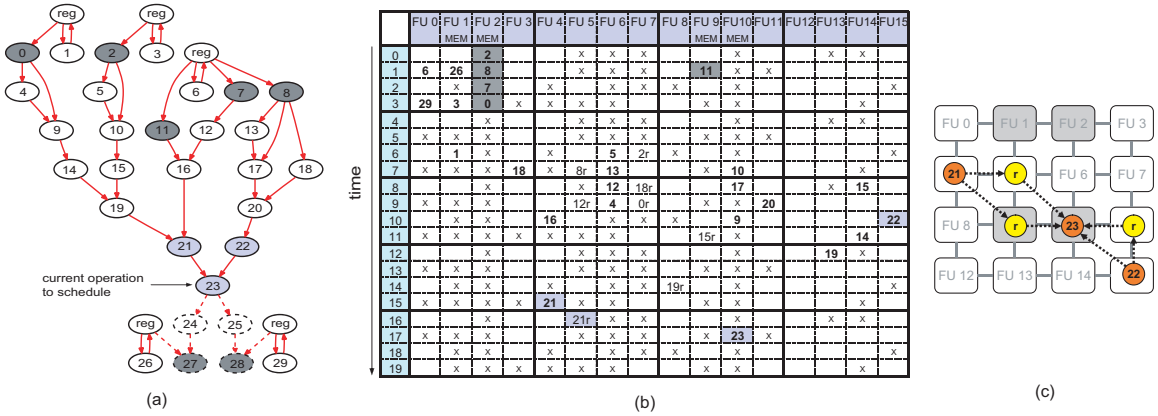


Figure 2.2: Example to illustrate the challenges of CGRA scheduling: (a) the dataflow graph for the fsed application, (b) the reservation table for a partial schedule on a 4x4 array, (c) possible routings from 23's producers. In (a) and (b), dark grey shading indicates memory operations and light grey shading is used to highlight the current operation being scheduled (node 23) and its immediate predecessors. Bold numbers indicate computation operations, other numbers followed by 'r' (e.g. '8r') indicate routing slots for corresponding computation operations. 'reg' nodes indicate live-in values stored in the central RF.

Modulo scheduling is a software pipelining technique that exposes parallelism by overlapping successive iterations of a loop [48]. The goal is to find a valid schedule such that the interval between successive iterations (initiation interval, or II) is minimized. The II-cycle code region that achieves this maximal overlap is called the kernel. When the number of iterations is large, the performance of the loop is determined by the II to a first order; thus, it is more important to minimize the II than to minimize schedule length. Initially, the scheduler chooses the target II to be the maximum of the resource-constrained lower bound (ResMII) and the recurrence-constrained lower bound (RecMII). If a valid modulo schedule cannot be found, the target II is

incremented and scheduling is attempted again.

Scheduling for CGRAs is quite different from scheduling for general VLIW architectures due to the different hardware characteristics. Factors that complicate CGRA scheduling include:

Explicit routing. In a VLIW architecture, routing from producer to consumer is implicitly guaranteed by storing intermediate values in a multi-ported, centralized register file. However, in a CGRA, interconnect is much more sparse and values must be explicitly routed using FUs, local register files, and mesh connections.

Intelligent routing. FUs are used for both computation and routing; thus, scheduling can easily fail if poor routing choices are made. Furthermore, the scheduler must not only generate a valid schedule, but also minimize the routing resources used so that more FUs are available for computation.

Heterogeneous nodes. All nodes can perform addition and logical operations, but "expensive" operations such as multiplies, loads, and stores may only be supported by a subset of nodes. In such an architecture, it is important to avoid scheduling inexpensive operations on expensive nodes, because this limits the scheduling flexibility of the expensive operations.

Modulo constraint. Resources are used in a periodic fashion, since the loop kernel repeats every II cycles. Thus, unlike in acyclic scheduling, it is not possible to guarantee routability by extending the schedule, and scheduling can easily fail due to the previously scheduled operations.

To illustrate the complexities of CGRA modulo scheduling, Figure 2.2(a) shows

the dataflow graph (DFG) for the dominant loop from one of our benchmark applications, `fsed`, an image halftoning algorithm. Memory operations are shaded dark grey. The DFG is being scheduled onto a 4×4 CGRA, similar to the one shown in Figure 2.1, with II=4. The partial schedule is shown in Figure 2.2(b). schedule is shown. Bold numbers are computation operations; other numbers followed by 'r' (e.g. '8r') are routing operations for the corresponding computation operations; and, Xs represent slots that are occupied due to the modulo constraint. 'reg' nodes indicate live-in values that are stored in the central RF. All operations above operation 23 (light grey) in the DFG have been scheduled at this point.

There are several points to observe. First, only FUs 1, 2, 9, and 10 support memory operations, thus all of the memory operations must be scheduled on those FUs. Next, observe how values are routed to operation 23, which is considered for execution on FU 10 at time 17, and has two producers: 21 and 22. Figure 2.2(c) shows the possible routes of the operands from two producers. One possible way to route the operand from 21 to 23 is through FU 9. The operand is first routed diagonally from FU 4 to FU 9 via a shared register fie, then it is routed to the neighboring FU 10 via the mesh connection. However, taking this option leaves only two memory slots for the unscheduled memory operations (27 and 28). Therefore, the operand of 21 is routed through FU 5 rather than through FU 9. Similarly, the operand of 22 is routed directly from FU 15 to FU 10 rather than through FU 11. The value is stored in a rotating register file for 6 cycles and is read out by 23 at time 17. The challenge here is how to guarantee the availability of storage in the register file. The available

storage must be carefully considered during scheduling as simply pushing register allocation to after scheduling can result in costly spilling and may require complete rescheduling of the loop. It can be seen that routing is complex, and various resources including FUs, registers, register file ports, and connection links must be modeled by the compiler to properly orchestrate the flow of values from producers to consumers. Further, this routing adds latency to the schedule: operation 23 has an earliest start time of 11, but is actually scheduled at time 17.

CHAPTER 3

Modulo Graph Embedding

3.1 Introduction

In this chapter, we propose a modulo scheduling technique for CGRA architectures that leverages graph embedding commonly used in graph layout and visualization [33], referred to as *modulo graph embedding*. Graph embedding is a technique in graph theory in which a guest graph is mapped onto a host graph. With CGRAs, scheduling is reduced to placing operations of a loop body on a three dimensional grid. The three dimensions consist of the FU array that comprises two dimensions and the time slots of a modulo scheduled loop that form the third dimension.

Modulo scheduling is performed by considering groups of equal height operations from the top of the dataflow graph (DFG) to the bottom. The three dimensional scheduling grid is filled in a skewed manner by restricting the subset of FUs and time slots available for each group of operations. This stylization increases routability of operands and can dynamically adapt to different shape DFGs. A discrete cost

function between pairs of DFG nodes is designed and the placement algorithm tries to reduce this cost function. The cost function consists of different components: routing cost, which ensures that producers and consumers are placed close to one another; affinity cost, which ensures that operations with common consumers are placed close together; and, position cost, which ensures that operations are left-justified on the set of eligible resources. Left justification ensures operations are tightly packed and enables operand routing to subsequent operations using the righthand portion of the array.

The central advantages of modulo graph embedding are summarized as follows:

- It scales well with respect to number of operations in the DFG and thus is capable of handling large loop bodies.
- It handles a wide variety of CGRA configurations, including sparse interconnect and fully distributed register files.
- It is a systematic technique that assigns operations to the nodes in a CGRA and thus convergence to a solution is faster along with producing higher quality schedules.

3.2 Modulo Graph Embedding

This section describes modulo graph embedding, our approach to modulo scheduling for CGRAs. We break the description down into two parts: Section 3.2.1 presents the important concepts of the approach in isolation, and Section 3.2.2 brings every-

thing together to discuss the complete scheduling algorithm.

3.2.1 General Concepts

3.2.1.1 Resource and Connectivity Management

During instruction scheduling, a reservation table is maintained to keep track of which resources are used in each time slot. As resources are repeatedly used every II cycles by successive iterations of the loop, the modulo scheduler maintains a Modulo Reservation Table (MRT) which has only II time slots [48]. Considering that the scheduler for a CGRA must perform routing of values as well as placement of operations, routing information should be recorded by the scheduler. This routing information can be included in the reservation table because FUs are used both for computation and routing. Management of the interconnect network is not necessary as all of the connections are dedicated point-to-point connections, meaning that no congestion can occur in the network.

For resource management, the concept of the Modulo Routing Resource Graph (MRRG) from the DRESC compiler framework [38] is used. The MRRG is a graphical representation of the scheduling space where nodes represent routing resources and edges describe the connectivity of those resources. Scheduling in the CGRA becomes a problem of placement and routing of each operation on the MRRG.

The original MRRG has a detailed description of the CGRA [38]. MRRG nodes are created for each port on the FUs and register files, in addition to the MRRG nodes for the FUs and register files themselves. We take a simplified approach to

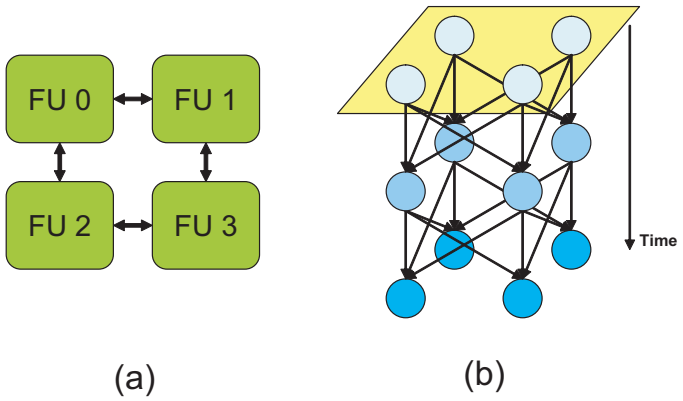


Figure 3.1: Modeling resources in a CGRA: (a) example 2x2 CGRA, (b) resource management model for 2x2 CGRA with II=3.

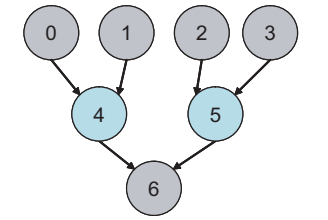
model the CGRA. A single node is created for each FU and register file. Since port information for FUs can be easily discovered by analyzing the resulting schedule along with the instruction format, it is not necessary to create nodes for the individual FU ports. Port information for register files can also be discovered in the same way, but two additional nodes are used to limit the number of read/write accesses to register files. Our resource management model can be considered as a distributed MRT with connectivity information. Each node represents either an FU or register file and is equipped with a MRT to keep track of the resource usage.

Figure 3.1(b) shows our resource management model constructed for the 2x2 CGRA in Figure 3.1(a) with II = 3. Nodes for register files and wrap-around edges were omitted for simplicity. Each of the four nodes in the CGRA has a 3-entry MRT, and each edge specifies that a value can be routed from the source to the destination resource. When an operation is placed on an FU, the MRT in the corresponding node is marked as occupied at the schedule time. If there are any placed producers or consumers, a valid route is discovered by traversing nodes along the edges.

3.2.1.2 Register Assignment and Allocation

With modulo scheduling, the number of registers required by a loop is not known before scheduling. In addition to conventional register allocation constraints, it may be necessary to keep multiple copies of registers depending on how many iterations separate the first producer and last consumer. This can cause a problem for the small, distributed register files in CGRAs as the number of total registers required at a single FU can exceed the local register file capacity. The available storage must be carefully considered during scheduling as simply pushing register allocation to after scheduling can result in costly spilling and may require complete rescheduling of the loop.

Our approach is to perform a simple register allocation and assignment during modulo scheduling. The modulo constraint that is enforced for FUs is also enforced for registers, i.e., there is an MRT kept for the each register file. A register value can stay in the same register up to II cycles, but the value will be overwritten by the same instruction in the next iteration II cycles later. When a register value is live for longer than II cycles, it has to be explicitly routed to another register file (or to another register in the same file). Specific entries in the register file are allocated for each virtual register using a simple greedy algorithm. While this approach may seem overly simplistic, it effectively guides the scheduler to distribute register usage across the CGRA.



(a)

(b)

Time	FU 0	FU 1	FU 2	FU 3
0	Op 0	Op 1	Op 2	Op 3
1	Op 4			Op 5
2		route	route	
3		Op 6		

(c)

Time	FU 0	FU 1	FU 2	FU 3
0	Op 0	Op 1	Op 2	Op 3
1		Op 4	Op 5	
2		Op 6		
3				

(d)

Figure 3.2: Example showing the placement of producers affects the routing cost of consumers: (a) DFG for loop, (b) target architecture which is a 1x4 CGRA, (c) poor schedule that results in an extra cycle for routing values to Op 6, and (d) good schedule that results in no additional routing.

3.2.1.3 Height-based Scheduling

The problem of modulo scheduling for a CGRA can be viewed as mapping applications onto the 3-D space consisting of the FU array stacked up II times. With this finite scheduling space, minimizing the *routing cost* is a critical issue in scheduling, as fewer resources being used for routing leads to more resources being available for computation. Routing cost is defined as the number of FUs being used for routing (passing data from one node to another) rather than computation. This cost depends on the positions of producer and consumer operations in the CGRA due to the sparse interconnect network. This requires the scheduler to be cognizant of producer and consumer relations so that they can be placed close to each other.

Figure 3.2 shows how the placement of operations impacts the routing cost of their consumers. Figure 3.2(a) is an example DFG and Figure 3.2(b) is a hypothetical architecture with sparse interconnect where FUs are allowed to communicate only

with adjacent FUs. Figures 3.2(c) and Figure 3.2(d) show two different schedules, both minimizing the routing cost for operations 4 and 5. When operation 6 is placed, the minimal routing cost is affected by the positions of its two producers (operations 4 and 5). This suggests that the scheduler must proactively choose placements to reduce routing costs.

To effectively manage routing costs, we employ two complementary techniques: height-based scheduling and the affinity-based placement which is discussed in the next section. Height-based scheduling is a common heuristic used in list scheduling where operations are scheduled in the order of dependence height. Operations with greater height are scheduled first, followed by operations with lower height. But, for operations with the same height, a CGRA scheduler cannot process them individually because placement of one operation has cost implications on the placement of others. Careless placement of one operation might increase the routing cost of other operations, or even make it impossible to place by blocking all of its routing possibilities. Therefore, operations with the same height are considered together to achieve an optimal placement rather than being scheduled separately. Possible schedule slots (resource/time pairs) are identified for each operation, and a combination of schedule slots (called a *layout*) that minimizes the total routing cost is selected.

3.2.1.4 Affinity Graph

Routing cost is difficult to minimize during scheduling because the true cost is not known until all producer-consumer pairs are placed. With height-based scheduling,

consumers are generally placed after the producers (except for operations on a recurrence cycle). Therefore, routing cost associated with just the producers is considered when an operation is placed. Even though the routing cost associated with consumers cannot be measured at the time that the producers are scheduled, it is desirable to account for these consumers in some way to avoid making greedy decisions. Ideally, operations with a common consumer should be placed close to each other so that the routing cost can be minimized later.

A measure of affinity is utilized to perform more intelligent scheduling by using information about common consumers. The affinity between a pair of operations with the same height is a measure of how close their common consumer is in the DFG. Operations with an immediate common consumer have the highest affinity between them, while operations without a common consumer have zero affinity. Operations with indirect common consumers have moderate affinities that decrease based on the distance to the common consumer. The goal is to place operations with high affinity close together to minimize the routing cost of the common consumers.

For each pair of operations, the affinity is calculated by looking at their common consumers. An affinity graph is then constructed that consists of nodes representing operations and edges representing affinity between operations. An example of the affinity graph is shown in Figure 3.3. The affinity graph is constructed for the operations in the first row of the DFG in Figure 3.3(a). Figure 3.3(c) is the resulting affinity graph where solid edges represent high affinity between operations (a value of 2 in the example) and dotted edges represent low affinity between operations (a

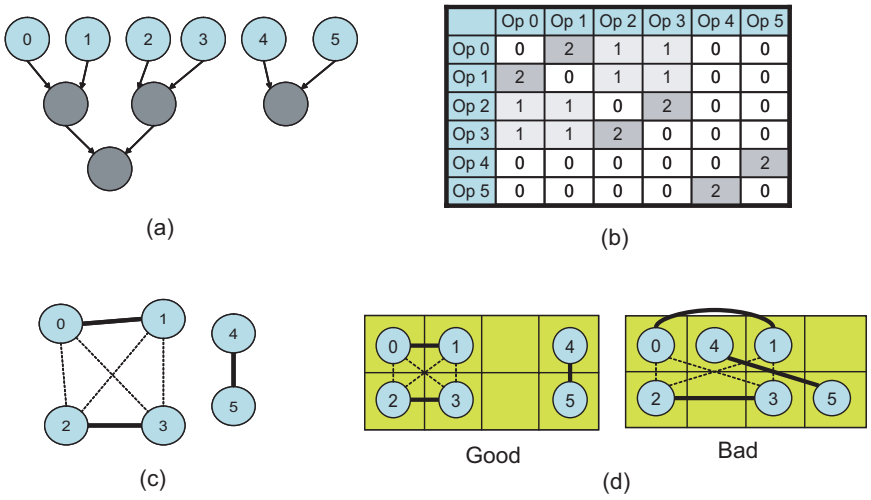


Figure 3.3: Example affinity graph: (a) DFG for loop, (b) calculated affinities between each pair of operations, (c) affinity graph, and (d) possible operation assignments to a 2x4 CGRA.

value of 1). Pairs of operations without edges have an affinity of zero.

For each pair of operations A and B with the same height, the affinity value is calculated using the following equation. Only the common consumers within the range of *max_dist* are considered in the calculation of the affinity value. The variable *num_cons(A,B,d)* denotes number of common consumers of A and B whose distance from A and B in the dataflow graph equals d.

$$affinity(A,B) = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times num\_cons(A,B,d) \tag{3.1}$$

When scheduling operations, the scheduler attempts to place operations close together according to their affinity. Two alternate schedules for the operations are shown in Figure 3.3(d) that illustrate the use of affinity to eliminate explicit routing operations by performing more intelligent assignment of operations to nodes in the CGRA. The schedule on the left is better because operations with affinity edges are

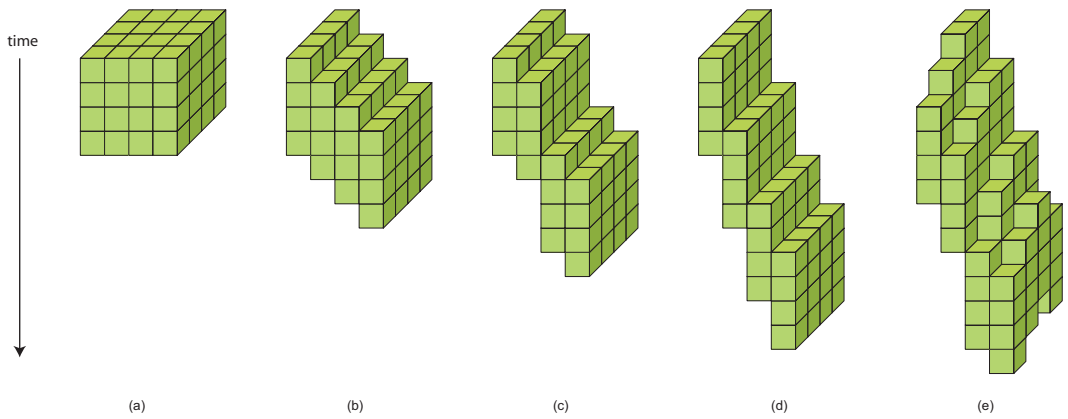


Figure 3.4: CGRA scheduling spaces: (a) normal scheduling space, (b) skewed scheduling space, (c,d,e) variations of skewed scheduling space.

placed closer on the array.

3.2.1.5 Graph Embedding

In this work, we leverage graph embedding that is commonly used in graph layout and visualization. Graph embedding is a particular drawing of a graph onto a target space (usually a planar space). Drawing large graphs "nicely" is not an easy task. Here, a nice graph usually refers to non-crossing edges and a regular distribution of nodes. The spring embedder model [10] is a well known heuristic approach to graph embedding. It simulates a mechanical model of rings attached with springs. Each ring represents a node in the graph and each spring between two rings represents forces that attract or repel the nodes in the graph. The spring embedder is a suitable model for our scheduling. Each weighted edge in the affinity graph can be thought of as a spring that attracts two nodes in the graph. An edge with high affinity will attract two operations so that they are placed on the same or nearby resources.

A large amount of research has been conducted for effective graph drawing using

the spring model. Kamada and Kawai proposed an iterative algorithm that calculates attractive and repulsive forces for each node and gradually moves the nodes with respect to the calculated forces [23]. Davidson and Harel employed a simulated annealing method that improves the cost of the graph based on the spring model [8]. However, most works do not fit into our scheduling problem as they assume a continuous space rather than the discrete, finite 3-D scheduling space. Graph embedding onto a grid-based space is well studied in the area of VLSI cell layout, known as force-directed placement. These works have somewhat different objectives, such as minimum edge bends. Li and Kurata proposed a grid layout algorithm of biochemical networks [33]. It uses simulated annealing for embedding complicated biochemical graphs onto the grid space. We found this solution best suited for our problem as its target space is discrete and the objective is placing nodes with edges close together.

Compared to the target graphs of typical graph embedding algorithms, our affinity graph has quite a small number of nodes. This is because we are not scheduling the whole application at one time. Instead, graph embedding is performed for each height level of the DFG and it is unusual for more than 20 operations to have the same height. Also, the search space is limited by pre-placed operations because pre-placed producers limit the possible slots of their consumers due to the sparse interconnect. For the search space that is sufficiently constrained, a simple exhaustive search can find an optimal layout of operations quickly. Li and Kurata's algorithm is employed only for large search spaces where the exhaustive search cannot finish in a reasonable time.

3.2.1.6 Skewed Scheduling Space

One of the difficult challenges of scheduling for CGRAs is ensuring that the necessary routing can take place as the CGRA is filled up with more operations. At the start of scheduling, the CGRA is empty, thus routing is not difficult. But, as scheduling proceeds, the scheduler can easily back itself into a corner and get stuck where the necessary routing cannot be performed. The affinity heuristic tries to minimize the overall number of resources used for routing, but this is not enough. When schedule times get larger than II, difficulties often result due to pre-placed operations (repeated resource use by the same operation every II cycles) and already-placed producers.

The conventional approach used in modulo scheduling is backtracking, where one or more operations are unscheduled to allow the current operation to successfully schedule [48]. However, backtracking for CGRAs is much more complicated. First, placing an operation usually requires more than one resource as routing is involved. This means that many operations can be unscheduled to overcome a routing failure. Moreover, re-scheduling operations requires both routing to its consumers as well as from its producers. It's difficult for the scheduler to make forward progress with backtracking.

A different approach is to prevent routing failures in advance. In general, routing failures to a consumer can be avoided if all the resources are free in time slots later than a producer's time slot. This is why the acyclic scheduling does not suffer from routing failures as it has an infinite scheduling space. Likewise, modulo scheduling does not suffer from routing failures within an II cycle window. Further, most applications

don't have enough parallelism that requires all the CGRA FUs in one cycle. These two observations encourage clustering of the CGRA. With clustering, the FUs in the CGRA are partitioned into subsets. The scheduler can utilize one subset, or cluster, without any routing failures for II cycles. When the cluster is full, the scheduler can then use another cluster for the remaining operations, and so on.

Instead of partitioning the CGRA statically, our approach clusters the CGRA dynamically where the cluster boundaries are not strictly defined. The clusters are formed in a left-to-right manner on the array. The scheduler gives priority to the leftmost available FUs. But when the application parallelism is high, the cluster is dynamically enlarged by being forced to assign operations to lower priority FUs on the right. The scheduler utilizes a *position cost* to accomplish dynamic partitioning. When an operation is considered on an FU, its position cost is computed. The position cost is determined by the column in which the FU lies. Low cost is assigned to the leftmost available FUs, while higher cost is assigned to the FUs that lie further to the right.

When a partition of FUs to the left becomes full, values must be routed to FUs to the right. To guarantee this is possible, the concept of a skewed scheduling space is introduced as shown in Figure 3.4(b). Unlike the traditional scheduling space (see Figure 3.4(a)) where all the slots are available at the given schedule time, the start times of FUs are restricted such that they stagger down the right side of the CGRA. Since each FU is only available later than the FU on its left, the last schedule slot is always available to the output value of the last schedule slot of its left FU.

When no operation is placed on an FU at the original start time, the start time increases, which slides down the scheduling space of the FU. When the scheduling space of an FU is lowered, scheduling spaces of FUs to its right are also lowered to guarantee the routability. Therefore, the skewed scheduling space dynamically changes as operations are placed in the CGRA. As the operations at the same height are considered together to get an optimal layout, the parallelism in the application at the given height determines the shape of the scheduling space. Some applications may not even require all four FUs in one column. In this case, the position cost is augmented with the row cost and the FUs in the upper rows are utilized first. Figure 3.4(c), (d) and (e) show several other possible skewed scheduling spaces.

Assignment of operations to the skewed scheduling space works well for forward dependence patterns, but difficulties arise with recurrence cycles. Recurrence cycles contain a communication pattern where a producer is scheduled after its consumer. Thus, the producer will be likely to be placed on the right of its consumer and routing becomes difficult since most schedule slots on the left are already utilized. To address this routing problem, our approach is to reserve in advance slots for such cycles when the consumer is placed. When a producer is placed later, it can use this reserved route. Again, we take the preventative approach to avoiding routing problems rather than solving them when they occur.

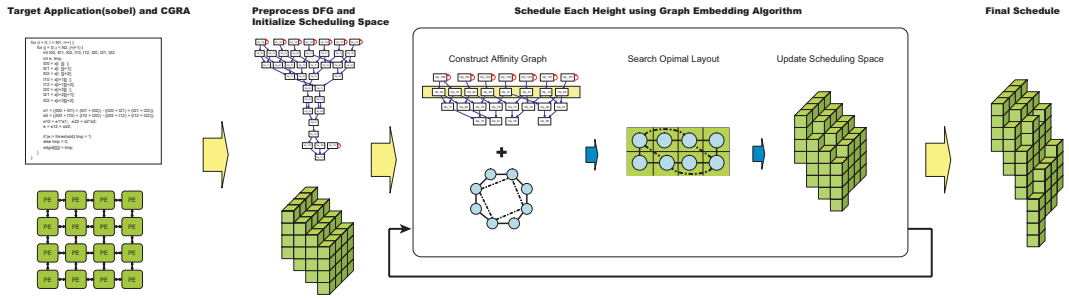


Figure 3.5: Overview of the CGRA scheduling system: input is the assembly code for the loop body and a description of the CGRA; preprocessing analyzes the loop to compute heights and skew the available scheduling cycles for the FUs; the graph is iteratively scheduled at successive dependence height levels by constructing the affinity graph and performing modulo graph embedding of the affinity graph on the CGRA.

3.2.2 Implementation

Figure 3.5 presents an overview of our system. It takes the target loop body and description of the CGRA as input. The scheduling process consists of an initial preprocessing step to analyze the DFG and set up the skewed scheduling space. This is followed by the main scheduling loop that iterates over each level of the DFG to find a placement of all the operations at a particular height using modulo graph embedding.

3.2.2.1 Preprocessing

The target application is first preprocessed to calculate the heights of all operations based on the distance from the terminating operation (e.g., the loop back branch). The height of an operation determines when it is considered for scheduling and the height difference between producer and consumer is a rough estimation of the live range of the intermediate values.

The scheduling space is skewed by assigning different start times to FUs. The same start time is assigned to all FUs in one column. Starting from zero for the first column on the left, the start time staggers downward with each increasing column number.

3.2.2.2 Scheduling Process

Scheduling proceeds through successive dependence height levels of the DFG considering all operations at a level simultaneously. Scheduling is converted into a graph embedding problem that maps the affinity graph onto the skewed scheduling space. Our modulo scheduler is implemented based on Li & Kurata's grid layout algorithm [33]. In the remainder of this section, we review basic ideas behind grid layout and describe our modified algorithm.

Grid Layout: Grid layout treats graph embedding as an optimization problem. A discrete cost function is defined for each pair of nodes based on the topological relation and the geometric positions of the nodes in the layout. Namely, high cost is given when two nodes connected by an edge are placed far apart and low cost is given when they are placed close together. The cost of a layout is given as a summation of costs for all node pairs. Simulated annealing is employed to find the layout with the lowest cost.

Modulo Graph Embedding: Unlike the original problem in grid layout, our problem has more constraints and costs to consider. Specifically, scheduling operations at each height has the following objectives:

- Place operations with a common consumer close to each other
- Minimize the routing cost for values from producers
- Ensure the routability of values to consumers

To achieve the objectives above, the scheduling concepts in Section 3.2.1 are realized in a cost function composed of three terms: routing cost, affinity cost, and position cost. They are calculated for operations by the following equations. where A and B are operations to be placed and $affinity(A, B)$ is given by Equation 3.1 from Section 3.2.1.4:

$$routing\_cost(A) = \#\ FUs\ used\ for\ routing\ values\ from\ producers\ to\ A \tag{3.2}$$

$$affinity\_cost(A, B) = distance(FU(A), FU(B)) \times\ affinity(A, B) \tag{3.3}$$

$$position\_cost(A) = column\ \#\ of\ FU(A) \times BASE\_COST \tag{3.4}$$

$$layout\_cost = \sum_{A \in ops} \Big(routing\_cost(A) + position\_cost(A) \Big) + \sum_{A,B \in ops} affinity\_cost(A, B) \tag{3.5}$$

Grid layout employs a simulated annealing search to find an optimal layout of operations at each level by minimizing *layout_cost*. While the original grid layout maps a graph onto a 2-D plane, our target space is 3-D scheduling space which

can have an infinite search space with varying schedule times. Therefore, we limit the search space by placing operations only in slots that minimize routing cost, called *primary slots*. Primary slots are identified before placing any operations. Even though each individual primary slot has the same routing cost, the total routing cost of a layout might vary because routing for one operation might block routing for another. Therefore, the routing cost is still considered in the cost function.

Once primary slots are identified, the size of search space is the product of the size of each operation's primary slots. Sometimes the search space can be quite small since pre-placed producers limit the placement of consumers. For small search spaces, exhaustive search is employed rather than using the grid layout. A flow chart of the scheduling process is presented in Figure 3.6.

The grid layout process begins with an initial layout obtained by randomly placing operations in one of their primary slots. Beginning with the initial layout, the scheduler enters a loop where the cost of current layout is iteratively reduced using simulated annealing. First, operations are randomly moved or swapped with other operations to generate a neighbor of the current layout. The neighbor layout is then locally optimized. Local optimization greedily performs moving or swapping operations whenever the cost is reduced, and these actions are repeated until no further improvement can be achieved. The locally minimized layout is evaluated for acceptance as an optimal layout. At some points, an uphill movement is taken to escape from a local minimum. After the optimal layout is discovered, the scheduling space is adjusted to reflect the chosen placement of operations at the current height and the

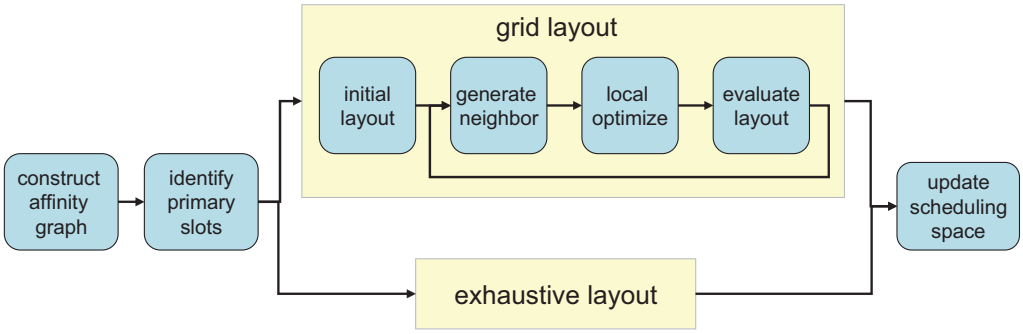


Figure 3.6: Scheduling process for operations at each successive dependence height.

scheduler proceeds to the next height.

3.2.2.3 Scheduling Example

The process of scheduling each height of the application onto the skewed scheduling space is illustrated in Figure 3.7 with sobel, an image edge detection algorithm. The II in this example is 4. Due to space limitations, scheduling of operations for the first three heights is presented. Figure 3.7(a) shows the DFG of sobel and the target 4x4 CGRA. Scheduling for the selected heights is illustrated in Figure 3.7(b).

For each height, the affinity graph for the operations is shown at the top with solid edges representing high affinity and dotted edges representing low affinity. The table in the middle, where FUs are represented horizontally and time vertically, shows the resulting layout of operations. Note that the FUs in the left two columns only appear in the table since the other FUs are not used in this example. Each entry in the table represents a schedule slot and shaded entries constitute the scheduling space of the CGRA (also shown in 3-D graph at the bottom). Since FUs are repeatedly used every II cycles, entries are marked with X's when they are occupied by previously

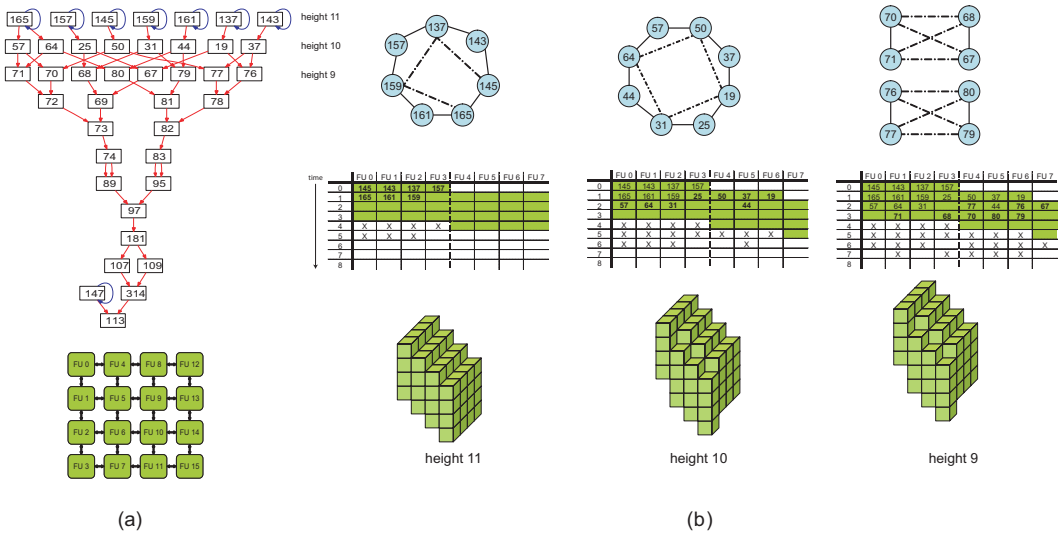


Figure 3.7: Example of modulo graph embedding: (a) DFG of sobel and target CGRA, (b) scheduling results of first three heights.

scheduled operations.

At height 11, operations are placed only in the first column due to the limit of the skewed scheduling space. Also, operations with high affinity represented in solid edges are placed in adjacent schedule slots. For example, 145 is placed adjacent to 165 and 143 due to its high affinity with these operations. Conversely, 145 is placed apart from 157 because there is no affinity between 145 and 157. Note that routing cost is not considered at this height since there are no producers placed.

At height 10, all the costs, including routing cost, are considered. As the operations at height 11 were intelligently placed based on the affinity, the scheduler places operations at height 10 without using any resources for routing. FUs in the second column are also utilized to support the parallelism in the application. Since no operation is placed on FU 7 at its original start time of 1, FU 7's start time is increased by 1 and its scheduling space is slid down. This also implies that the scheduling spaces

Design Name	#RFs	#FUs per RF	#Regs	#Read ports	#Write ports
Dedicated RF	16	1	4	2	1
Shared RF	4	4	12	8	4
Central RF	16 local	1	4	2	1
	1 central	16	32	8	4

Table 3.1: Register file configurations for three CGRA designs used for evaluation.

of FU 11 and FU 15 are slid down to guarantee routability.

Operations at height 9 are scheduled similarly to those at height 10, again accounting for all costs. Note that the unoccupied slots in the second column at time 0 can be utilized II cycles later when output values of operations placed in the first column cannot otherwise be routed due to the modulo constraint. For example, the output values of operations 68 and 71 at height 9 can be routed using schedule slots of the second column at time 4.

The final scheduling space of sobel is shown on the righthand side of Figure 3.5.

3.3 Experimental Results

3.3.1 Experimental Setup

CGRAs can be characterized by many parameters. To evaluate the performance of our scheduler, three designs were tested. All three designs have the same architectural parameters except for their register file configuration. All are 4x4 homogeneous CGRAs connected with a mesh network, with operation latencies of the ARM926 (e.g., 3 cycles for multiply, 2 cycles for load/store, and 1 cycle for simple arithmetic).

Table 3.1 shows the register file configurations for the three designs. These designs are the same as those pictured in Figure 2.1. The central RF design is the same as the dedicated RF design except that it has an additional central register file shared by all 16 FUs.

To evaluate the modulo graph embedding scheduler, twelve loop kernels are taken from various application domains: signal processing (fft, fir, iir, viterbi), encryption (blowfish), image processing (dct, fsed, sharp, sobel), network processing (channel), and video compression (idct, dequant). Only the innermost loop is considered for modulo scheduling for multidimensional loop nests.

3.3.2 Evaluation of Affinity Graph Heuristic

The main objective of the affinity graph heuristic is to minimize total routing cost by using common consumer information. In modulo scheduling, total routing cost is affected by other factors, such as recurrence cycles and the modulo constraint. In acyclic scheduling, we can exclude the influence of the modulo constraint as we can always find time slots where resources are available by increasing schedule time. Thus, we evaluated the performance of our affinity graph heuristic in the domain of acyclic scheduling; only loop kernels without a constraining recurrence cycle were tested. The dedicated RF design in Figure 2.1(a) was used as the target architecture because it has the sparsest interconnect and therefore is the most affected by the placement heuristic.

Two cost models were compared to evaluate the affinity graph heuristic. One is

	With Affinity		Without Affinity	
Bench	SchedLen	RouteFUs	SchedLen	RouteFUs
blowfish	32	4	34	14
channel	16	31	17	52
dct	15	24	19	53
fft	12	22	14	35
fir	8	3	9	5
fsed	11	2	12	6
sharp	21	19	25	23
sobel	11	2	13	12
viterbi	20	52	20	57

Table 3.2: Effectiveness of the affinity heuristic using acyclic scheduling.

implemented with both routing cost and affinity cost. (Position cost is not considered as the scheduling space is not skewed in this experiment.) The other model does not consider affinity in its cost function, and only tries to minimize routing cost when operations are placed. The quality of the schedule is measured by schedule length and number of FUs used for routing. The second and third columns of Table 3.2 show the quality of the schedules obtained with the affinity graph heuristic, while the fourth and fifth columns show the result without it. For all of the benchmarks, the affinity graph heuristic works well in reducing both the schedule length and number of FUs used for routing. Clearly, guiding placement using downstream information about consumers is important for CGRAs.

3.3.3 Evaluation of Modulo Scheduler

Two experiments are performed to evaluate the effectiveness of modulo graph embedding. First, a detailed analysis using the dedicated RF CGRA is presented. Then, we compare the most important parameter in scheduling for CGRAs, utilization

or fraction of the cycles the FUs in the array perform useful computation, for all three register file configurations.

Scheduling results for the dedicated RF design are shown in Table 3.3. The second and third columns show the number of operations and the number of communication edges in the applications, respectively. These numbers roughly describe the communication patterns of the application. The fourth column shows the effective number of operations; for this metric, multi-cycle operations are counted multiple times according to their latency. Even if these operations can be pipelined with other operations of the same opcode, they increase the difficulty of the scheduling problem as the write-back resources of the node must be used at operation completion. The fifth and sixth columns in the table contain the minimum IIs for each benchmark. The maximum utilization that can be achieved is limited by these IIs.

The next two columns show the II and schedule length achieved by the modulo graph embedding scheduler. Unlike acyclic scheduling, II is a better measurement of performance than schedule length. The achieved II translates into the utilization of FUs shown in the "util" column. The utilization is calculated by dividing the number of schedule slots used for computation by the total number of slots which equals to (# FUs x II). "Total util," shown in the next column, takes into account the FUs being used for routing. All benchmarks show a utilization of greater than 43%. Fir has the lowest utilization, but more than half of the operations are multi-cycle operations, including four multiply operations. Iir also has low utilization, but its RecMII is 4, which limits the achievable utilization. Fft and sharp have relatively low utilization

because they have a high number of one-to-many communication patterns. Routing cost increases with the number of consumers, as the value has to be individually routed to each consumer.

On average, the scheduler achieves 56% utilization for all benchmarks, with individual values ranging from 44% to 69%. This average utilization is similar to that achieved by the DRESC compiler, even though the target architecture of DRESC had a central register file and denser network connectivity. This shows that the modulo graph embedding scheduler is able to achieve quality solutions for significantly lower cost CGRAs.

The modulo scheduler runtimes (last column of Table 3.3) are reasonably fast, as all benchmarks are scheduled within 5 seconds on a 3 GHz Pentium-4 machine with 1G of RAM. This is because the search space is limited to operations in the DFG with the same height; thus, fewer than 20 operations are generally considered at a time. Also, scheduling does not employ backtracking, nor random movement of operations. Rather, systematic heuristics derived from the DFG guide the scheduler.

The impact of different register file configurations was evaluated by scheduling the same set of benchmarks on the other two CGRA designs (shared RF and central RF). The utilizations of the resulting schedules are shown in Figure 3.8. For all the benchmarks, the smallest II was achieved for the central RF, showing highest utilization in the graph except for blowfish and dequant. Blowfish and dequant were scheduled at the same II for shared RF and central RF, but utilizations are slightly higher for shared RF because different number of multi-cycle operations are pipelined.

Bench	ops	edges	eff ops	Res	Rec	II	len	util	total util	time
blowfish	85	99	107	6	1	10	46	0.650	0.800	2
channel	121	180	187	8	1	16	30	0.617	0.878	4
dct	114	150	142	8	1	13	30	0.625	0.836	5
fft	52	78	78	4	1	9	25	0.479	0.798	1
fir	23	30	41	2	1	4	14	0.437	0.734	1
fsed	38	48	45	3	1	4	16	0.687	0.906	1
iir	23	33	32	2	4	4	15	0.453	0.625	1
sharp	56	95	72	4	4	9	37	0.486	0.854	1
sobel	39	59	52	3	1	5	17	0.612	0.675	1
viterbi	104	181	124	7	1	14	30	0.526	0.843	1
idct	119	200	215	8	2	18	41	0.576	0.833	5
dequant	84	141	106	6	3	10	25	0.600	0.806	1

Table 3.3: Modulo graph embedding results for the dedicated register file CGRA.

Since the central RF is connected to all 16 FUs, each FU can communicate with any other FU in 1 cycle, subject only to the availability of ports and register entries. With these additional routing resources, more FUs can be used for computation. The shared RF design achieves higher utilization than the dedicated RF design, as each register file can be used as a routing resource among the four FUs that share it. This result shows how increasing register file sharing can improve the quality of the schedule, giving more routing options to the scheduler.

3.4 Related Work

3.4.1 Architectures

Many CGRA-like designs have been proposed in the literature. The designs have different scalability, performance, and compilability characteristics as discussed in Section 2.1. The ADRES architecture [38] is an example of an 8x8 mesh of process-

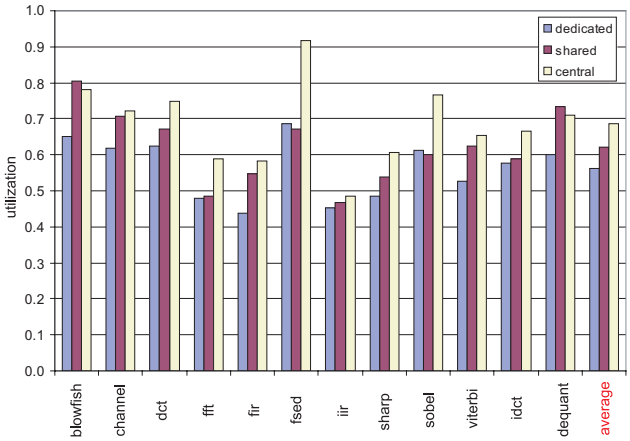


Figure 3.8: Comparison of utilization rates for three register file configurations.

ing elements with both individual and central register files. MorphoSys [36] is another example of an 8x8 grid with a more sophisticated interconnect network; each node contains an ALU and a small local register file. In the RAW architecture [52], each node is actually a MIPS processor, including memory, registers, and a processor pipeline. In addition, there are both dynamic and static routing networks. PipeRench [18] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining. RaPiD [13] consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

3.4.2 Compilation Techniques

Many techniques have been proposed for compiling to CGRAs. Lee et al. [29] propose a compilation approach for a generic CGRA. They generate 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. Our work proposes a generic scheduling strategy, and memory sharing and other such optimizations can be integrated into our system as a preprocessing step. Convergent scheduling is proposed as a generic framework for instruction scheduling on spatial architectures [31]. Their framework comprises a series of heuristics that address independent concerns like load balancing, communication minimization, etc. Whereas convergent scheduling focuses on ILP and proposes a scheduling method for acyclic regions of code, we focus on loop level parallelism. The work of Mei et al. [38] is closest to our work. They propose a modulo scheduling algorithm for CGRAs based on simulated annealing. Our approach differs significantly in that we apply systematic placement decisions and on a skewed scheduling space to achieve better convergence and faster compilation times.

Similar to CGRAs, clustered VLIW machines are also spatial architectures. Much work has been done towards compiling for clustered VLIW machines [15, 42, 50]. Although some of the concepts from these works can be adapted for CGRA compilation, they do not consider the issue of routing values through the sparse interconnection network, which is a crucial step. The measure of affinity used in our scheduler is similar to that used in Krishnamurthy's affinity-based clustering [26].

[57] employs similar concept of affinity to minimize communication penalty in the resource allocation phase. 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. However, this approach that decouples resource allocation from scheduling is not suitable in modulo scheduling. Since each resource can be utilized only II times, it is not always possible to find proper time slots for operations on their pre-assigned resources. In our affinity graph heuristic, resource allocation is considered jointly with time slot assignment.

3.5 Summary

This chapter proposes modulo graph embedding, an effective modulo scheduling technique for CGRAs. The sparse interconnect and distributed register files of the CGRA present difficult challenges to a compiler. Our approach leverages classic graph embedding to draw loop bodies onto a three dimensional graph representing the CGRA. We introduce two key concepts to generate high-quality solutions by reducing routing cost. First, an affinity graph heuristic analyzes producer/consumer relations to place operations with common consumers closely. Second, the scheduling space is skewed by restricting the assignable FUs and time slots available for each group of operations to enable dense packing of operations onto the array while still ensuring operand routing paths are available. Overall, modulo graph embedding achieves average compute utilization of 56–68% for three different register file configurations, including a CGRA with no shared register files. Prior approaches have only achieved such utilization rates in CGRAs augmented with multiported shared register files. Our scheduler also performs substantially faster than existing solutions since we limit the search space to operations at the same height and employ a systematic placement

based on the producer/consumer relations.

CHAPTER 4

Edge-centric Modulo Scheduling

4.1 Introduction

An effective compiler is essential for exploiting the abundance of computing resources available on a CGRA. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of a compiler. Traditional schedulers that just assign an FU and time slot to each operation are insufficient because they do not take routing into consideration. Scalar operand values must be explicitly routed between producing and consuming operations. Further, dedicated routing resources are not provided. Rather, an FU can serve either as a compute resource or as a routing resource at a given time. A compiler scheduler must manage the computation and flow of operands across the array to effectively map applications onto CGRAs.

To efficiently make use of the CGRA resources, modulo scheduling (or other software pipelining variations) of loops is generally used [48]. This provides the opportu-

nity to exploit both loop-level and instruction-level parallelism to efficiently make use of the CGRA resources. To deal with the complex topology and routing challenges, the DRESC (Dynamically Reconfigurable Embedded System Compiler) proposes a modulo scheduling algorithm based on simulated annealing [38]. It begins with a random placement of operations on the FUs, which may not be a valid modulo schedule. Operations are then moved between FUs until a valid schedule is achieved. The strength of simulated annealing is its ability to deal with both sparse connectivity and complex resource usage that are common in a CGRA. DRESC consistently achieves the leading performance results over other methods on a variety of CGRAs. However, the random movement of operations in the simulated annealing technique can result in a long convergence time for loops with modest numbers of operations. Also, the algorithm is ad-hoc in the sense that no information about the structure of the loop's dataflow graph is utilized in making scheduling decisions.

In this chapter, our goal is to develop a more systematic approach where compilation time is a first-class constraint. We initially chose to adapt iterative modulo scheduling to CGRAs because it both produces efficient results and offers short compilation times even for large loops [48]. The central changes were adapting the scheduler to understand the decentralized resources of a CGRA as well as performing routing of operands between producing and consuming operations. While this approach was successful at creating correct schedules, loop throughput was reduced by 10-50% in comparison to the simulated annealing method. An analysis of the resultant loops showed that *node-centric modulo scheduling* is a poor match for CGRAs. Traditional

schedulers are node-centric in that the focus is assigning operations (nodes) to FUs. The straightforward adaptation of this approach is operation assignment followed by operand routing to determine if the assignment is feasible. However, even with large numbers of free FUs, the scheduler inevitably fails due to the inability to route an operand. Further, backtracking is ineffective due to the complex interrelations between scheduler decisions.

The key insight from this experience was that a CGRA scheduler must consider routing efficiency as the primary objective. Selecting intelligent paths from producing to consuming FUs that do not block other operand paths is essential to achieving higher throughput schedules. Further, operation assignment can be viewed as a by-product of a successful route, thus no successive placement step is required. In essence, by getting an operand between two points, the necessary operations can be performed along the way for free. We refer to this technique as *edge-centric modulo scheduling*, or EMS. This paper presents the design, implementation, and evaluation of the EMS algorithm.

4.2 Core Concepts

Prior to describing the EMS algorithm, we describe several of the important concepts along with their rationale. These concepts are described in isolation (and hence will appear disconnected), but they are tied together in Section 4.3.

4.2.1 Integrated Placement and Routing

CGRA scheduling can be broken down into two tasks: placement of operations into computation slots (FU and time) and routing of operands. Previous techniques ([38], [45]) address the scheduling problem in a node-centric manner, meaning that the scheduler places operations first and then does the routing. When an operation is scheduled, it is placed in a slot where it can execute, and operands from other producers or consumers are then routed to the scheduled slot. However, scheduling failures usually occur during the routing phase because of the limited connectivity between resources. In this work, we propose an edge-centric approach where the scheduler primarily focuses on routing, and placement occurs during the routing process.

Node-centric Approach.

Node-centric approaches place operations in a way that minimizes a heuristic routing cost. The routing cost consists of various metrics that determine the quality of placement (e.g., the number of resources used for routing) [45]. The scheduler visits candidate slots one by one until it finds a solution. The operation is placed in each candidate slot, and edges to the placed producers and consumers are routed. Figure 4.1(b) shows how an optimal placement is found with this approach. A DFG containing two producers P1 and P2 and a shared consumer C is mapped onto the hypothetical 1×5 CGRA in Figure 4.1(a). For illustration purposes, we assume no register file in this architecture. P1 and P2 are already placed and the scheduler places the consumer C by visiting all the empty slots as shown in Figure 4.1. The slots with dotted circles are failed attempts where the scheduler could not route values from P1

or P2 due to resource conflicts. After visiting those slots, the scheduler successfully places C on FU 4 at time 4 (slots will be referred as (FU #, time) hereafter).

One can observe two inefficiencies with this approach. First, the scheduler makes unnecessary visits to empty slots (0,2), (0,3), and (0,4). This is because the scheduler places operations without routing information. The second inefficiency is that there are redundant routings made when the scheduler visits (2,1), (2,2), (2,3), (2,4), and (3,4). For example, when the scheduler visits slot (3,4), it already knows that there is a path P1→(2,1)→(2,2)→(2,3) since it was discovered when slot (2,3) was visited. These observations show that placement without routing information can lead to redundant routing calls, which increases compilation time. One can argue that a different visiting order can solve this problem (visiting slots in the same FU first). Even though this can work for this particular case, there is no general order that works for all the cases in the node-centric approach.

A node-centric approach can also lead to a poor solution because it does not consider routing information when placing an operation. Figure 4.1(d) shows a different example where P is already placed and the edge from P to C is about to be routed. Here, we assume that C can be placed in only two slots, (4,2) and (2,4). Note that slot (3,1) is the only remaining memory access slot, thus it is critical to avoid using this slot for routing if possible. Since the node-centric approach visits slot (4,2) before slot (2,4), it will simply choose the path to slot (4,2) in Figure 4.1(d), using the memory slot for routing. If any memory operation still needs to be scheduled, the II must be increased. Here, we are assuming that the node-centric approach visits slots

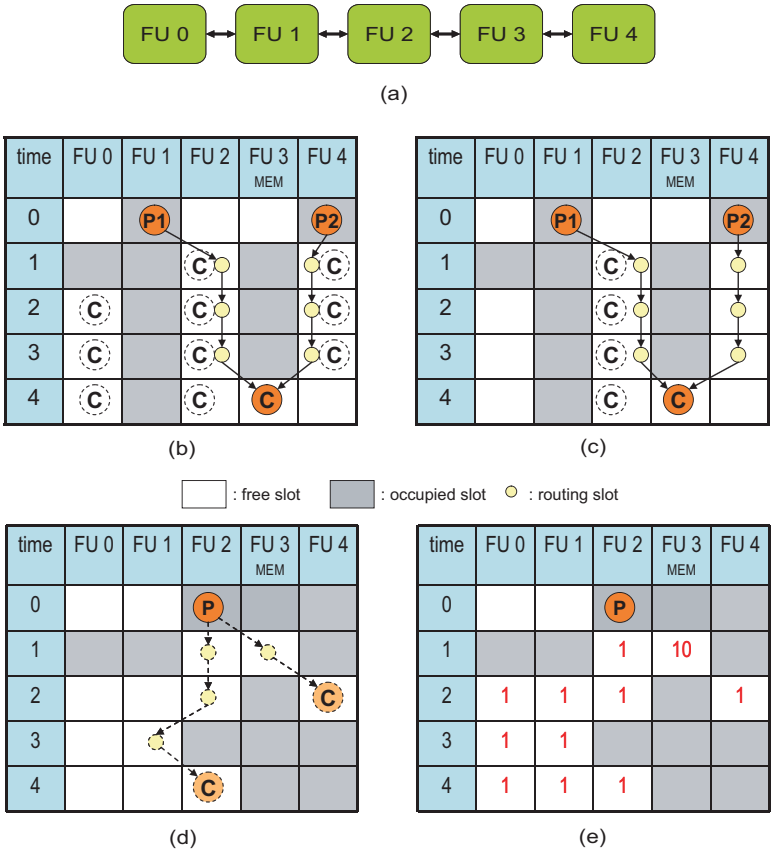


Figure 4.1: High level comparison of scheduling approaches: (a) 1x5 CGRA, (b) compile time example of node-centric, (c) compile time example of edge-centric, (d) performance example of node-centric, (e) performance example of edge-centric. Shaded boxes in the reservation tables indicate slots occupied by other operations.

in an increasing order of time. Although a different visiting order can give priority to slot (2,4) over slot (4,2), that particular order cannot be applied to general cases without routing information. In general, the node-centric approach needs to perform an exhaustive search of all the available slots to handle this problem.

Edge-centric Approach. In an edge-centric approach, the placement of an operation is integrated into the routing function, and the placement decision is deferred until routing information is discovered. When scheduling an operation, the scheduler

does not place the operation up front. Instead, it picks an edge from the operation's previously-placed producers or consumers and starts routing the edge. The router will search for an empty slot that can execute the target operation, rather than routing towards a placed operation. Once a compatible slot is found, the target operation is placed in the slot and the scheduler continues routing edges to other producers or consumers.

Figure 4.1(c) shows the same example of Figure 4.1(b), but the consumer is scheduled using an edge-centric approach. The scheduler begins with the edge from P1 to C, instead of scheduling operation C directly. When an empty slot is encountered, the scheduler temporarily places the target operation and checks if there are other edges connected to the consumer; if so, it recursively routes those edges. For example, when the router visits slot (2,1) in Figure 4.1(c), it temporarily places C there and recursively calls the router function to route the edge from P2 to C. When it fails to route the edge from P2 to C, routing resumes from slot (2,1), not from P1, and a solution is eventually found at slot (3,4). So, slots (2,1), (2,2), (2,3), (2,4), and (3,4) are all visited in one routing call. Compared to 11 routing calls made for the edge from P1 to C in Figure 4.1(b), only one routing call is required to find the same solution in the edge-centric approach. The number of routing calls for the edge from P2 to C is same for both approaches (5 calls), as the router is only called for that edge if the edge from P1 to C is routed successfully.

The second benefit of an edge-centric approach lies in the aspect of solution quality. In the example in Figure 4.1(d), it is desirable not to use slot (3,1) for routing. The

edge-centric approach avoids using the memory slot (3,1) for routing by assigning a higher cost to the slot as shown in Figure 4.1(e). Here, a cost of 10 was assigned to slot (3,1) and all the other slots were assigned a cost of 1. Then, the edge-centric approach will automatically find a path that avoids slot (3,1) by prioritizing the route path by cost. So, it successfully finds a path to slot (2,4) using the left path in Figure 4.1(d).

An edge-centric approach can perform faster and achieve a better result than a node-centric approach. However, it has a greedy nature in that it optimizes for a single edge at a time, and the solution can easily fall into local minimum. There is no search mechanism in the scheduler at the operation level and every decision made in each step is final. We address this problem by employing intelligent routing cost metrics explained in the next section.

4.2.2 Routing Cost Metrics

The routing function is the basic building block of the edge-centric scheduler, and every scheduling task, including placement, occurs in the routing function. The final schedule is formed by calling the routing function for each edge in the DFG.

It is important to achieve a good mapping for each individual edge. The routing function needs to have a global perspective of the entire mapping since individual decisions affect the routing of other edges. The order in which the router visits each scheduling slot is determined by a *routing cost* associated with each slot. Thus, it is crucial to develop a good routing cost function.

There are two main objectives when routing a single edge:

- Minimize the number of routing resources used, to leave more slots available for routing other edges.

- Proactively avoid routing failure: avoid using resources that will block future routes, and reserve computation slots for expensive operations.

4.2.2.1 Minimizing the Number of Routing Resources

Using the fewest routing resources is simple when considering a single edge. Each routing resource is assigned a statically-determined fixed cost, and the router will find a path that minimizes the total cost.

Typically, an operation is connected to multiple producers and consumers, so the router must consider the usage of routing resources when the other edges are routed as well. To address this issue, an *affinity cost* was proposed in previous work [45]. The affinity value for a pair of operations reflects their proximity in the DFG. In the edge-centric scheduler, each slot is assigned an affinity cost depending on how close it is to any already-placed operations that have high affinity with the target operation. This gives a preference for placing an operation near its producers and consumers, hence reducing the number of routing resources used.

4.2.2.2 Proactively Avoiding Routing Failure

Figure 4.2 gives an example of when naïve routing of an edge can lead to routing failures of other edges. The DFG on the left is mapped onto the example CGRA in Figure 4.1(a). The six operations at the top are being placed and the three at

the bottom have not been placed yet. The operation ST at the bottom is a store operation; assume that only FU 4 can execute memory operations. When routing the edge from P1 to C1, there are three possible paths (R0, R1, and R2) as shown in Figure 4.2(b). All three paths use the same number of routing resources. However, there is a preferred choice when routing of other edges is considered. First, the path on the left (R0) should not be selected because it would block the only path between P2 and C2, causing a subsequent routing failure from P2 to C2. The path in the middle (R1) is preferred to the path on the right (R2) because occupying slot (4,3) leaves only two memory slots of FU4 for the ST operation. So, the scheduler will have fewer options when scheduling the ST, leading to a greater chance of routing failure in the future.

From the previous example, we can see that the scheduler needs to know the resources that are likely to be used by other edges in the future. To account for this, the scheduler associates an occupancy probability with each scheduling slot. The probabilities are calculated for two different types of operations: expensive operations and placed operations.

Expensive operations are defined as ones that only a subset of FUs can execute, such as memory and multiply operations. For each scheduling slot that can execute expensive operations, the probability is calculated by dividing the number of unscheduled expensive operations by the number of remaining slots that are compatible. When non-expensive operations are scheduled, the router prefers to avoid using slots that are capable of supporting expensive operations. For operations already

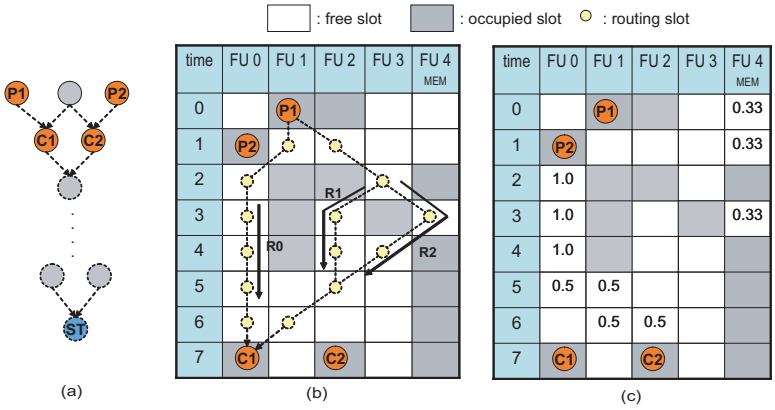


Figure 4.2: Routing cost example: (a) dataflow graph, (b) possible mappings, and (c) probabilistic cost.

placed in the scheduling space, the scheduler determines how many routing options there are for routing values to either producers or consumers.

For the placed operation P2 in Figure 4.2(c), probabilities are annotated in each reachable slot depending on the number of routing options. Empty slots in FU 4 are also annotated with a probability of 0.33 calculated by dividing the number of memory ops left by the number of available slots. These probabilities are accounted for when the routing cost is calculated for each slot, and the router will visit slots in the order of routing cost.

4.2.3 Stage Re-assignment

In modulo scheduling, better throughput (smaller II) is often achieved by scheduling some operations up front. A good example is operations on recurrence cycles. Since each iteration is executed every II cycles, all operations in the recurrence cycle must be scheduled within II cycles. For this reason, most modulo scheduling

algorithms process operations on recurrence cycles prior to other operations.

When placing an operation in a recurrence cycle early in the scheduling process, it is likely that there are no producers or consumers placed already. In a conventional modulo scheduler, the scheduler utilizes ASAP/ALAP (as soon/late as possible) times calculated statically by looking at the longest paths between operations. In CGRA scheduling, the ASAP/ALAP time is not an accurate measure of the actual time slot because routing can take multiple cycles. If an operation is scheduled too early, the scheduler will fail to place its predecessors. If an operation is scheduled too late, there can be a waste of routing resources or increase in register pressure.

Accurate ASAP/ALAP times are not easily obtained in CGRA scheduling because they depend on routing latency which is not known a priori. Thus, we take an alternative approach: placed operations can be lowered or hoisted along the time axis by re-assigning the stage. Since only stage count is changed, the resource occupancy status does not change. When an operation's stage is changed, operations connected to it in the scheduling space and routing between them must be moved as well. Since all the connected components are moved together, the stage reassignment is a local transformation and does not affect other operations.

An example of stage re-assignment is shown in Figure 4.3(a). Operations B and C form a recurrence cycle and are initially scheduled in stage 1 (times 2 and 3). Later, when operation A is being scheduled, the router is called for the edge from A to B. Since resources are repeatedly used every II cycles, FU 3's slot at time 6 is also occupied by operation B. Operations A and B are not connected by any placed edge,

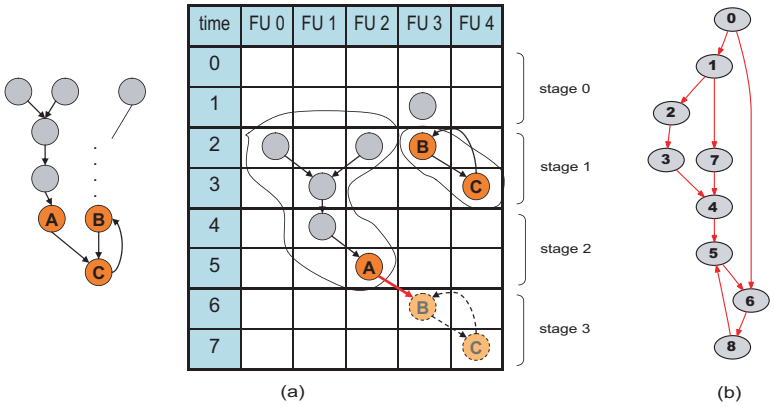


Figure 4.3: (a) Stage re-assignment example (II = 2) that re-assigns the recurrence cycle B-C from time 2-3 to time 6-7 after operation A is scheduled; (b) Example dataflow graph to illustrate non-critical edges.

so B can be re-assigned to time 6 (in stage 3). Since operation C is connected to B by a placed edge, it is also re-assigned to time 7.

4.2.4 Edge Categorization

Modulo scheduling for the CGRA is a problem of allocating a fixed number of routing resources to the edges in the DFG. It is important to observe that not all edges are the same in terms of how important they are to the overall schedule. In EMS, edges in DFGs are categorized as described below, and different routing approaches are applied for each edge type.

Recurrence edges. It is crucial to schedule the edges in a recurrence cycle ahead of other operations, especially when the II is close to the length of the recurrence. These edges are thus scheduled with highest priority.

Simple edges and high-fanout edges. Simple edges are defined as the outgoing edge of an operation that has only one consumer. When there are multiple consumers,

the outgoing edges are called high-fanout edges. With the limited number of routing resources, edges routed earlier are likely to use less routing resources than edges routed later, since there is more flexibility when slots are not yet occupied. Therefore, the scheduler needs to intelligently decide which edges are routed first.

The edge-centric scheduler gives priority to simple edges over high-fanout edges for the following reason. When a simple edge is routed later and thus is not optimized very well, it will likely end up using more resources than required. Since there is no other consumer for the producer of the simple edge, those additional resources are just being wasted. However, additional resources in a high-fanout edge can actually be helpful when routing edges from the same producer to other consumers, since there are more resource slots that contain the producer's value.

An analysis on simulated annealing's result also shows this trend. Frequently, an operation that has multiple consumers is located far apart from its consumers on the time axis, while operations connected with simple edges are located close to each other. This observation motivates our priority calculation method using fanout clustering, described in the next section.

Non-critical edges. When there are multiple disjoint paths between a pair of nodes in the DFG, dependencies are generated between edges in different paths. An example is shown in Figure 4.3(b). Assume the recurrence cycle at the bottom (operations 5, 6, and 8) was scheduled first. When node 0 is scheduled, the scheduler sees that its consumer node 6 is already scheduled. However, the edge from 0 to 6 should not be routed yet because it is not on the critical path from 0 to 6. The

scheduler should wait until all of the edges in the critical path are routed before routing the 0→6 edge. Therefore, a dependency is generated from the 0→6 edge to the critical path between 0 and 6. Similarly, dependencies are generated for edges on paths between nodes 1 and 4. In this case, edges 1→7 and 7→4 depend on the critical path between nodes 1 and 4. When an edge has a dependency on a pair of nodes, the routing of the edge is deferred until the edges on the critical path are scheduled.

4.3 Implementation

This section describes the implementation of EMS. The system flow is shown in Figure 4.6. First, the DFG of the target loop is converted into a reduced form by collapsing some nodes. The reduced DFG is then clustered by ignoring high-fanout edges and operations are prioritized based on the clustered result. Then, the operations are scheduled either by calling a placement function or calling a routing function depending on whether they have previously placed producers or consumers. After finding a legal schedule for the given II, the collapsed nodes are expanded first and configurations are generated for each component. If scheduling fails, the scheduler increases II and repeats scheduling.

4.3.1 Prepass Steps

Generating the Reduced Dataflow Graph

First, the DFG is converted into a reduced form where certain nodes are collapsed into edges. An operation is collapsible if it is inexpensive (can execute on any FU

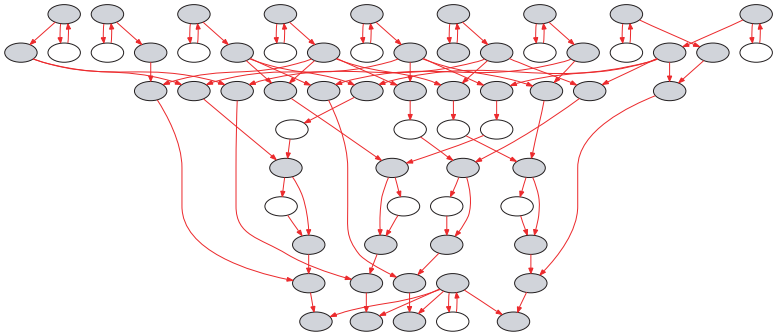


Figure 4.4: An example dataflow graph from H.264.

in the array), and has only one producer and one consumer. When such a node is found, the scheduler removes it and draws an edge directly from its producer to its consumer. The new edge is annotated with the number of nodes that were collapsed. This simplifies the DFG, and also allows the router to treat a path of nodes as a single edge during routing, potentially leading to a better schedule for that path.

In the DFG in Figure 4.4, collapsible nodes are shown in white. When these nodes are collapsed into edges, a reduced DFG (RDFG) is generated as shown in Figure 4.5. In all, 17 out of 65 nodes were collapsed, resulting in a smaller scheduling problem. For the loops in the media applications evaluated in Section 4.4, 18% of nodes were collapsed on average.

Priority Calculation using Fanout Clustering

The scheduling priority of operations in the RDFG are calculated in such a way that simple edges get higher priority than high-fanout edges, as described in Section 4.2.4. First, the DFG is clustered by ignoring high-fanout edges. Each group of nodes connected by simple edges forms a cluster as shown in Figure 4.5. The scheduler processes clusters such that each cluster is scheduled as soon as all of its

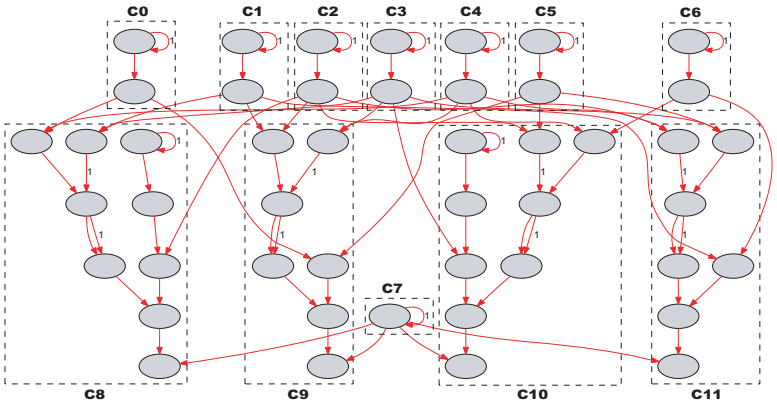


Figure 4.5: Example from Figure 4.4 after fanout clustering.

producers are placed. Within a cluster, producer operations are also scheduled before consumers. Basically, nodes are visited in a post-order traversal starting from the bottom.

For the target loop in Figure 4.5, the operations in recurrence cycles are scheduled up front. Then, the scheduling order of each cluster is determined. The scheduler will start with C8, which is one of the clusters at the bottom. A post-order traversal gives an order of C0, C3, C1, C4, C2, C7 and C8. The final order for clusters are C0, C3, C1, C4, C2, C7, C8, C5, C9, C6, C10, and C11. Within a cluster, operations are scheduled the same way.

4.3.2 Edge-centric Modulo Scheduler

Once priorities are calculated for all nodes in the RDFG, the nodes are scheduled. For each target operation, first the scheduler determines whether there are any placed producers or consumers. If not, the target operation is placed in a scheduling slot with minimum cost; this is the only time where the placement function is called. For an

operation that has placed producers or consumers, the scheduler decides which edge to route first. The decision is made based on various factors such as schedule time and stage-changeability of producers or consumers, and how many routing options are available.

When an edge is selected, the router is called and it first decides the routing direction. Forward routing starts from the producer and finds a compatible slot for the consumer; backward routing does the opposite. When both producer and consumer are placed, both directions are possible, and the decision is made based on stage-changeability of the producer and consumer. Since only operations at the end of a route can have their stages re-assigned, the router will select a direction that starts from a fixed operation.

4.3.2.1 Search Window Setup

The router will visit neighboring scheduling slots starting from a slot where a source operation is placed. The scheduler needs to set up the time axis of the search window with care. A search window that is too small can result in failure to find a compatible slot, while there can be a waste of time if a window is too large. Even though ASAP/ALAP times are not an accurate measure of the time slots for operations to be placed, they can be a good lower/upper bound for routing. The search window is determined by ASAP/ALAP time of the target operation considering stage re-assignment. When routing an edge from a placed producer (P) to a non-placed consumer (C), ASAP time can be calculated by Equation 4.1. p denotes a placed

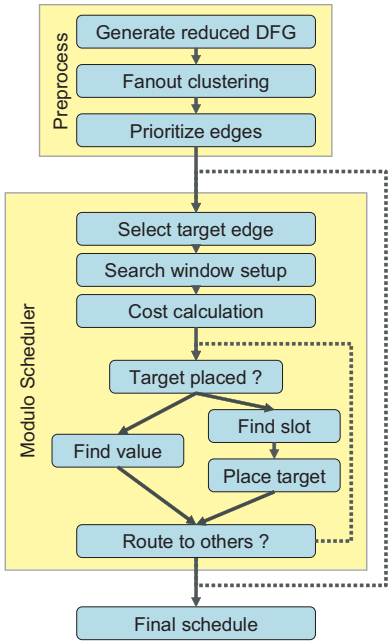


Figure 4.6: System flow for edge-centric modulo scheduling.

predecessor of C. *d(x, y)* is the longest path delay between x and y. *up(x)* is the max number of stages x can be hoisted and *dn(x)* is the maximum number of stages x can be lowered. Similarly, ALAP time is calculated by Equation 4.2 where s denotes a placed successor of C.

$$ASAP(C) = MAX(time(p) + d(p, C) - (up(p) - dn(P)) \times II) \tag{4.1}$$

$$ALAP(C) = MIN(time(s) - d(C, s) + (up(P) - dn(s)) \times II) \tag{4.2}$$

4.3.2.2 Routing Cost Calculation

When scheduling an edge, a routing cost is calculated for each available slot. This cost is used by the router to determine the order in which to explore slots during

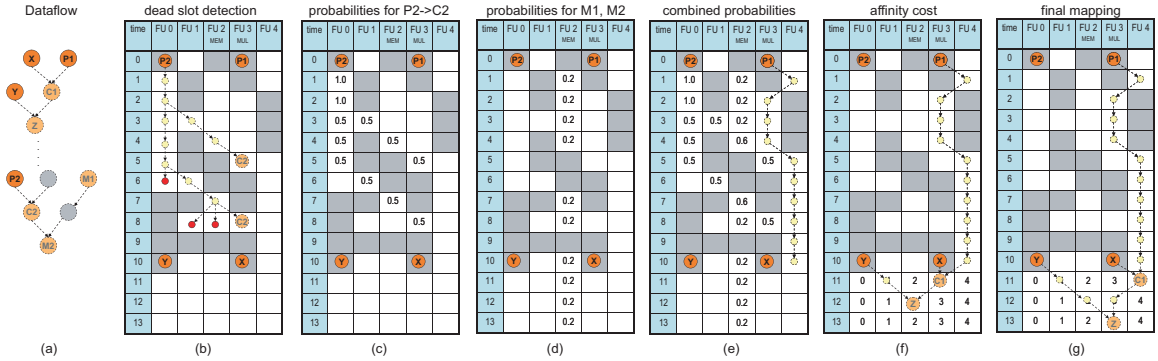


Figure 4.7: Routing cost calculation example: (a) dataflow graph, (b) - (g) reservation table with computed routing costs.

routing. Routing cost has three primary components, described below.

Static cost. A fixed cost C_{static} is assigned to each slot so that the scheduler can minimize the number of routing resources used.

Affinity cost. As described in Section 4.2.2.1, affinity cost is calculated based on a slot's distance from placed producers. Equation 4.3 calculates the affinity between two operations A and B. Affinity is given to a pair of operations that have common consumers (direct or indirect use of the destination of A and B). Common consumers within *max_dist* in the DFG are considered for affinity calculation. *num_cons(A,B,d)* denotes the number of common consumers of A and B at the distance d in DFG.

$$affinity(A,B) = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times num\_cons(A,B,d) \tag{4.3}$$

The affinity cost C_{aff} is then calculated for each slot as follows, where *dist* is the distance in hops from the current slot to the slot where the producer is placed. When there are multiple placed producers, C_{aff} is summed for all producers.

$$C_{aff} = \begin{cases} 0 & affinity(A,B) = 0 \\ \frac{dist}{affinity(A,B)} & affinity(A,B) > 0 \end{cases} \tag{4.4}$$

Probability cost. The router should take care not to block certain slots because they may be required for routing of future edges. Thus, a cost is assigned to each slot reflecting the probability that it will be required in the future. There are two cases: reserving expensive slots, and reserving slots to route results of previously placed nodes. The individual probabilities are calculated as described in Section 4.2.2.2. These probabilities must then be combined together, as a given slot may support multiple types of expensive operations and/or be used to route multiple placed nodes. Since the individual probabilities are correlated, getting the exact overall probability for a slot is difficult. An approximation is obtained by treating the probabilities independently. The following equation expresses the total probability P of a slot given n individual probabilities p_i:

$$P = \sum_{k=1}^{n} \left((-1)^{k-1} \sum_{\substack{I \subset \{1,\ldots,n\} \\ |I|=k}} \prod_{i \in I} p_i \right) \tag{4.5}$$

Total routing cost. The total routing cost C for a slot is obtained by combining the three costs above:

$$C = \begin{cases} C_{static} + w_{aff} \times C_{aff} + w_P \times P & P < 1 \\ \infty & P = 1 \end{cases} \tag{4.6}$$

The costs are combined with weighting factors w_{aff} and w_P. In addition, if $P = 1$, the slot will definitely be required in the future and cannot be used for routing the current edge; thus, routing cost is infinite.

4.3.2.3 Finding the Target

Once all routing costs are updated, the router will start finding a path from the source to the target operation. Starting from a slot that contains the source operation, the router visits neighboring slots in the CGRA using a maze routing technique. Each neighboring slot is put into a priority queue and the router visits the slots in order of their routing costs as calculated above.

When a collapsed edge is routed, the router ensures that it finds a path that goes through at least as many FUs as the number of collapsed nodes, so that the collapsed nodes can be expanded later into those FUs. A similar approach is taken for high-fanout edges. Because the high-fanout edges are scheduled with low priority, the corresponding values are likely to have long lifetimes. Therefore, when high-fanout edges are routed, the scheduler attempts to find a path that goes through a register file.

If the target is already placed, the route is towards the slot that contains the target operation. Otherwise, it will find a slot that can execute the target operation. Once a slot is found, the scheduler checks if other edges connected to the target need to be placed, and recurses to route those edges. When an edge has a dependency on other edges as described in Section 4.2.4, the routing is deferred until all edges in

more critical paths are scheduled. When all of the edges are successfully routed, the scheduler moves on to the next operation in priority order.

When the scheduler places recurrence cycles, edges are placed even if their target operations are not placed yet. By calling the router function recursively for all operations in the cycle, the scheduler can put more effort into finding a legal mapping for the recurrence cycles. To prevent exponential compile time for large recurrence cycles, the number of recursive calls is limited to a fixed value. When the scheduler successfully routes all the connected edges, it finalizes the placement of the target operation and proceeds with the next one.

4.3.2.4 Routing Example

Figure 4.7 shows an example of how EMS routes an edge with updated routing costs for each slot. Again, we assume no register files in the target architecture for illustration purposes. The DFG in Figure 4.7(a) is mapped onto the 1x5 CGRA. Here, we assume that P1, P2, X, and Y are already placed and the scheduler is about to route the edge from P1 to C1. Further, C2 is a multiply operation and can only execute on FU 3, and M1 and M2 are memory operations and can only execute on FU 2. First, the scheduler calculates probabilities of routing slots generated for the unplaced edge from P2 to C2 (Figure 4.7(b)). Then, it identifies dead slots that will not lead to any compatible slots for C2 , as indicated by dark small dots in Figure 4.7(b). Once all the dead slots are identified, probabilities are propagated along the routing live slots. Figure 4.7(c) shows the final probabilities. Slot (0,2) gets

1.0 since there is only one path from P2. Slots (0,3) and (1,3) get the probability of 0.5 since there are two routing options from the previous slot.

Next, probabilities are generated for the expensive operations, M1 and M2, that are not placed (Figure 4.7(d)). With two expensive operations and 10 available slots on FU 2, each slots gets a 0.2 probability.

The probabilities in Figure 4.7(c) and Figure 4.7(d) are combined using Equation 4.5 resulting in Figure 4.7(e). Based on the probabilities calculated for unplaced edges and nodes, the router finds a path for the edge from P1 to C1 as shown in Figure 4.7(e). There are two candidate slots for C1; slot (3,11) and slot (4,11). Since C1 and Y have a common consumer Z, the placement of C1 can affect the number of routing resources used later when the edge from Y to Z is routed. As shown in Figure 4.7(f) and (g), slot (3,11) is preferred to slot (4,11) when considering the common consumer Z. EMS utilizes the affinity heuristic [45] to make this decision. For each slot, the affinity cost is assigned in a way that a higher cost is given as the distance from Y increases. Therefore, the scheduler prefers slots that are close to Y and (3,11) is selected. Later when Z is scheduled, the routing cost can be reduced since Y and C1 are placed close to each other.

4.3.2.5 Register Constraints

In CGRAs, values with long live ranges can be more efficiently routed through distributed register files. The scheduler must carefully manage register resources so that values stored in the register file are successfully routed to consumers. Traditionally,

register allocation is performed after scheduling, and spill code is inserted when the register requirement exceeds the register file capacity. Spilling in the CGRA is quite costly since it involves routing to/from the memory units and may require complete rescheduling of the loop. Moreover, spilling can easily happen due to the small size of the register files.

EMS performs register allocation during scheduling to avoid spilling and guarantee routability through the register files. Register allocation occurs frequently, as it is needed whenever the router visits a register file. So, a simple and fast allocation scheme was developed that focuses on the routability of stored values. Since EMS gives low priority to high-fanout edges, consumers of the same value are typically scheduled in different times. The scheduler needs to ensure that values stored in register files can be routed to all of their future consumers. The details are omitted in this paper due to space constraints.

4.3.3 Postpass Steps

When EMS finds a legal schedule, it generates the contents of the CGRA's configuration memories. First, it expands the collapsed operations onto the FU slots that were found. Then, control bits for the routing and computation resources are generated, including MUX selection bits, FU opcode bits, and register file addresses.

4.4 Experimental Results

4.4.1 Experimental Setup

To evaluate the performance of EMS, we took 214 loops from four media applications from the embedded domain (H.264 decoder, 3D graphics, AAC decoder, and MP3 decoder). The loops, varying in size from 4 to 142 operations, were mapped onto different CGRA configurations.

The target CGRA architecture is a 4×4 heterogeneous array as shown in Figure 2.1. Functionality for memory access is limited to 4 FUs and multiplication to 6 FUs. The array contains a 64 entry (16 of which are rotating) central RF with 8 read and 4 write ports wherein only FUs in the first row can directly read/write. All other FUs can only read from the central RF via column buses. The central RF is primarily used for storing live-in values from the host processor. Each FU has its own local RF consisting of 8 rotating register with one read and one write port. Local RFs can be also written by FUs in diagonal directions (upper right/upper left/lower right/lower left). For example, local RF in PE 5 can be written by FUs 0, 2, 5, 8 and 10 and only FU 5 can read from it.

We created three architecture instances by differentiating FU and RF connectivity: mesh-plus, mesh-only and no-RF-sharing. In mesh-plus, FUs are connected in a mesh network, meaning that each FU is connected to its immediate neighboring FUs. Additionally, FUs that are two hops apart are also connected. This is a similar configuration to ADRES [38]. In the mesh-only configuration, FU connectivity is

limited to a simple mesh network. The no-RF-sharing configuration has same FU connectivity as mesh-only, but local RFs are not shared by FUs in diagonal directions, meaning that each RF can be written/read only by the neighboring FU.

The performance and compile time of EMS were compared to three different modulo scheduling techniques: **IMS**: traditional iterative modulo scheduler that does not consider routing efficiency; **NMS**: node-centric modulo scheduler that employs the same heuristics as EMS, but scheduling is conducted in a node-centric way; and, **DRESC**: IMEC's simulated annealing based modulo scheduler. All evaluations were taken on an Intel Core 2 Duo system running at 2.66GHz with 2GB memory. Compile time was measured by using only one core of the system. Scheduling results were verified with a cycle accurate simulator.

4.4.2 Results

In modulo scheduling, MII defines the theoretical upper bound of the performance of the scheduled loop. Therefore, we calculated the performance of the modulo scheduler by dividing MII by the achieved II in each loop. The performance comparison of the four different modulo scheduling techniques is shown in Figures 4.8, 4.9, and 4.10 for the mesh-plus, mesh-only, and no-shared-RF configurations, respectively. The first four groups show the performance results of the loops within each domain and the last group shows the overall performance across all 214 loops.

A more detailed view of the performance comparison between EMS and DRESC is presented in Figure 4.11 for the mesh-plus configuration. The x-axis shows all 214

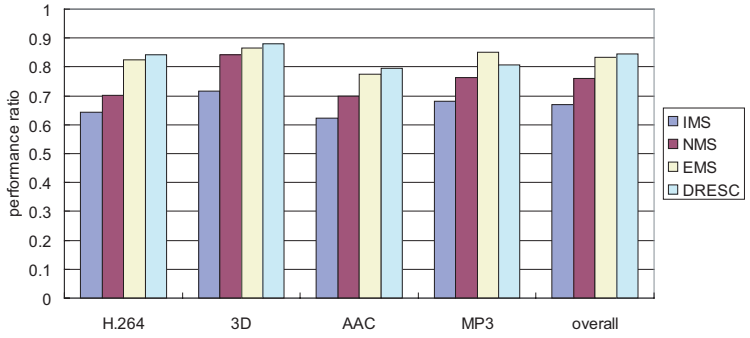


Figure 4.8: Performance comparison of scheduling strategies for the mesh-plus architecture. The fraction of the theoretical maximum performance is plotted.

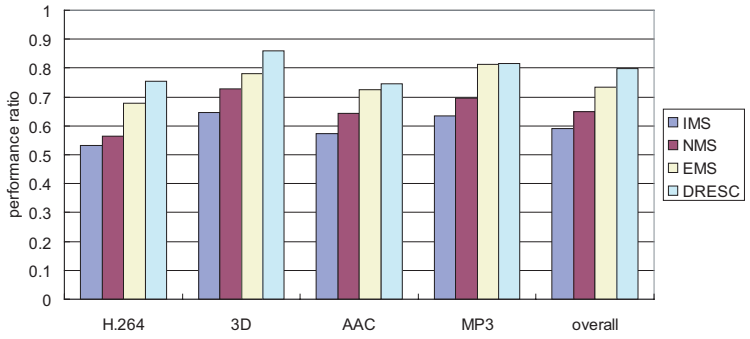


Figure 4.9: Performance comparison of scheduling strategies for the mesh-only architecture.

target loops grouped by application. Within each application, loops are sorted by increasing MII. The gray line shows the value of MII for each loop. The achieved II for EMS is shown as solid circular dots. The achieved II for DRESC is shown only when it differs from EMS's achieved II, as a vertical line extending from the dot. For the mesh-plus architecture, EMS achieves an average ILP of 9.6 across all the loops.

The final measurement performed is compilation time. The total compile time of all 214 loops for each scheduling technique is shown in Table 4.1.

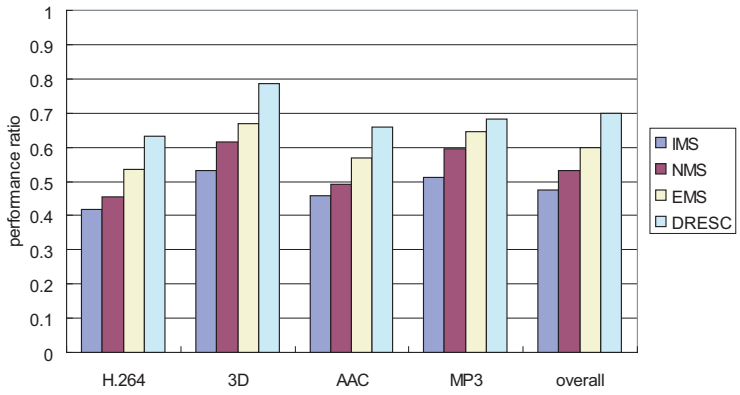


Figure 4.10: Performance comparison of scheduling strategies for the no-RF-sharing architecture.

arch	IMS	NMS	EMS	DRESC
mesh-plus	655	2105	1185	22341
mesh-only	1122	3046	2228	48035

Table 4.1: Compile time comparison (in seconds).

4.4.3 Analysis and Discussion

Comparison with IMS. EMS always outperforms traditional IMS by more than 25% for both mesh-plus and mesh-only configurations. Even though IMS works quite well for conventional VLIWs, the lack of a global resource management strategy causes frequent routing failures which forces II to be increased.

Comparison with NMS. EMS and NMS share most of the heuristics developed in this paper, such as the various cost metrics, stage reassignment, and the reduced dataflow graph. However, EMS achieves 10-13% performance increase while compile time was reduced by 27-46% compared to NMS. This demonstrates the benefits of the edge-centric over the node-centric approach in both performance and compile time measures, as illustrated in Section 4.2.1.

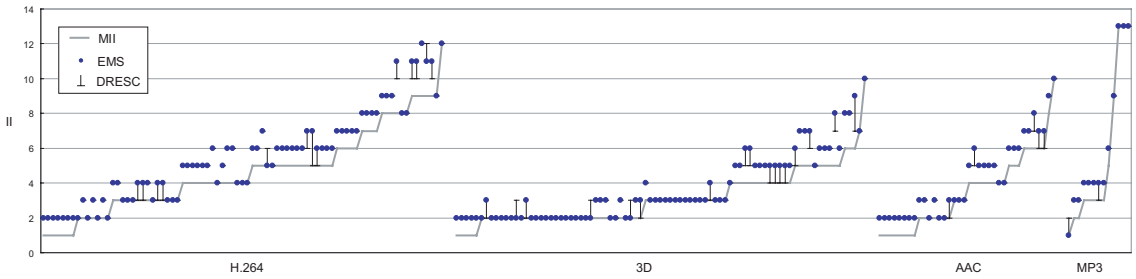


Figure 4.11: Performance comparison of EMS and DRESC for the mesh-plus architecture.

Comparison with DRESC. DRESC consistently achieves the best IIs for most of the applications, except MP3 in the mesh-plus architecture. Simulated annealing is an effective strategy for CGRA scheduling, but its high performance comes at the cost of slow compile time. When compared to DRESC, EMS shows quite competitive performance results, achieving 98% and 91% of DRESC's overall performance for mesh-plus and mesh-only architectures, respectively.

For the mesh-plus architecture, EMS shows virtually the same performance as DRESC, achieving the same II or better for more than 85% of loops (Figure 4.11). For most of the loops that are scheduled at higher IIs, the large number of live-ins was the bottleneck for EMS. Since all of the live-ins are stored in the central RF, there is high contention for central RF ports among the operations that consume live-ins. Though EMS reserves these high contention resources by calculating probabilities in advance, it still fails to achieve the same II as DRESC when the contention is too high.

For the mesh-only architecture, EMS does not perform as well, especially for H.264 and 3D. Those two domains have many communication patterns in which one pro-

ducer feeds multiple consumers. The execution of such communication patterns is significantly limited with the sparse interconnect in the array. This trend is more obvious when looking at the results of no-RF-sharing configuration 4.10. EMS is achieving 85% of DRESC's performance when interconnected further reduced by removing shared links to local RFs. This result shows that EMS is more vulnerable to a lack of routing resources. We are currently investigating CGRA designs that have low hardware cost but still enable EMS to achieve high performance.

Compile time. Since there are no intelligent heuristics for global management of routing resources in IMS, it shows the fastest compile time among the four scheduling techniques. Except for IMS, EMS performs the fastest, showing more than 18x speedup over DRESC. A systematic approach for placement and routing indeed allows a reasonable compile time while achieving competitive performance. Compile times for mesh-only are larger than mesh-plus because the achieved IIs are usually higher. Since the scheduler starts at the MII for each loop, it takes more time to get to the solutions with higher IIs.

Effectiveness of Heuristics. EMS employs various heuristics to guide the scheduler towards intelligent routing. The effectiveness of individual heuristics varies based on the application characteristics. The probability heuristic is effective for loops that have high contention on limited resources such as central RF ports or memory slots. Prioritizing edges based on the edge dependency analysis effectively schedules loops with large recurrence cycles, especially when there are many recurrence cycles and some nodes are included in multiple cycles. Stage-reassignment is effective when

DFGs have narrow and tall shapes.

4.5 Related Work

Architectures. Many CGRA-like designs have been proposed in the literature. The designs have different scalability, performance, and compilability characteristics as discussed in Section 2.1. The ADRES architecture [38] is an example of an 8x8 mesh of processing elements with both individual and central register files. MorphoSys [36] is another example of an 8x8 grid with a more sophisticated interconnect network; each node contains an ALU and a small local register file. In the RAW architecture [52], each node is actually a MIPS processor, including memory, registers, and a processor pipeline. In addition, there are both dynamic and static routing networks. PipeRench [18] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining. RaPiD [13] consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

Compilation Techniques. Many techniques have been proposed for compiling to CGRAs. Lee et al. [29] propose a compilation approach for a generic CGRA. They generate 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. Our work proposes a generic scheduling strategy, and memory sharing and other such optimizations can be integrated into our system as a preprocessing step. [1] investigated a loop-scheduling problem in CGRA by dividing it into covering, partitioning and

layout subproblems. It spatially partitions the CGRA and maps each loop iteration onto the partitioned CGRA. Modulo scheduling differs from this approach in that it time-multiplexes the array for different loop iterations.

RAWCC [30] tackles the scheduling problem for the RAW architecture where all the communication is fully exposed to the compiler. The scheduling problem is broken down into two tasks: spatial assignment and temporal assignment. Operations are placed in each tile first, and time slots are assigned for operations in each time. Convergent scheduling [31] is another compiler technique proposed as a generic framework for instruction scheduling on the RAW architecture. Their framework comprises a series of heuristics that address independent concerns like load balancing, communication minimization, etc. [40] and [7] were also proposed for instruction scheduling of tiled architectures. The scheduling problem in tiled architectures is quite similar to our problem in that the compiler has to manage communications explicitly among computation resources. The main difference is that tiled architectures usually have a dynamically routed network that can sustain some level of routing congestion during runtime. Having no such routing network in CGRAs, the scheduler is responsible for orchestrating every communication so that no congestion occurs. Whereas [30], [31], [40] and [7] focus on ILP and propose scheduling methods for acyclic regions of code, we focus on loop level parallelism. The work of Mei et al. [38] is closest to our work, as discussed in Section 4.1.

Similar to CGRAs, clustered VLIW machines are also spatial architectures. Much work has been done towards compiling for clustered VLIW machines [15, 42, 50]. Al-

though some of the concepts from these works can be adapted for CGRA compilation, they do not consider the issue of routing values through the sparse interconnection network, which is a crucial step. The measure of affinity used in our scheduler is similar to that used in Krishnamurthy's affinity-based clustering [26].

Stage scheduling [14] re-assigns operations' stages to minimize register pressure for modulo scheduled loops. While stage scheduling is applied as a post pass, EMS re-assigns stages during the modulo scheduling process.

4.6 Summary

In this chapter, we proposed edge-centric modulo scheduling, an effective modulo scheduling technique for CGRAs. The distributed nature of CGRAs, including sparse interconnect and distributed register files, presents difficult challenges to a compiler. EMS focuses primarily on the routing problem, with placement being a by-product of the routing process. Various routing cost metrics were introduced to give a global perspective of resource management to the scheduler. Edges in the dataflow graph are categorized based on their characteristics and EMS uses different strategies to route them. Overall, EMS improves performance by 25% over traditional modulo scheduling and achieves 85-98% of the performance compared to a state-of-the-art simulated annealing technique. EMS also reduces compilation time by 18x compared to simulated annealing.

CHAPTER 5

Control Path Optimization

5.1 Introduction

A major bottleneck for deploying CGRAs into a wider domain of embedded devices lies in the control path. The appealing features in the datapath of CGRAs ironically come back as a major overhead in the control path. The distributed interconnect and register files require a large number of configuration bits to route values across the network. The abundance of computation resources simply adds up the list for configurations to the control path. As a result, the total number of control bits to configure the whole array can reach nearly 1000 bits each cycle, and the control path takes up to 43% of the total power consumption in existing CGRA designs [25, 5]. Moreover, control bits are read from the on-chip memory every cycle regardless of the array's utilization. Even when only a small portion of the resources are active in the array, the configurations for all the resources must be fetched, which makes CGRAs very inefficient for the codes with limited parallelism. This inefficiency pre-

vents CGRAs from wider uses including outer loop level pipelining [49] or simply running acyclic code to reduce the communication overhead with the host processors. Finding an efficient way to reduce the control power reduction will not only relieve the power overhead in the control path, but also opens the future application of CGRAs to more variety of workloads.

While there are many studies on architecture exploration, code mapping, and physical implementation [38, 1, 28], relatively little work has examined efficient control in CGRAs and other tiled accelerators. One exception is [25] wherein a hybrid configuration cache is proposed that utilizes the temporal mapping for control power reduction. Temporal mapping only utilizes a single column of PEs in the array to map the entire loop and the execution of the loop is pipelined by running multiple iterations on different columns in the array. The control power can be substantially reduced by transferring the configurations in one column to its right each cycle, letting only the leftmost column read from the configuration memory. However, temporal mapping can be applied to only certain types of loops and it is not a general approach that can scale to different types of applications. [5] reduced the control path power of CGRAs as a by-product of an architecture exploration. A Pareto optimal design of a CGRA was discovered that required a lesser number of resources in the datapath thereby resulting in a power reduction in the control path. In this chapter, we propose a new control path design that improves the code efficiency of CGRAs by leveraging token networks originally proposed for dataflow machines.

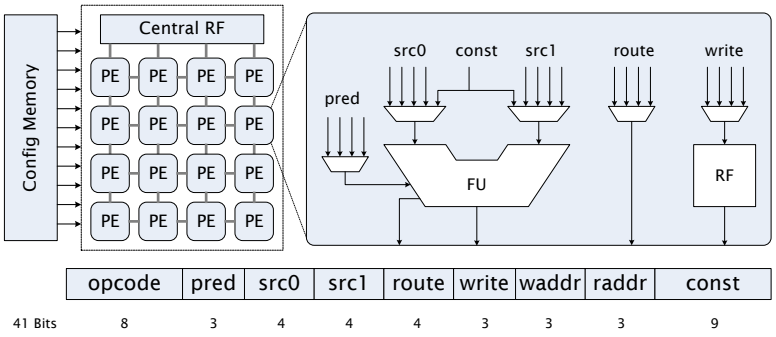


Figure 5.1: CGRA overview: 4x4 array of PEs (left), a detailed view of a PE (right), and a PE instruction (bottom)

5.2 Motivation

Figure 5.1 shows our target CGRA, similar to [38]. There are 16 PEs connected in a mesh-style interconnect and a central register file for transferring values from/to the host processor. Each PE has one FU for computations and an 8-entry local register file that are shared by other neighboring PEs. An FU has three source multiplexors (MUXes) for predicate and data inputs. Here, we assume an additional MUX(route) in each PE to increase the routing bandwidth of the array. So, the PE can do both computation and routing in one cycle. There are several MUXes as a result of the distributed interconnect and each of them requires selection bits encoded in the instruction field. Also, each register read/write port requires an RF address field. Along with PE instructions, there are instructions for central register files, and other buses that also require configuration. As a result, each PE instruction is 41 bits, and a total of 845 bits is required to configure the CGRA each cycle. Typically, control signals in CGRAs are stored as a raw data (fully decoded instructions) and directly fed to the datapath as shown in Figure 5.2(a). Fetching 845 bits every cycle is indeed

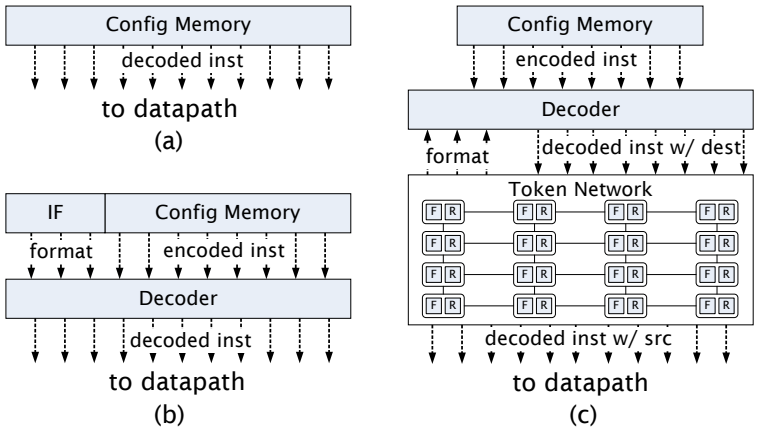


Figure 5.2: Different Control Path Designs: (a) No compression, (b) Fine-grain code compression with static instruction format, (c) Fine-grain code compression with a token network (F and R indicate FU token module and RF token module, respectively)

a large overhead. Control path power can obviously be reduced by increasing code efficiency through some form of code compression technique.

Conventionally, code compression is performed at the instruction level with no-op compression or a variable length encoding. No-op compression is widely used in VLIW processors and many DSPs [54, 51, 43, 32, 34]. However, instruction-level compression does not work well in CGRAs due to the highly distributed nature of the resources. Even if an FU is sitting idle, the register file in the same PE can still be accessed by neighboring PEs. Also, the FU can be used for bypassing data from one PE to another. We examined the schedules of several hundred compute-intensive kernels taken from multimedia applications mapped onto our CGRA design and discovered that only 17% of PE instructions are pure no-ops (all the components in the same PE is not active), while the average utilization of FUs is 55%. Thus, no-op compression would have limited effectiveness.

However, there is a good opportunity for a fine-grain code compression: compressing instruction fields (e.g., opcode, MUX selection, register address) rather than the whole instruction. On average, only 35% of all instruction fields contain valid data, thus efficiency can potentially be increased by removing unused fields. Figure 5.2(b) shows a high-level organization that utilizes a static fine-grain compression approach. In the simplest variant, presence bits are added for each field to indicate whether the field exists or not. Instruction encoding consists of the presence bits(instruction format) followed by the subset of valid instruction fields concatenated together. With this approach, decoding can become complex due to the variable length nature of the encoding, but all unused fields can be removed in principle.

The biggest challenge for applying static fine-grain compression lies in the instruction formats. Using a simple fine-grain static compression scheme that we designed for a CGRA, the code efficiency increases by 24% with the average number of instruction bits decreasing from 845 to 647. However, 172 of the 647 bits are used for encoding the instruction formats. Since the instruction format of 172 bits needs be read from the configuration memory every cycle regardless of the number of fields present, the instruction format itself becomes a significant overhead in the control path. To address this limitation, we propose to dynamically discover the instruction formats by applying a dataflow token network explained in Section 5.3.

Another issue in employing the fine-grain code compression is decoder complexity. Since compression is performed in a finer granularity, the overhead of the decoder is more substantial than instruction-level code compression. In Section 5.4, we ana-

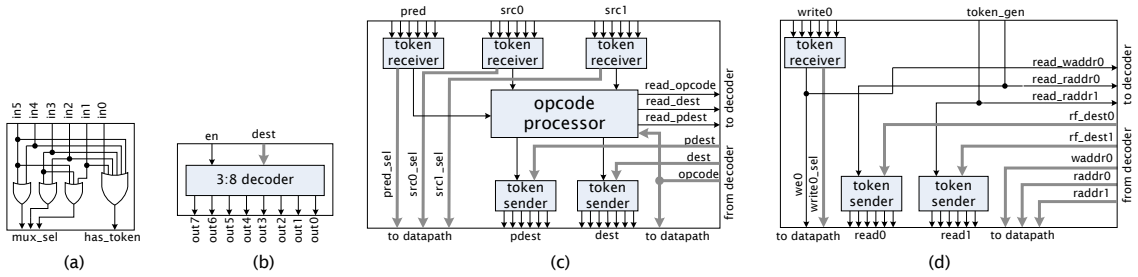


Figure 5.3: Token Modules: (a) token receiver, (b) token sender, (c) FU token module, (d) RF token module

lyze the decoder features that affect the overall complexity and discuss an efficient partitioning of configuration memory to reduce decoder complexity.

5.3 Dynamic Discovery of Instruction Formats

In this section, we propose a dynamic discovery of instruction formats by adopting the concept of a token network from dataflow machines. The concept is explained first, and then we propose a token network that can assist the fine-grain code compression to reduce the overall power consumption in the control path of CGRAs. Lastly, we discuss how the token network is extended to support modulo scheduled loops [48] to exploit loop-level parallelism in kernel loops.

5.3.1 Concepts

The basic idea of dynamic instruction format discovery is that resources need configurations only when there is useful data that flows through them. By looking at the locations of data coming into a PE, we can infer the instruction format of the

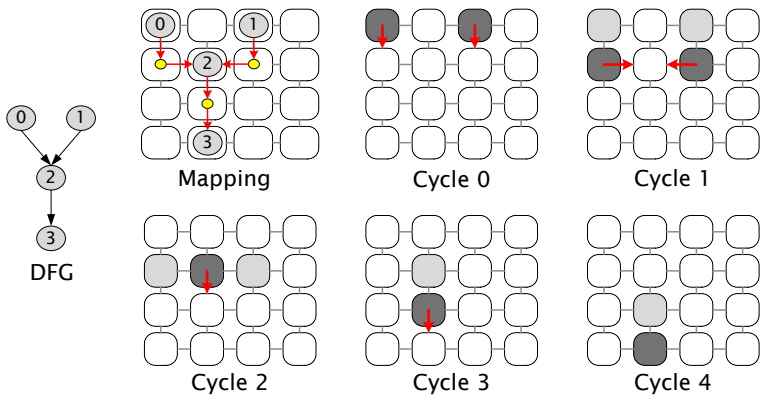


Figure 5.4: Dynamic configuration of PEs using tokens

current instruction. For example, two data coming into src0 and src1 MUXes of the FU in Figure 5.1 indicate that this FU will perform an ALU operation. So, an opcode field and src0/src1 MUX selection fields are required in that cycle. If there is no data coming into the predicate input MUX, the ALU operation is not predicated and the selection bits for pred MUX is not needed. When there is only one data coming into either the src0 or src1 MUX, the FU is performing a move operation and the opcode field is not required. In the same way, a data coming out from the register file in Figure 5.1 indicates a read address field is required.

We can utilize the token network in dataflow machines [44] to provide information on where data flows in the distributed network. A token is sent from a producer to its consumers one cycle ahead of the actual data execution. Originally, the consumer gets fired when it accumulated sufficient tokens. However, this concept can be altered as all tokens for a single instruction are guaranteed to arrive at the same time. Hence, the set of tokens uniquely determine the instruction format so that the necessary fields can be fetched from the instruction memory. When the actual data arrives in

the subsequent cycle, the required instruction fields are already decoded and the PE is ready to execute the scheduled operation.

Figure 5.4 shows the big picture of how PEs are configured dynamically in the token network. A simple dataflow graph (DFG) is shown on the far-left and its mapping onto the CGRA datapath is shown next to it. PEs with a small dot indicate they are used for routing. The PEs in the array are incrementally configured each cycle using tokens as in the figure. In each cycle, dark grey PEs are configured and send out tokens to their consumers. In the subsequent cycle, PEs executing the given instructions are shown in light grey. At cycle 0, PE[0,0] (row 0, column 0) and PE[0,2] are configured first to execute operations 0 and 1, respectively, and they send out the tokens to their consumers. At the next cycle, PE[1,0] and PE[1,2] receive the tokens from their producers and are configured to route the data to PE[1,1]. In a similar fashion, PEs are configured as tokens flow over the array and all the necessary PEs to execute the DFG are configured at cycle 4.

5.3.2 Token Network

To utilize tokens for instruction format discovery, a token network is inserted between the decoder and the datapath as shown in Figure 5.2(c). The token network consists of two components: token interconnect and token modules. Each datapath element, such as an FU, RF and MUX, has a corresponding token module in the token network. Example token modules are presented in Figure 5.3. Token modules are connected by a 1-bit token interconnect that has the same topology as the datapath

interconnect. The token network takes the decoded instructions from the decoder and sends tokens across the token interconnect. The token network has two responsibilities. First, the token network provides the instruction formats to the decoder. Second, it generates control signals for the datapath.

5.3.2.1 Token Generation and Routing

Tokens are first generated at the start of data streams in the dataflow graph: live-in values. A token generated at the top of the dataflow graph flows across the array visiting different resources and finally terminates when it either reaches a register file or merges into another token in an FU. A token terminated in a register file can be re-generated later, creating another token stream.

For tokens generated from live-in, the generation information (time and resource) needs be encoded in the configuration memory since there is no producer that sends token to those nodes. The tokens coming out from register files also require their generation information stored in the configuration memory since the tokens can be re-generated anytime once they are stored in the register file. Therefore, the configuration memory will hold the token generation information for all the tokens coming out from register file read ports. Each cycle, the token generation information stored in the configuration memory fires tokens into the token network and the configurations for the datapath are generated as tokens flow across the array. (token_gen signal in Figure 5.3(d)).

After tokens are generated, they are routed following the edges in the dataflow

graph. To send tokens from producers to consumers, the destination information is stored in the configuration memory instead of the source information. The MUX selection bits in a PE instruction (Figure 5.1) are replaced by dest fields. As in dataflow machines, only two destinations are allowed for each data generating component (FU output ports, RF read ports). An analysis on the scheduling result of our benchmark loops shows that 86% of the communication patterns are unicast (requiring only one destination), and 98% of communications can be covered by two destinations. Therefore, the performance degradation with the limited number of destinations is minimal. The impact of this limitation is discussed in Section 5.5. For illustration purposes, only one dest field is shown in Figure 5.3(c) and (d).

5.3.2.2 Token Processing

Tokens flowing on the token network are utilized for two tasks. First, the instruction formats are discovered with tokens and they are sent back to the decoder. With these instruction formats, the decoder can decode the compressed instructions for the subsequent cycle. Also, the dest fields in the decoded instructions are converted into the source fields for MUX selection bits and sent to the datapath.

Token Receiver: Since only destination fields are encoded in the configuration memory, the source fields (MUX selection bits) for the datapath need be discovered when tokens are coming into the input ports of each resource. For each MUX in the datapath, a token receiver (Figure 5.3(a)) is created. A token receiver generates the MUX selection bits(MUX_sel) by looking at the position of an incoming token.

Since only one input of a token receiver can have incoming token, the MUX selection bits can be generated with several OR gates as in the figure. Along with the MUX selection bits, it also notifies the attached module (FU/RF token module) whether there is a token coming into this input port or not (has_token).

Token Sender: For each output port of a datapath element (FU output ports, RF read ports), a token sender (Figure 5.3(b)) is created in the token network to send out tokens to the consuming resources. It simply decodes the dest field (dest) and sends out tokens to the connected modules.

FU token module: Figure 5.3(c) shows an example of FU token module that has both predicate and data parts. The input MUXes of the FU have been translated into token receivers and the FU itself is replaced with an opcode processor. For the output ports of the FU, token senders are created in the figure. The opcode processor first takes 'has_token' signals from the attached token receivers and discovers the instruction format. The opcode processor sends out a 'read_opcode' signal when both src0 and src1 have incoming tokens. Also, it sends out read signals for dest fields of both data (dest) and predicate (pdest) if there is any incoming token in the input ports. The opcode processor also determines the latency of computation by looking at the opcode field. The dest fields from the decoder are fed into the token senders directly. When the opcode processor signals the token senders with an enable signal, they send out tokens to the designated consumers specified in the dest fields.

RF token module: A token module for a RF with 2 read/1 write ports is shown in Figure 5.3(d). Similar to FU token modules, a token receiver and token senders

are created for the write port MUX and two read ports, respectively. Any incoming token into the write port sends a read signal to the configuration memory for the write address field and it also sends a write enable signal. For the read ports of register files, there are no incoming tokens from the token network. Instead, the generation of tokens from the read ports are encoded statically in the configuration memory. When a token generation signal comes in, the RF module sends a read signal for the read address and the dest field.

5.3.3 Supporting Modulo Scheduled Loops

Compute intensive loops are generally mapped onto CGRAs using modulo scheduling: a software pipelining technique that exposes loop level parallelism by overlapping the executions of different loop iterations. The basic concept of modulo scheduling is illustrated in Figure 5.5(a). In modulo scheduling, each iteration starts execution before its previous iteration finishes. By overlapping the executions, modulo scheduling can exploit loop-level parallelism when there are enough resources. The time difference between beginnings of successive iterations is called initiation interval (II). In a steady state, modulo scheduling repeats the same pattern for II cycles and this is called kernel code. Only the kernel code is encoded into the instruction memory, while the pipeline fill/drain (prologue/epilogue) are controlled by staging predicates [48].

5.3.3.1 Initialization for Kernel Code Execution

In our encoding scheme, the configuration memory contains only the kernel code of target loops and this requires special support for executing modulo scheduled loops with the token network. Figure 5.5(b)-(c) illustrate the problem that arises. Here, we assume the loop kernel in Figure 5.5(a) is mapped onto an 1x4 CGRA. Figure 5.5(b) shows a possible mapping of operations X, Y, and Z on FU 2. The edges with an arrow head indicate tokens flowing on the token network. For operation X, a token arrives at cycle 3 and the operation is activated at stage 0. Similarly, operation Y and operation Z receive tokens at cycles 4 and 9, respectively, and they are activated at stages 1 and 2, respectively. The kernel code of the loop is presented in Figure 5.5(c). In the steady state, operations are executed in the order of Y, Z, and X and the opcodes for them are stored in the configuration memory in the same order. Therefore, opcodes in FU 2 should be consumed in the order of Y, Z, and X. The problem occurs in the prologue when a token arrives at FU 2 in cycle 3. Since Y and Z are activated in later stages (at cycles 4 and 9), the opcodes for Y and Z are not consumed yet from the configuration memory. As a result, FU 2 reads the opcode Y instead of X's opcode.

The solution for this problem is to maintain the kernel state from the beginning of the loop execution. We can achieve this by initializing the token network with the state of cycle II - 1. Once the state is initialized with the state of cycle II - 1, the tokens can flow through the network and generate the kernel code from the beginning. For initialization, the snapshot of the token network at cycle II-1 is stored separately in the configuration memory. At cycle -1(one cycle before loop execution starts), the

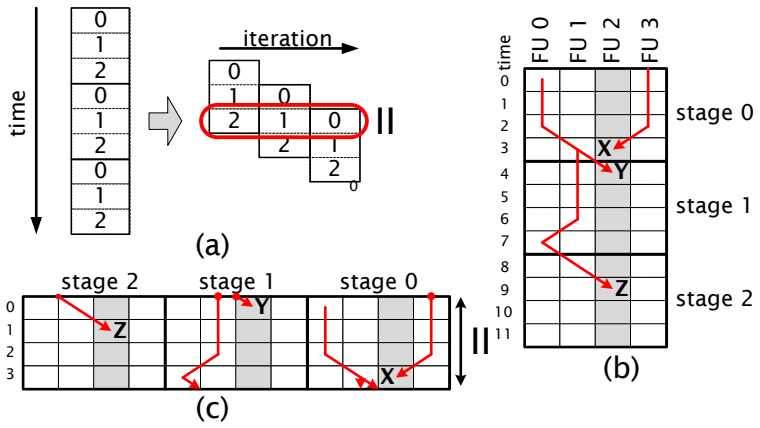


Figure 5.5: Modulo scheduling basics: (a) Concept, (b) An example mapping for FU 2, (c) Kernel mapping.

initial state is loaded and the token network can maintain the kernel state during the prologue.

5.3.3.2 Migrating Staging Predicates into Token Network

As previously mentioned, staging predicates are used to fill and drain the pipeline by selectively enabling operations to fill and drain the pipeline. A staging predicate is assigned to each stage of the schedule and it becomes true when current stage is activated in the pipeline. Staging predicates are routed through the predicate network in the datapath and separate configurations are required to manage the routing. Nearly 15% of the configuration bits are used for routing staging predicates in modulo scheduled loops. The kind of information carried with staging predicate is actually control data, hence its inefficient to manage it in the datapath.

For this reason, we propose to migrate the staging predicates from the datapath into the control path. We can simply increase the size of tokens by 1 bit and use the

extra bit (valid bit) for the staging predicates. If a resource receives a token with the valid bit set, the incoming data is in the right stage and the operation mapped on the resource can execute. When a token terminates in a register file, it needs to store the valid bit in the register file so that the valid bit information can be retrieved when a token is re-generated later from the same register file. Therefore, RF token modules will include a 1 bit register file that has the same configuration as the original register file in the datapath.

There are several benefits to migrating the staging predicates. First, the configurations for routing the staging predicates in the datapath is not necessary anymore. The routing information of the valid bit in the control path is same as the token routing information, so no additional configuration is required. The second benefit is a performance gain for loops. Removing the staging predicates in the datapath also removes the staging predicate edges in the dataflow graphs. With less scheduling restrictions, the compiler can find better schedules for the same target loops. Also, the predicate network in the datapath is not used for routing staging predicates anymore and can be dedicated to support predicates for if-converted code. The overhead of this approach is mainly in the hardware side. The interconnect in the token network is increased by 1 bit and a 1 bit clone of each register file in the datapath is added to the RF decoders. Also, there is an encoding overhead for the activation stages for live-in values. The trade-off for migrating staging predicate will be discussed in Section 5.5.

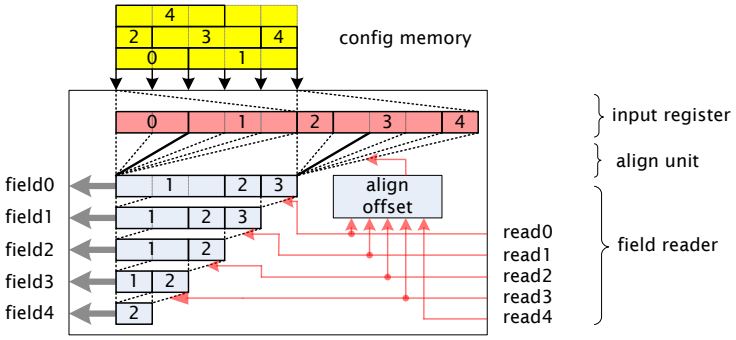


Figure 5.6: Decoder for fine-grained code compression

5.4 Configuration Memory Partitioning

The decoding logic for fine-grain code compression is shown in Figure 5.6. It is composed of three components: input register, align unit, and field reader. Encoded instructions are stored in the configuration memory as shown in the figure. Input register buffers each word line of the configuration memory and align unit makes sure that the instruction to be decoded is placed at the leftmost position in the field reader. Based the instruction format given(read signals in Figure 5.6), each instruction field is fetched in the field reader.

Obviously, having a giant 845-bit wide configuration memory is not a feasible design and also increases the complexity of the decoder drastically. Therefore, the configuration memory needs be partitioned and it needs be done in an efficient way that reduces the complexity of the decoder. Configuration memories are generally built with SRAMs and their power consumption is determined by the width of the memories. Partitioning the configuration memory into smaller SRAMs increases the total power consumption of all the SRAMs. This is because each individual SRAM has

its own peripherals and they add up to the total power consumption. When a single 128 bit-wide SRAM is partitioned into eight 16 bit-wide SRAMs, the total power consumption for reading 128 bit data increases 46% for the partitioned case than the original 128bit-wide SRAM. On the other hand, a small configuration memory has the benefit of decreased complexity for the decoder attached to it. For example, a decoder with 4 fields can be built with 22 MUXes, but doubling the number of fields require 71 MUXes. Therefore, having two decoders with 4 fields is 40% more efficient than one decoder with 8 fields. Therefore, partitioning the configuration memory needs be done efficiently considering the trade-off between the SRAM power consumption and the complexity of decoders.

Field Uniformity: When partitioning the configuration memory, it is also important to determine which fields are bundled together and stored in the same memory. Different widths in the same configuration memory increase the complexity of the align unit and introduce an encoding overhead with padding bits. Therefore, we allow only same type of instruction fields to be bundled together.

Sharing of Field Entries: The width of each partitioned configuration memory determines the maximum number of instruction fields that can be fetched in each cycle. Since the width of the memory is also related to the complexity of the attached decoder, we can optimize the decoder complexity by limiting the maximum instruction fields for a single cycle. For example, let's assume that 4 constant fields from four FUs are bundled together and stored in the same memory. The worst case scenario is that all 4 constant fields are used in the same cycle, and the decoder has to have

4 field entries. If the worst case rarely happens, we can limit the number of active constant fields in each cycle. For example, the memory can have only 2 entries and 4 constant instruction fields can share them. While only two constant fields can be active in the same cycle, the complexity of the decoder decreases. The trade-off here lies between the performance of the schedule and the decoder complexity. We learned that the average utilization of instruction fields varies from 10% to 80% depending on the type of instruction fields. For under-utilized fields, it is definitely beneficial to allow instruction fields to share field entries in the decoder.

Design Space Exploration: The design decisions in each component have trade-offs with other components. Thus, we performed a design space exploration to find a good partitioning of the configuration memory. The configuration memory was partitioned differing the bundling of instruction fields, the number of partitioned memories, and the sharing of field entries in a memory. Due to the space limitations, only the final result is shown in Section 5.5.

5.5 Experiments

In this section, we evaluate our control path design with the token network. we created four instances of the token network differing in multicasting capabilities and staging predicate and compared them with design (a) and (b) in Figure 5.2.

5.5.1 Experimental Setup

Target Architecture: The target CGRA architecture is a 4×4 heterogeneous CGRA shown in Figure 5.1. 4 PEs have load/store units to access the data memory and 6 PEs have multiply units. There is a 64-entry central register file with 6 read and 3 write ports wherein only FUs in the first row can directly read/write. All other FUs can only read from the central RF via column buses. The central register file is primarily used for storing live-in values from the host processor. There is also a predicate register file that has 64 entries and 4 read/4 write ports. Each FU has its own 8-entry local register file with one read and one write port. Local register files can be also written by FUs in diagonal directions (upper right/upper left/lower right/lower left). For example, local RF in PE 5 can be written by FUs 0, 2, 5, 8 and 10 and only FU 5 can read from it.

Target Applications: For performance evaluation, we took 214 kernel loops from four media applications in embedded domains (H.264 decoder, 3D graphics, AAC decoder, and MP3 decoder). The loops, varying in size from 4 to 142 operations, were mapped onto the CGRAs and configurations were generated by the compiler. The performance is measured by the average throughput of all 214 loops for each control path design.

Compiler Support: We developed a modulo scheduler that can supports our control path restrictions. First, the compiler makes sure that a value generated in an FU or a register file read port can be consumed up to two neighboring resources to meet the two destinations limit. Also, the compiler actively limits the number of

field type	opcode	dest	const	crf_read	crf_write	ldrf	control
memory configurations	2x(8, 8)	4x(8, 5), 4x(8, 6)	2x(6, 10)	1x(11, 9)	1x(6, 7)	4x(6, 3)	1x(1,68)

(a)

					power (mW)				area (mm^2)			
design	m	v	# bits	perf	sram	dec	token	total	sram	dec	token	total
baseline	1	0	845	100.0	104.0	5.4	0.0	109.4	0.539	0.015	0.000	0.554
static	0	0	647	98.5	56.4	18.2	0.0	74.6	0.412	0.120	0.000	0.532
token 0	0	0	485	98.5	31.9	16.5	3.5	51.9	0.309	0.109	0.030	0.448
token 1	1	0	606	99.6	37.2	22.2	3.5	62.9	0.386	0.139	0.029	0.555
token 2	0	1	456	103.8	27.2	17.1	4.8	49.1	0.291	0.113	0.048	0.452
token 3	1	1	567	105.4	30.6	23.1	4.7	58.4	0.361	0.145	0.046	0.553

(b)

Figure 5.7: (a) Configuration memory partitioning, (b) Performance, power and area comparison of control path designs

active fields in each cycle as to the sharing degree of the configuration memories.

Power/Area Measurements: Area and power consumption were measured using the RTL Verilog model and synthesized with Synopsys design compiler using typical operation conditions in IBM 65nm technology. Power consumption was calculated using Synopsys PrimeTime PX. The SRAM memory power was extracted from data generated by the Artisan Memory Compiler. The model contained both the datapath and control path and was targeted at 200MHz. Our control path design with the fine-grain code compression decoder and the token network fits in a single pipeline stage between the configuration memory and the datapath, and it does not affect the critical path of the datapath.

5.5.2 Configuration Memory Partitioning

We performed a design space exploration for partitioning the configuration memory as explained in Section 5.4. The final result of the optimal partitioning is shown in Figure 5.7(a). The first row shows different types of instruction fields in our target CGRA and the partitioning result is shown in the second row for each field type.

Three numbers in each entry of the table indicate the followings: the number of configuration memories, the number of field entries in a memory, and the bitwidth of each field. For example, there are two memories for opcode fields and each memory has eight 8-bit field entries. Since the opcode fields are frequently utilized, the total number of field entries in the opcode memories is equal to the number of FUs. This means that all 16 FUs can be activated at the same cycle. On the other hand, there are only 12 field entries for const fields(2 memories with 6 field entries). So, only 12 FUs can utilize the const fields at the same cycle. In addition to the configuration memories for instruction fields, the control memory in the last column is created to manage the behavior of the token network. The control memory has the token generation information as explained in Section 5.3.2.1 and read signals for other configuration memories.

In the original control path design shown in Figure 5.2(a), 845 bits of configurations are distributed in 7 configuration memories(six 128-bit memories and one 77-bit memory) with 128 word lines. In our partitioning scheme, there are 19 configuration memories and the total width of them is 881 bits. Even though the total bits of all the configuration memories has slightly increased, these memories are less frequently accessed since the code size is decreased. Therefore, we can achieve the power reduction in the control path. Also, the increased code efficiency decreases the memory requirements and the number of word lines in each memory can be reduced, resulting in area reduction of the SRAMs. When compared to a naive partitioning scheme where configuration memories are partitioned for each PE, our optimal partitioning

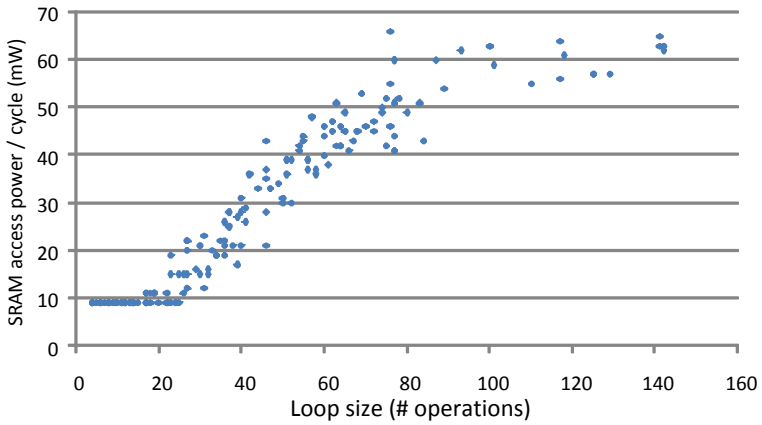


Figure 5.8: Cache effect on SRAM power consumption

achieves 22% power reduction and 33% area reduction for the decoder, while the performance degradation due to sharing of field entries is less than 1%.

5.5.3 Token Network Evaluation

Six control path designs were evaluated for performance, area, and power consumption and the results are shown in Figure 5.7(b). *baseline* design is the conventional control path of CGRAs that has no code compression(Figure 5.2(a)). *static* design employs a fine-grain code compression, but the instruction format is statically encoded in the configuration memory as shown in Figure 5.2(b). For control path designs with the token network, we created four instances that differ in multicasting capability and staging predicate support. The second column in Figure 5.7(b) indicates whether the design allows only two destination fields for each datapath component or allows multicasting as *baseline* design. The third column shows if the control path design contains valid bit network to support staging predicates(Section 5.3.3.2). For each control path design, the average number of configuration bits per cycle for all the

target loops are shown in the fourth column. The performance of each design is normalized to the performance of *baseline* and shown in the fifth column. The rest of the table contains power consumption and area of the designs. The control path is broken down into three categories(SRAM, decoder, and token network) and each category's power and area are shown separately in the table. *baseline* and *static* don't have a token network and the decoder in *baseline* is composed of only a pipeline register between the configuration memory and the datapath. For other designs, the pipeline register is included in the decoder(*static*) or in the token network(*token* designs).

For the performance and the number of configuration bits, all 214 loops were evaluated and the average is shown in the table. The SRAM power in the table is the average power per cycle for all 214 loops. For power consumption of the decoder and the token network, an average activity equivalent to the average utilization of FUs(55%) was assumed. The area of SRAMs for each design was calculated based on the amount of configurations required for the loops in MP3 decoder which require 128 word lines in *baseline* design.

Fine-grain Code Compression: Comparison between *baseline* and *static* designs reveals that the fine-grain code compression can improve both power consumption and area of the control path with increased code efficiency. Overall, the power consumption was reduced by 32% and the area decreased by 4%. There is a small performance degradation of 1.5% due to the sharing of field entries and lack of multicasting capability.

We can notice that the SRAM read power reduction ratio(46%) is greater than

the reduction ratio in the number of configuration bits(24%). This is due to the cache effect of the input register in the decoder(Figure 5.6). If all the configurations of a single loop can fit in the input register(two word lines in the configuration memory), the SRAM access happens only at the beginning and the content in the input register does not change throughout the execution of the loop. This occurs quite often when fine-grain compression is applied especially for less frequently used fields such as const fields or predicate fields. In *baseline* design, this cache effect is only achieved when loops are scheduled at II=1 (only 5% in our target loops).

Figure 5.8 shows the overall cache effect for the 214 target loops. X-axis shows the number of operations in each loop and Y-axis shows the average SRAM read power per cycle for each loop. In this figure, SRAM access power for instruction formats is not included. For small loops, the SRAM power is greatly reduced since most of the configurations can fit in the input register. As the size of a loop increase, the cache effect is minimized and the SRAM access power increases.

Among the average configuration bits of 647 in *static* design, the instruction format takes 172 bits and it needs be read from the memory every cycle. The power consumption of reading instruction format alone is 24.6 mW, which is almost one-third of the total power consumption in the control path. So, there is potential for further enhancing the control path design in the instruction format.

Token Network: We can evaluate the token network by comparing *token 0* to *static*. The only difference between two designs is how the instruction format is discovered. In *token 0* design, the token network is added for dynamic discovery of the

instruction format. The overhead of the token network is relatively small, introducing only 3% and 5% of *baseline* design's power consumption and area, respectively. However, introducing the token network improves all three features of the control path: code efficiency, power consumption, and area. *token 0* design further reduces the power consumption by 31% over *static* design and by 53% over *baseline* design. The area of the control path also decreases even with the overhead of the token network since the instruction format is no longer stored in the configuration memory.

To evaluate the limitation of two destinations, we created *token 1* design by adding multicasting capability in *token 0* design. To enable multicasting, each destination field is extended to a bit vector whose width equals to the number of destinations. While there is a small performance gain of 1.1%, the lengthened destination fields lead to poor code efficiency and the power consumption increases by 21%.

An interesting result can be found with migrating the staging predicates into the control path. A valid bit was added to the token networks of *token 0* and *token 1* designs to create *token 2* and *token 3* designs, respectively. Although there is some overhead for having valid bits in the token network, this overhead is mitigated by the improvements in the SRAM, and the overall power consumption and area decrease. This is because the configuration bits for routing staging predicates are not necessary anymore and the code efficiency improves. Moreover, there is a performance improvement of 6% in both cases of *token 2* and *token 3*. By removing staging predicate edges in the dataflow graph, scheduling restrictions are lessened and the chance of the compiler's finding a better schedule increases.

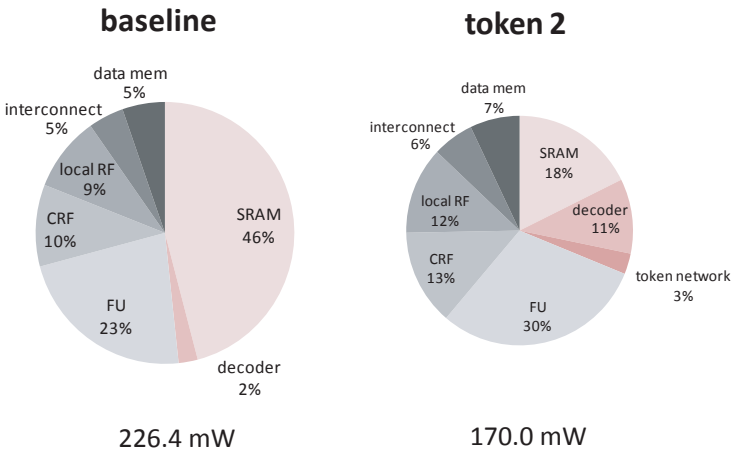


Figure 5.9: Power breakdown of *baseline* and *token 2* designs for a kernel loop in H.264

System Power Consumption: From the results in Figure 5.7(b), we concluded that *token 2* design is the most efficient control path design for our target CGRA. When compared to *baseline* design, the power consumption was improved by 56% and the area was decreased by 19%. Even with the limitation of two destinations, migrating staging predicates into the control path provides the overall performance improvement of 3.8% over the *baseline* design. Figure 5.9 shows the comparison of the power consumption of the system including the control path and the datapath, with two control path designs of *baseline* and *token 2*. Power was measured by running a kernel loop in H.264 that was scheduled at II=5. The overall utilization of the FUs for this loop is 61%. The numbers at the bottom indicate the overall power consumptions of two designs. When the token network is introduced, the portion of the control path power decreases from 48% to 35%, and the overall system power is decreased by 25%.

5.6 Summary

This chapter proposes a new control path design for CGRAs that utilizes the concept of a token network in dataflow machines for fine-grain code compression. The datapath is cloned to create a token network where tokens are flowing to discover the instruction formats. A design methodology for the control path with a token network is provided and an optimized solution was found through design space exploration. The resulting control path reduces the control power consumption by 56% while enabling a performance gain of 4%. Also, the area of the control path decreases by 19% since the configuration memory requirement is lowered with better code efficiency. Overall, our new control path design achieves a 25% saving in the system power consumption.

CHAPTER 6

Polymorphic Pipeline Array

6.1 Introduction

The PPA design is inspired by coarse-grain reconfigurable architectures (CGRAs) that consist of an array of function units interconnected by a mesh style interconnect [38, 39].

This chapter offers the following three contributions:

- An analysis of the available parallelism and its variability in three media applications (MPEG4 audio decoding, MPEG4 video decoding, and 3D graphics rendering).

- The design, operation, and evaluation of the PPA - a customizable media accelerator for mobile computing.

- A virtualized modulo schedule that can execute innermost loops with a run-time varying number of PPA resources assigned to it.

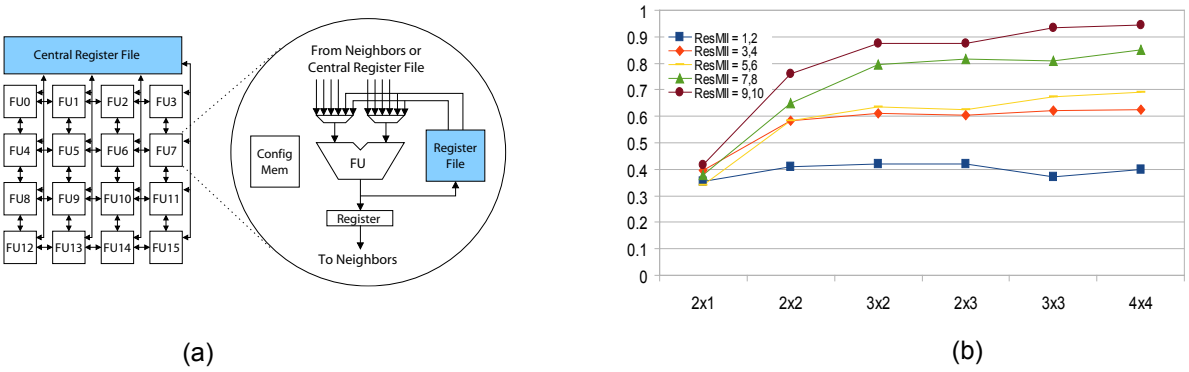


Figure 6.1: (a) CGRA loop accelerator, (b) Impact of the array size on the performance

6.2 Analysis of Multimedia Applications

In this section, we examine three applications from different multimedia domains that are expected to be used in mobile environments: audio decoding, video decoding and 3D graphics acceleration. We first examine the different levels of available parallelism, its variability over the run of the application, and finally and suggest some high-level architectural choices to achieve a high single-thread performance in power-constrained systems. The applications consist of:

- AAC decoder: MPEG4 audio decoding, low complexity profile
- H.264 decoder : MPEG4 video decoding, qcif profile
- 3D : 3D graphics rendering accelerator

As a baseline media accelerator, a coarse-grain reconfigurable architecture (CGRA) similar to ADRES [38] (Figure 6.1(a)) is used. ADRES consists of 16 function units (FUs) interconnected by a mesh style network. Register files are associated with each

FU to store temporary values. The FUs can execute common word-level operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to increase hardware efficiency. As a result, they have short reconfiguration times, low delay characteristics, and low power consumption. With a large number of computing resources available on CGRAs, loop level parallelism can be exploited by software pipelining compute intensive nested loops.

ADRES can also function as a VLIW processor to execute acyclic and outer loop code. The first row of FUs and the central register file provide VLIW functionality, while the remaining three rows of FUs are de-activated for non-innermost loop code. Other CGRAs simply execute non-innermost loop code on the host processor and de-activate the entire array. ADRES provides a higher performance option by eliminating slow transfer of live-in values between the host and the array as well as dedicating more functional resources to the acyclic code than a typical host processor would have.

In this analysis, we first investigate how much fine-grain parallelism resides in the benchmarks with software pipelining of the compute intensive innermost loops. Then, we take a look at opportunities for exploiting coarse-grain pipeline parallelism.

6.2.1 Fine Grain Parallelism

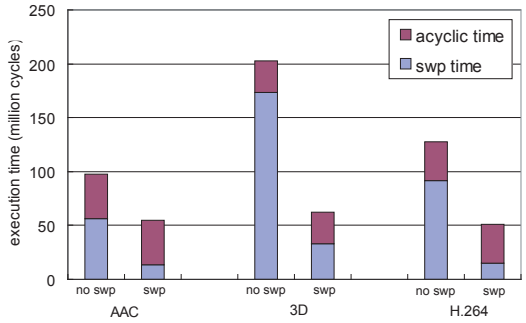
Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. They can be efficiently accelerated by using software pipelining that can increase the throughput of the innermost nest by overlapping

the executions of different iterations. For our three target benchmarks, we analyzed how much fine-grain parallelism resides in each benchmark by looking at the number and the execution time of loops that are software pipelineable. Figure 6.2(a) shows the number of total loops and the number of software pipelineable loops in each benchmark. The execution time breakdown between software pipelineable and the remainder of the code is shown in Figure 6.2(b).

We implemented a modulo scheduler taking the concept of [46]. Modulo scheduling is an efficient software pipelining technique that exploits loop level parallelism by overlapping the execution of different iterations. Each bar in Figure 6.2(b) shows the breakdown of execution time spent in the software pipelineable regions (swp time) and the rest of the application (acyclic time). The left bar of each benchmark in Figure 6.2(b) is the breakdown of execution time spent when only the VLIW processor (top row of the CGRA accelerator) is used for the whole application (no software pipelining), while the right bar shows the breakdown when swp regions are executed on the entire CGRA. First, we can see that there are many opportunities for exploiting fine-grain parallelism in the benchmarks. On average, 35% of loops are software pipelineable and 71% of execution time is spent in swp regions. When the CGRA accelerator is employed to map the software pipelineable loops, there are great performance gains for all three benchmarks of 1.76, 3.25, and 1.48, respectively for AAC, 3D, and H.264. So, it is very important for multimedia applications to exploit fine-grain parallelism inherent in them, and CGRAs are effective platforms executing such loops.

	total loops	swp loops
AAC	102	36
3D	260	83
H.264	269	81

(a)



(b)

Figure 6.2: (a) Number of software pipelineable loops, (b) Breakdown of execution time for software pipelineable region and acyclic region

An interesting question at this point is how we improve the performance even further when more resources are available in an embedded system. One possible solution is scaling the accelerator to a bigger array. By introducing more resources into the accelerator, we can possibly reduce the execution time spent in swp regions. To assess the impact of scaling the accelerator, we took all the swp regions in the three benchmarks and mapped them onto accelerators with various array sizes. First, we categorized the software pipelineable loops into groups based on the number of instructions and measured how the average throughput of each group changes as the size of the CGRA increases. The array sizes of the CGRA are shown in the X-axis of Figure 6.1(b), and the scaled throughput on the Y axis. Throughput is normalized to the theoretical upper bound of each loop when mapped onto the 4x4 array.

Here, we can notice that the throughput saturates as we increase the size of the CGRA. Even for the biggest group, the throughput does not increase that much beyond the size of 4x4. Moreover, the execution time spent on swp loops is relatively small when all the software pipelineable loops are mapped onto the accelerator as

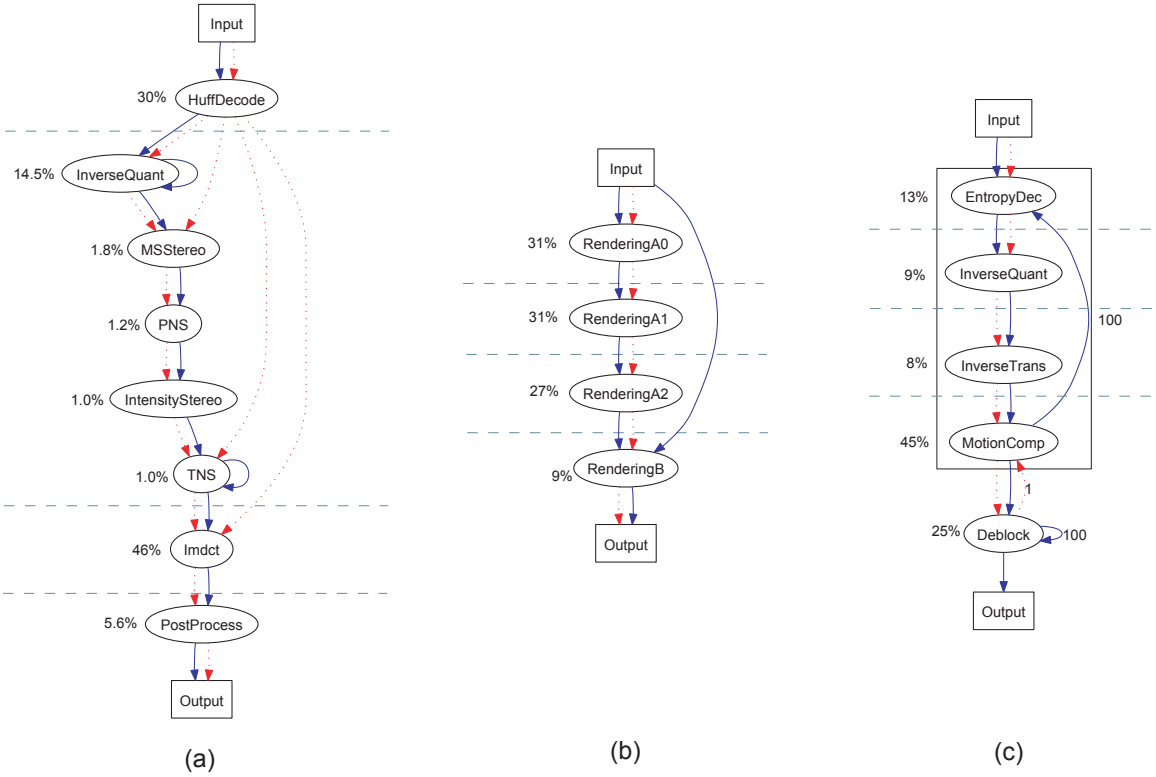


Figure 6.3: Task Graphs: (a) AAC, (b) 3D, (c) H.264, nodes represent tasks, solid edges show control flow, and dotted edges show data transfer

shown in Figure 6.2(b). For H.264, the execution time of swp region is only 20% of the total time after accelerating them. By Amdahl's law, we need to find a way to increase the performance of acyclic region to further increase the overall performance of these applications.

6.2.2 Coarse-Grain Pipeline Parallelism

In addition to fine-grain parallelism, coarse-grain pipeline parallelism is also available in these applications due to their streaming nature [20]. Figure 6.3 shows the task graphs of the target benchmarks. Solid lines indicate control flow edges, while

data communications between tasks are shown as dotted lines. Data enters the input node in a task graph and goes through various computing kernels represented as oval nodes, and it finally exits at the output node. After a packet of input data is processed, the next data packet can be processed in the same manner. Hence, there is an implicit outer loop around these task graphs that loops over input data packets. Coarse-grain pipeline parallelism can be extracted when the task graph can be split into multiple stages that communicate in a feed-forward fashion and without any inter-iteration dependences contained within a single stage. By mapping stages into different pieces of hardware, the execution of the outer loop iterations can be pipelined and the overall throughput can be increased. Stages can consist of loops as well as acyclic blocks of code, hence parallelism of this form is not limited to innermost loops. The amount of execution time in each stage is annotated next to the nodes. Here, we assume no accelerator in the system and only a VLIW processor (first row of the CGRA) is used.

Based on the execution time ratio, we can partition the task graph into pipeline stages as shown as dotted horizontal lines in Figure 6.3. For AAC(Figure 6.3(a)), there are two nodes that have high execution time ratio: HuffDecode and Imdct. So, the task graph can be partitioned in a way that the two compute intensive nodes are isolated. 3D has two major kernels that perform a rendering process: RenderingA and RenderingB. Originally, RenderingA is encapsulated in a loop with three iterations, but we fully unrolled to get the task graph in Figure 6.3(b). Pipelining for H.264 is a bit tricky since there is a data dependency between outer loop iterations. The

motion compensation kernel (MotionComp in Figure 6.3(c)) uses the output of the deblocking filter(Deblock) from the previous iteration. Because of this dependency, the execution of the outer loop iterations need be performed in a sequential manner, preventing pipelining. However, we could find an opportunity for pipelining in an inner loop in the solid box. Each frame in H.264 is broken into macroblocks and they are individually processed in the inner loop. The iteration count of the inner loop is quite large (100) and 75% of execution time is spent in this loop. So, we can partition the task graph as shown in Figure 6.3(c).

When pipelining the three benchmarks with the stages shown in Figure 6.3, the performance gain of AAC, 3D, and H.264 are 2.09x, 3.11x, and 1.93x, respectively. Indeed, the coarse-grain pipeline parallelism can expose a great deal of performance gain, but the opportunities for fine-grain parallelism are still available. The stages that are limiting the overall throughput of the pipeline can be accelerated if fine-grain parallelism is exploited. For example, Imdct in AAC can run 4.7x faster if it is mapped on the full CGRA. Similarly, RenderingA in 3D and MotionComp in H.264 can run 3.9x and 2.4x faster, respectively.

6.2.3 Computation Variance

Another important behavior to characterize in these applications is the dynamic variance in the computational requirements. This metric is important because it indicates whether a static apportioning of resources would yield predictable execution times and utilization of the hardware. Conventional wisdom is that processing time

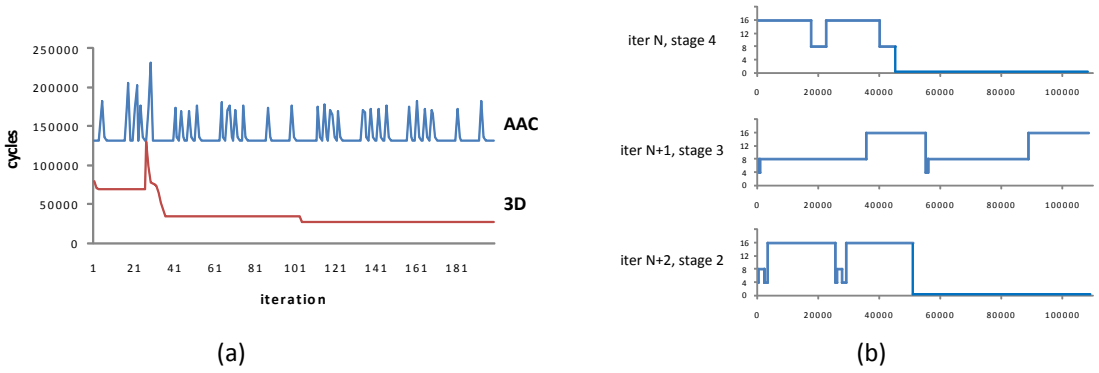


Figure 6.4: Execution Pattern Variation in Coarse-Grain Pipelining: (a) Stage Execution Time, (b) Resource Requirements

is relative constant for media applications, e.g., each iteration of a loop might operate on the row of an image.

Figure 6.4(a) shows the execution time for one stage in AAC and 3D over the first 200 frames (outer loop iterations). The x-axis is the iteration number and the y-axis is the execution time in cycles on a 4x4 CGRA. As shown, execution time is not constant. In fact, there is a large variation in execution time. AAC regularly oscillates between 150k and 200k cycles, while 3D starts off high and gradually becomes less. This behavior is due to several factors. First, there is an abundance of control flow in these applications that changes the amount of processing required. Second, there is some predictable regularity to the behavior. For example, frame of different types occur at regular intervals and require relatively constant processing time. Finally, in 3D, the processing time levels off after the initial startup. Again, such behavior is not constant, but is predictable.

To view the variability in a different manner, Figure 6.4(b) shows the resource

requirements for 3 consecutive coarse grain stages from AAC over time. The x-axis is cycle number and the y-axis is the number of resources (function units apportioned as 2x2 units) that achieves the best performance. As can be seen, resource requirements change during the execution of a single frame. For the top and bottom stages, the resource requirements are dramatic, going from 16 to near zero. In general, resources are allocated based on a worst-case scenario. In this case, each stage would require 16 resources. But, the utilization will be very poor with this approach. Rather, this behavior indicates that idle resources could possibly be loaned to neighboring stages or that shared resources could be designated.

6.2.4 Summary and Insights

The analysis of these media applications provides several insights. First, multimedia applications are rich in both fine-grain and coarse-grain pipeline parallelism. Further, these forms of parallelism are not mutually exclusive. Rather, they can cooperate to eliminate the opportunities that were left out when only one of them was exploited. Pipeline parallelism can accelerate the entire application including acyclic regions, while fine-grain parallelism can accelerate the pipe stages that limit the overall throughput of the pipeline. These are typically innermost loops with large bodies and/or high trip counts. Second, resource requirements not only vary statically across different pipeline stages, but also dynamically both during the processing of a single frame of data and across different frames. Dynamic partitioning of resources is thus necessary to achieve high performance and utilization.

A central challenge is how to allocate finite resources across different pipeline stages. Pipeline stages have different potentials for fine-grain parallelism. For example, the HuffDecode kernel in AAC and Deblock in H.264 are inherently sequential and putting more resources will not improve the performance. Conversely, Imdct can greatly benefit with more resources by exploiting fine-grain parallelism. Also, the high dynamic variance in computation continually changes the resource requirements. For real time, worst-case execution times are often used. But, in these applications, worst-case will grossly exaggerate the number of needed resources. The conclusion is that a flexible execution substrate that facilitates changing the resource allocation over time is necessary.

The rest of the paper is organized as follows. First, we propose a flexible multicore accelerator design that can exploit both fine-grain and pipeline parallelism. Then, we describe the hardware extensions and compilation techniques that enable dynamic resource partitioning. A virtual schedule is constructed that can properly function under variable resource allocations. An experimental evaluation including performance and power, followed by related work and conclusion constitute the last three sections of the paper.

6.3 Polymorphic Pipeline Array

6.3.1 Overview

The Polymorphic Pipeline Array (PPA) is a flexible multicore accelerator for embedded systems that can exploit both fine-grain parallelism found in innermost loops and pipeline parallelism found in streaming applications. The PPA design is inspired by CGRAs but with extensions for both static and dynamic configurability. A PPA consists of multiple simple cores that are tightly coupled to neighboring cores in a mesh-style interconnect. A PPA with eight cores is shown in Figure 6.5(a). There are a total of 32 processing elements (PEs) in this PPA, each containing one FU and a register file. Four PEs are combined to create a core that can execute its own instruction stream. Each core has its own scratch pad memory and column buses connect four PEs to a memory access arbiter that provides sharing of scratch pad memories among cores.

The main characteristics of PPA can be summarized as follows:

- Simple and distributed hardware: The resources are fully distributed including register files and interconnect. Also, there is no dynamic routing logic. All the communications are statically orchestrated by compiler.

- Fast inter-core communications via direct connections between register files

- Each core can have its own instruction stream. So, the host processor can offload an entire application, not just the innermost loops, onto the PPA.

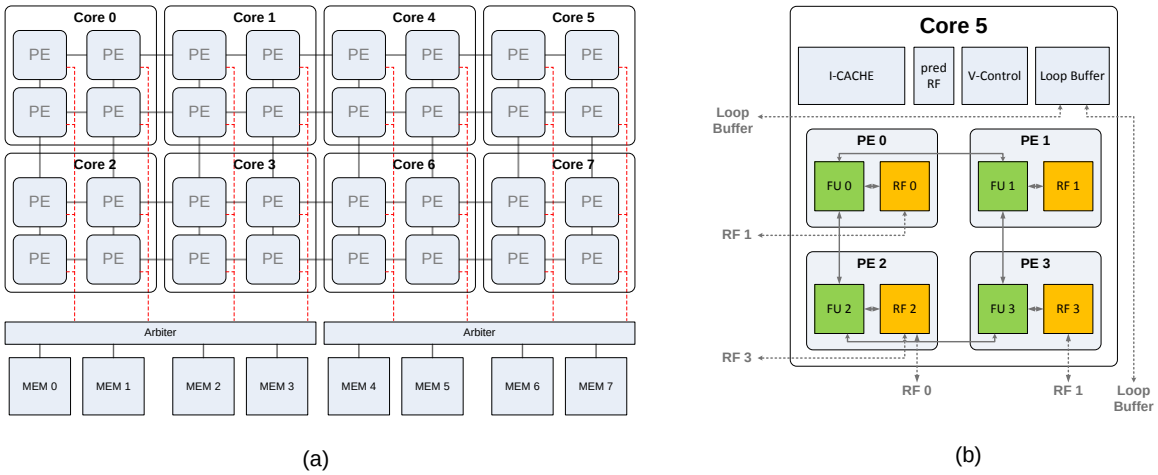


Figure 6.5: PPA Overview: (a) PPA with 8 cores, (b) Inside a single PPA core

- Cores can be combined to create a larger logical core to exploit the available fine-grain parallelism in large loop bodies.
- Virtualized execution: PPA can adapt to fluctuating resource availability and dynamically partition the array during the execution

6.3.2 Core Description

Inside a Core: Figure 6.5(b) shows a detailed diagram of a single PPA core. There is an instruction cache and a loop buffer. A loop buffer is a small SRAM that stores instructions for modulo scheduled loops. A loop buffer minimizes the instruction fetch power for high density code of loops. Each PE contains a 32 bit FU and a 16 entry register file with 3 read/3 write ports. Four PEs in a core share a 64 entry central predicate register file with 2 read/3 write ports, but there is no central register file for data. All FUs can perform integer arithmetic operations and

one FU per core can do multiply operation. A simple mesh network connects the FUs in a core. Register files are accessed by the same topology of mesh interconnect(not shown in the figure). All the FUs can read/write from the central predicate file.

Inter-core Connectivity: Inter-core interconnect is shown in dotted lines in the figure. There are three types of inter-core interconnect in a PPA: RFs, predicate RF, and loop buffer. Direct connections between neighboring RFs in different cores allow fast inter-core communications. These RF-to-RF connections can be utilized when a loop is mapped onto multiple cores. We found the direct connections between register files more efficient than FU connections especially for virtualization (discussed later). Each predicate register file can be accessed by an FU in the neighboring cores. Predicate register files are used for inter-core communications such as hand-shaking for coarse-grain pipelining and resource availability for virtualized execution. Finally, loop buffers can transfer instructions to the neighboring cores also for virtualized execution. The hardware components for virtualized execution is explained in details later in Section 6.4.

Memory System: For memory accesses, a memory bus connects FUs in the same column to the memory access arbiter, allowing only one memory access per cycle for FUs in the same column. A memory arbiter has three memory sharing configurations and provides different load latencies when multiple scratch pad memories are shared among FUs in different columns. The sharing modes for a memory arbiter is as follows.

- No sharing : FUs can access the memory in the same column only. Load latency

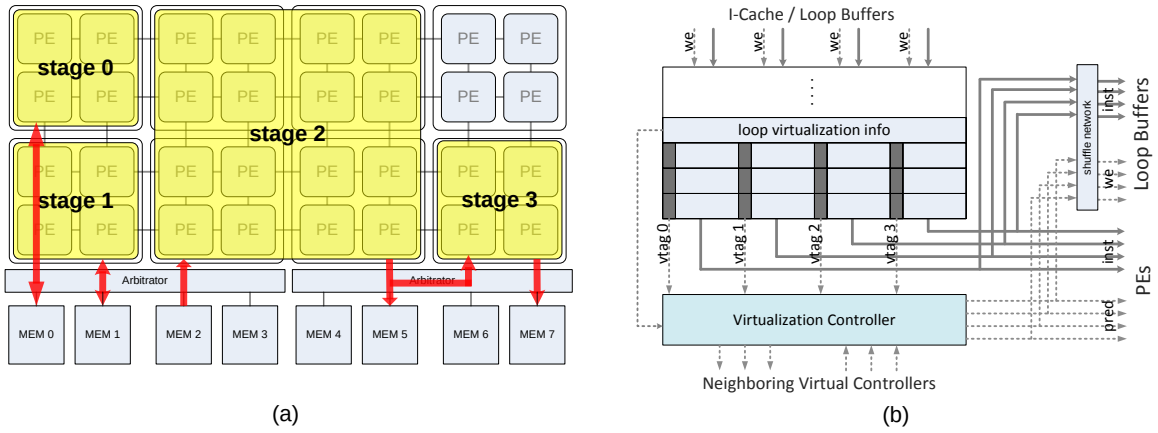


Figure 6.6: (a) An example of PPA running AAC in a pipelining fashion, (b) Virtualization Controller

is 2 cycles.

- Sharing of 2 : FUs in two columns can share memories in the same columns. Load latency is 4 cycles.

- Sharing of 4 : FUs in four columns can share memories in the same columns. Load latency is 6 cycles.

For example, when the arbiter operates in the sharing 4 configuration, all the FUs in four cores (i.e., cores 0, 1, 2, and 3) can access the four memories below them (mem 0, 1, 2, and 3) with a load latency of 6 and up to 4 memory accesses can be made per cycle. Memory sharing only occurs when cores are combined to create a bigger logical core for software pipelining of loops. The increased load latency is not really a big issue since software pipelining can often hide the long latency of operations. Memory sharing can also be used to form a bigger logical memory when memory requirement is high, behaving as a banked memory system.

6.3.3 Supporting Coarse-Grain Pipeline Parallelism

Figure 6.6(a) shows how the applications can be accelerated using different static resource partitions for a PPA with eight cores. Based on the analysis in Section 6.2, we provided the possible mapping of AAC on the PPA shown in Figure 6.5(a). For the stages with a high ratio of acyclic regions (not software pipelineable), a single core is allocated for execution. Stage 0 performs a Huffman decoding that is very sequential and one core is assigned to this stage. The memory requirement is not high in AAC, so all the memories operate in the no sharing mode. Bold solid lines in the figure show the stages that access memory. When a stage finishes processing data, the output of each stage is transferred to the next stage's memory by a DMA engine. DMA transfers can be omitted when memories are shared by the arbiter. For example, stage 2 can read the input from MEM 2 and write the processed data in MEM 5 that can be accessed by stage 3 directly. Even though memory sharing increases the load latency, both stage 2 and stage 3 contain high ratio of swp region that can tolerate the latency overhead.

6.3.4 Supporting Fine-Grain Pipeline Parallelism

The abundance of computation resources makes PPA an attractive solution for exploiting fine-grain parallelism. When there is large amounts of fine-grain parallelism in a inner-most loop, multiple cores in the PPA are merged together to create a bigger logical core. In the logical core, one core behaves as master and orchestrates the execution of all the participating cores in lock step.

Static Partitioning: The PPA array can be partitioned statically based on the resource requirements of each coarse-grain pipeline stage. For example, the third stage of AAC contains a large number of compute intensive filters and the whole application spends 71% of execution time in this stage. Therefore, a 2x2 core array is assigned to the third stage. On the other hand, the first stage performs Huffman decoding that is inherently a sequential process. Also, the execution times of the other stages (second and fourth) are relatively small, thus accelerating these stage does not necessarily lead to overall performance increase. Therefore, all the other stages are assigned with a single core array. The benefit of static partitioning lies in the highly optimized schedules since each compute intensive kernel is scheduled targeting only one sub-array. However, this approach does not adapt to dynamically changing resources availability discussed in Section 6.2.3. When an application has a large variation in execution pattern, static partitioning can either result in low utilization of resources, or not be able to fully accelerate the overall performance when there is not enough resources available.

Dynamic Partitioning with Virtualization: Coarse-grain pipeline stages in multimedia applications have different execution patterns. As a result, the resource availability in the PPA fluctuates at run-time and it is crucial to adapt to different availabilities and maximally utilize them for improving the overall throughput of the applications. One approach is to statically generate a set of different schedules each of which targets different numbers of resources. For example, a loop can be scheduled for 1x1, 1x2, 2x1 add 2x2 PPA cores beforehand, and an appropriate schedule can be

selected at run-time depending on the availability of resources. Each schedule can be highly optimized since the target is fixed. However, the resulting code bloat prevents it from an attractive solution for embedded systems where minimizing code size is important.

The code bloating problem can be minimized through virtualized execution where a single schedule is converted into different schedules dynamically with regard to the changing resource availability. For virtualization, both compiler and hardware support are required. The compiler is responsible for generating schedules that can be easily converted at run-time (see Section 4). Then, the hardware can dynamically allocate resources and perform the conversion of schedules. However, the major down side is the sub-optimal scheduling result. Since the compiler has to target multiple sub-arrays, the scheduling result might not be as efficient as static partitioning approach. Also, there is run-time overhead for virtualization.

6.3.5 Hardware Support for Virtualization

The major challenges in hardware are how to migrate the schedule across different cores at run-time for virtualized execution and how to communicate with neighboring cores for checking the resource availability. For these purposes, a virtualization controller (VC) is implemented in each core (Figure 6.6(b)). Since each core has a loop buffer, the PPA can prepare for a virtualized execution by migrating particular sections of a schedule into neighboring cores from the owning core where the whole schedule is stored.

Each instruction in the loop buffer is tagged with two bits of information (virtualization tag). This information is used on two purposes. First, it tells to which core the instruction is migrated when resource allocation is changed at run-time. When more resources become available, the VC copies a subset of instructions to the neighboring cores through the connections between loop buffers. The loop buffer interconnect goes through a shuffle network that can change the orientation of the copied schedule. Depending on the location of the available core, the schedule needs to be flipped horizontally or vertically. Another use of vtag is predicating the execution of inter-core communications that only execute when the schedule is spread over multiple cores. The VC compares the current resource allocation status and the vtag, and generates a predicate input for the inter-core communication instructions.

6.4 Compiler Support for Virtualization

In this section, we present the compiler support for the PPA focusing on the virtualized execution of modulo scheduled loops.

6.4.1 Edge-centric Modulo Scheduling

In the PPA, all the communications including inter-core communications are orchestrated by the compiler. To utilize the abundant resources available in a PPA, an effective compiler technique is essential. The major challenge in scheduling with the presence of distributed hardware is in managing the communications among resources. Without any centralized resources, the communication is often the bottleneck

to achieve good performance, more so than the actual computation.

We have adopted the EMS [46] that emphasizes the routing of operands

We employ the Edge-centric Modulo Scheduling(Chapter 4)

on a distributed network to achieve high throughput of modulo scheduled loops. EMS can achieve 98% of the theoretical best throughput on array-style architectures. In our baseline accelerator(6.1), only the innermost loops are mapped onto the array and the remaining (acyclic and outer loop) regions are executed on the VLIW processor. Since we are offloading acyclic regions onto PPA as well as loops, we modified the EMS algorithm so that it can support both cyclic and acyclic regions.

6.4.2 How to Virtualize

Virtualization requires the compiler to generate a single schedule that can be dynamically mapped onto different target arrays. There are two approaches for converting schedules: folding and expanding. In both approaches, converting (transforming a schedule from one array to another) should be performed in a way that observes the following constraints:

- Modulo constraint: each resource in a converted schedule is used only once in every II cycles
- Register pressure: : virtualization should not drastically increase register pressure

- Dependency constraint: producer-consumer relation is observed in a converted schedule

6.4.2.1 Folding Approach

Figure 6.7(a) shows how the folding scheme works for 1x2 array(two cores). First, the compiler generates a schedule shown on the left. Here, the schedule is composed of two sections(A and B). Each section is divided into two sub-schedules for each core. A0 and B0 run on core 0, and A1 and B0 run on core 1. Two iterations of the target loop is shown in the figure; the light gray boxes show the first iteration, and the dark gray ones for the second iteration. In kernel state(shown as an empty rectangle), all the sub-schedules(A0, A1, B0, and B1) run in parallel across the two cores. When only a single core is available, the whole schedule needs to run on a single core and the original schedule is folded to create a narrower and longer schedule shown as A' and B'. The new schedule is created by interleaving the two sub-schedules cycle by cycle. For example, each operation at cycle N in A0 is placed at cycle 2N in A', and one at cycle N in A1 is placed at cycle (2N + 1) in A'. Cycle-wise interleaving is the only way to observe the dependency constraint in the new schedule without re-scheduling. The resource constraint is also kept naturally since (A0, B0) and (A1, B1) time-share the resources in a single core. Since A0 and B0 execute in parallel in the original schedule in the same core, the modulo constraint can be observed in the new schedule(also observed for A1 and B1).

The major downside of folding is the increased register pressure. Since the two

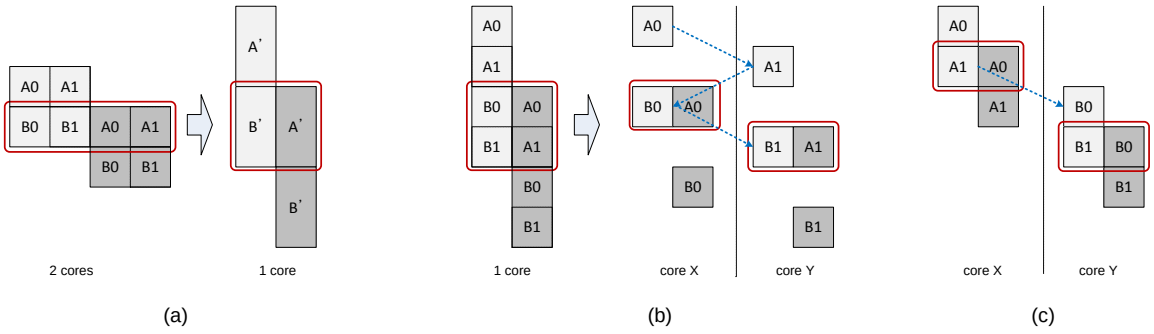


Figure 6.7: (a) Folding with interleaving, (b) Expanding with horizontal cut, (c) Expanding with vertical cut

sub-schedules are interleaved cycle-wise, the communications bypassing the register file in the original schedule need to be buffered for one cycle. They need either passing through a register file(requires re-scheduling) or buffering latches inserted for each interconnect(hardware overhead). Also, the register live ranges in each section overlap with the other section, further increasing the register pressure.

6.4.2.2 Expanding Approach

The expanding scheme starts with a schedule targeting a single core as shown on the left in Figure 6.7(b). Here, each section (A or B) is divided into two sub-sections(A0, A1 or B0, B1). The first expanding approach is pipelining each section across the two cores(shown on the right in Figure 6.7(b)). Basically, the original kernel schedule is cut horizontally in half and each half runs on a different core. In each core, the two sub-sections are running in parallel((A0, B0) or (A1, B1)). Since they were already running in parallel in the original schedule, the modulo constraint is naturally observed. The dependency constraint is also observed since operations

are placed in the original order. However, this approach results in frequent inter-core communications shown as the dotted lines in the figure. The communications across the sub-sections(i.g. A0 and A1) incur the register copy operations via RF connections.

Another expanding approach is cutting the original schedule vertically as shown in Figure 6.7(c). This approach pipelines each section within a single core, rather than across two cores. Again, the dependency constraint is naturally observed. Also, the inter-core communications are limited to the section boundary(between A1 and B0). The register pressure does not increase since the consecutive sub-sections are running back-to-back in a single core. The major challenge in this approach is the additional modulo constraints between sub-sections((A0, A1) or (B0, B1)). For example, A0 and A1 were running sequentially in the original schedule, but they run in parallel in the new schedule. Therefore, there is no guarantee that two sub-sections have exclusive resources usage in a single core.

Finally, there is an approach that converts the original schedule in Figure 6.7(b) into the left schedule of Figure 6.7(a). This cannot be done easily due to the dependency constraint of A0 and A1. Since there can be producer-consumer relations between A0 and A1, they cannot run in parallel without re-scheduling.

6.4.3 Virtualized Modulo Scheduling

Based on the observations in the previous section, we propose Virtualized Modulo Scheduling that takes the expanding approach with vertical cut in Figure 6.7(c).

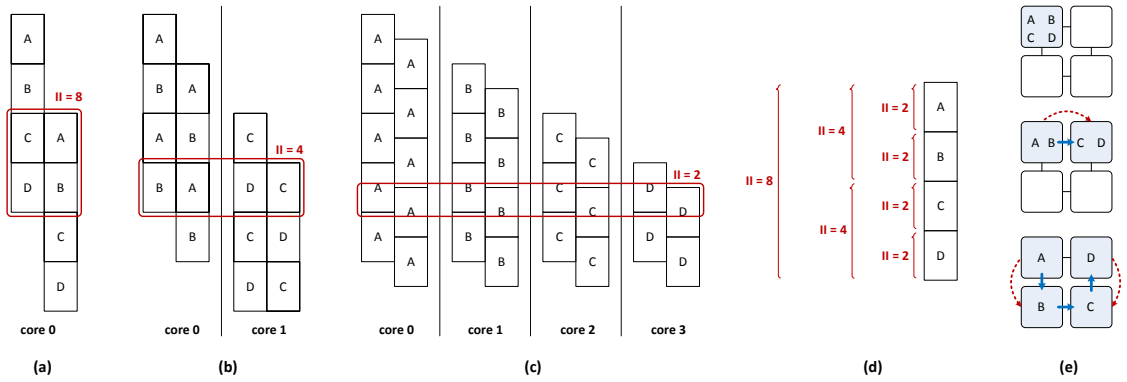


Figure 6.8: (a) Execution in a single core, (b) Execution in two cores, (c) Execution in four cores, (d) Multi-level modulo constraints, (e) Code expansion

This approach has no hardware overhead of buffer latches in folding and less inter-communication over horizontal expanding. Also, the register pressure is minimal compared to both of them. We proceed the discussion on VMS with a running example of a schedule that can be mapped onto three different target arrays: 1x1, 2x1, and 2x2 , shown in Figure 6.8. The major challenges for VMS are as follows.

- How to minimize the inter-core communication overhead
- How to impose the modulo constraints in different levels
- How to determine the IIs for different levels
- Register allocation

6.4.3.1 Minimizing Inter-core Communication Overhead

In VMS, a single iteration in a schedule targeting a smaller array is divided into the same number of sections as the number of cores in a bigger array. When the schedule

is expanded at run-time, each section is individually pipelined in a single core. For example, a loop in Figure 6.8(a) shows a schedule for 1x1 array. This schedule can be dynamically converted into schedules for 1x2 and 2x2 arrays at run-time. Since it can be mapped onto up to 4 cores, the whole iteration is divided into four sections(A, B, C, and D). When it is mapped onto a 1x2 array, A and B run on core 0, and C and D run on core 1(Figure 6.8(b). Each section will be mapped to an individual core when 2x2 array is available(Figure 6.8(c)). Therefore, the inter-core communications can occur only at the section boundaries. Since they can only use the limited inter-core interconnect and the live register values need be transferred across the cores, it is important to minimize the communications across the section boundaries.

For this purpose, the dataflow graph of the target loop is partitioned into four clusters minimizing the number of edge cuts. This is a traditional min-cut problem. Each edge-cut denotes inter-core communications through the interconnect or a register transfer. For the register value transfer, we implemented direct connections between register files across the cores. With this direct connectivity, register value transfer can occur without wasting the existing computation resources with move operations.

6.4.3.2 Multi-level Modulo Constraints

The biggest challenge in VMS is to enforce different levels of modulo constraints, so that no resource conflict occurs when the schedule is converted at run-time. Figure 6.8 shows a schedule that can be mapped onto different target arrays. Since

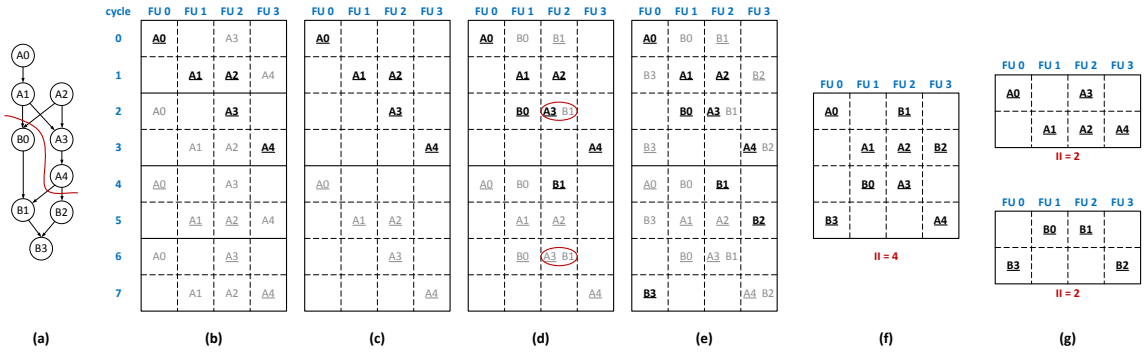


Figure 6.9: (a) Dataflow graph, (b) - (e) Mapping examples, (f) Modulo schedule for 1x1 array, (g) Modulo schedules for 1x2 array

there are three target arrays(Figure 6.8(a) - (c)), three levels of the modulo constrains are imposed to generate a schedule for 1x1 core as shown in Figure 6.8(d). In general, modulo constraints limit the number of available scheduling slots to (II x # resources). One might think that the additional modulo constraints can further reduce the number of available slots. In reality, the number of available slot stays the same since each level of modulo constraints has different scopes. The scopes of three modulo constraints are shown in Figure 6.8(d). For example, when scheduling the first section A, the scheduler needs to observe both II=8 and II=2 modulo constrains(B is not scheduled yet, so II=4 is not imposed yet). This reduces the number of slots to 8(2 x 4), but this is actually what is available in a single core when section A is individually pipelined in Figure 6.8(c). The scheduling slots that were limited by II=2 can be used when section B is scheduled, since the scope of II=2 is only valid for section A. When section B is scheduled, the modulo constraint of II=4 is imposed instead between section A and B.

Scheduling example: An example of scheduling with multi-level modulo constraints are shown in Figure 6.9. A dataflow graph is shown in Figure 6.9(a) and it is partitioned into section A and B. For illustration purposes, partitioning was performed arbitrarily(not min-cut partitioning). The target arrays are 1x1 array(1 core) and 1x2 array(2 cores) and target IIs are 4 and 2 for 1x1 and 1x2 array, respectively. First, section A is scheduled onto 1x1 array with II=4 and II=2. Figure 6.9(b) shows the scheduling result of section A on a single core with 4 FUs. Bold letters with underline show the actual placement of the operations. Gray letters with underline are occupancies due the to modulo constraint of II=4 and normal gray letters show occupancies due to the modulo constraint of II=2. After the scheduling of section A, only 12 slots are available(Figure 6.9(b)). However, when section B is scheduled, occupied slots due to modulo constraint of II=2 becomes available since the modulo constraint is limited to the section A. Figure 6.9(c) shows the available slots for section B. First, operation B0 is placed at FU1 in cycle 2(slot (1, 2)) in Figure 6.9(d). The II=4 modulo constraint marks slot (1, 6) as occupied and the II=2 modulo constraint makes slot (1, 0) and (1, 4) occupied. So, there is no resource conflict so far. When operation B1 is placed at slot (2, 4) as in Figure 6.9(d), II=2 modulo constraint marks slot (2, 2) and (2, 6) as occupied, but they are already occupied by A3. However, this is not a real resource conflict since II=2 modulo constraint is only valid for section B. II=2 modulo constraint becomes effective only when the schedule is expanded to 1x2 array where section A and section B run on different cores. The final schedule of section B is shown in Figure 6.9(e). Even though the II for a bigger

array is multiple of the smaller II in this example, the IIs don't have to be multiple of smaller one in reality. The constraints among the multi-level IIs are explained in the following section.

Code migration at run-time is also shown in Figure 6.8(e). Here, a schedule in a single core is expanded over to 2x2 array. Migration in this case is performed in two steps.

Determining Multi-level IIs: In conventional modulo scheduling, the minimum II is selected based on the number of available resources(ResMII) and the length of inter-iteration dependency cycles(RecMII). Starting from the minimum II, the scheduler increases the II until it finds a valid schedule. In VMS, scheduling is performed with multiple IIs as shown in Equation 6.1), where $\widehat{N}$ refers to the number of cores in a target array. Before starting the scheduling for virtualization, VMS generates test schedules for each target array without virtualization. The achieved II of each level ($TestII_k$ in Equation 6.3) determines the benefit of virtualization.

For the first level II, which targets the smallest array, ResMII and RecMII are calculated in a conventional way While the ResMII for level $\widehat{k}$ changes depending on the number of cores(C_k in Equation 6.2), the RecMII stays the same since it is not related to the number of resources available. The calculated ResMII and RecMII for each level define the lower bound of II to try(Equation 6.3). The lower bound is also determined by the II in the next level. When the II for a bigger array becomes greater than the II for a smaller array, there is no point of running the loop on a bigger array. Finally, the II(II_k in each level is limited to the achieved II of the previous

level (II_{k+1}) in the test run. This one also tests the benefit of the virtualization. The scheduling order of II sets is determined by the weighted summation of all the IIs(Equation 6.4).

$$IIs = (II_1, II_2, ..., II_N) \tag{6.1}$$

$$ResMII_k = ResMII_1 \ / \ C_k, \quad RecMII_k = RecMII_1 \quad (k > 1) \tag{6.2}$$

$$II_k \geq max(ResMII_k, RecMII_k), \ II_k > II_{k+1}, \ II_k < TestII_{k-1} \tag{6.3}$$

$$cost(IIs) = \sum_{k=1}^{N} \Big(w_k \times II_k) \tag{6.4}$$

Register Allocation with Multi-level IIs: Traditionally, register allocation is performed after scheduling, and spill code is inserted when the register requirement exceeds the register file capacity. Spilling in a highly distributed architecture like PPA is quite costly since it involves routing to/from the memory units and may require complete rescheduling of the loop. Moreover, spilling can easily happen due to the small size of the register files. For this reason, EMS performs register allocation during scheduling to avoid spilling and guarantee routability through the register files. We take the same approach of concurrent scheduling and register allocation in VMS. PPA supports rotating register files that implicitly copy the stored register values at II boundaries so that values can stay in the register file more than II cycles. Since VMS has multi-level of modulo constraints, register allocation needs be performed in a way that all the modulo constrains are observed inside the register files. To simply

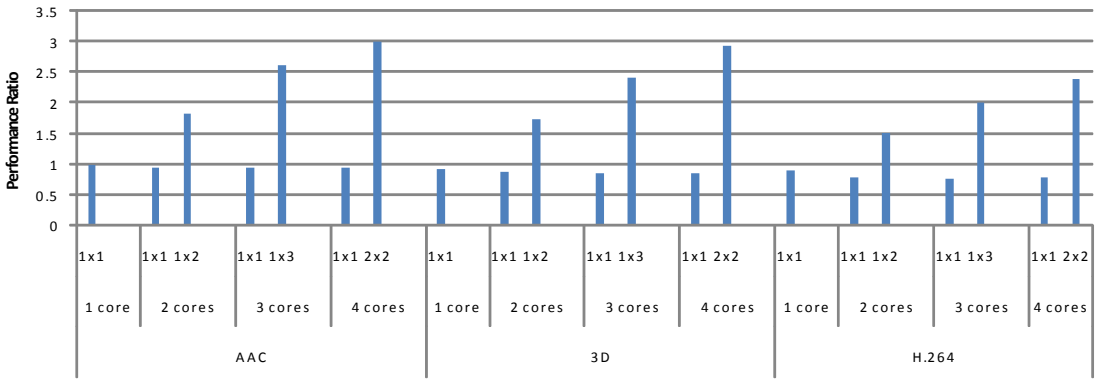


Figure 6.10: Performance Evaluation of VMS

state, the same concept of different scopes in multi-level modulo constraints can be applied to the register allocation. The details are omitted due to the space limitation.

6.5 Experiments

We evaluated the performance and power of a PPA that consists of eight cores as shown in Figure 6.5(a). First, the performance of VMS is presented for kernel loops in three multimedia applications. The performance was measured by the execution time of the three multimedia applications with different configurations of core aggregation and the power was measured only for H.264 application.

6.5.1 Virtualized Modulo Scheduling Evaluation

The performance of VMS was evaluated for 200 kernel loops that can be modulo scheduled in three benchmarks. Figure 6.10 shows the performance ratio of the schedules targeting different set of PPA sub-arrays. The performance ratio of the

schedules was compared to the theoretical upper bound of each loop when mapped onto a single PPA core(MinII). The first column in each benchmark shows the ratio of schedules targeting a single core. For all three benchmarks, the VMS achieves 90% of the maximum throughput for a single core. Considering the distributed hardware in PPA, the Edge-centric approach indeed provide good quality of schedules.

The rest of the columns show how the ratio changes when loops are scheduled targeting multiple sub-arrays for virtualization. The number of target sub-arrays is limited to two since we discovered that targeting more than two sub-arrays does not work well in VMS. The left column in each 'N core' group shows the performance of the schedule when running in a single core, and the right column shows the result for running in multiple cores. As we expected, the performance of the virtualized schedule in a single core decreases by 8% in average. due to the additional modulo constraint. However, mapping these virtualized loops onto multiple cores allows a big performance increase. On average, the speed up of virtualized schedules on 2, 3, and 4 cores are 1.96, 2.76, and 3.23, respectively. Even though there is some performance degradation for a single core, virtualization can accelerate the overall performance of the application in the presence of fluctuating resource availability.

6.5.2 Performance Evaluation of PPA

Figure 6.11 shows the performance from different configurations of the PPA across three applications. The graph shows the execution time for each application in million cycles. The first bar of each application(*acyclic*) represents the entire application

executing on a single PPA core without modulo scheduling. The execution time for AAC and 3D go off the chart and their numbers are shown in the graphs. The second bar(*modulo*) represents the application where the acyclic code runs on a single core, and the inner-most loops are modulo scheduled and execute on 2x2 PPA sub-array. The rest of the graphs shows the performance results when each application is mapped onto different number of PPA cores. Within each '*N* core' group, both static(*st*) and dynamic(*dy*) partitioning were applied.

First, we can notice that exploiting fine-grain pipeline parallelism with modulo scheduling allows 2.53x speed up in average on a 2x2 PPA sub-array. As shown in Section 6.2, increasing the number of cores beyond four does not allow much performance gain due to the limited amount of parallelism. When more resources are available, exploiting coarse-grain pipeline parallelism can further improve the overall performance. In this work, our focus is on the back-end scheduling of streaming applications. Here, we manually extracted the task graphs (Figure 6.3). Other streaming language such as StreamIt [20] can be employed to extract the coarse-grain pipeline parallelism and be fed to our VMS framework as input. We varied the number of PPA cores starting from the number of stages in each application, and increased up to 8 cores available in the PPA. For H.264, we merged the second and the third stages in Figure 6.3(c) since they have relatively small execution time. Since the outer-most loop in H.264 was not pipelined due to the memory dependency, it does not get as much benefit as AAC and H.264.

In general, increasing the number of cores provides the overall performance gain

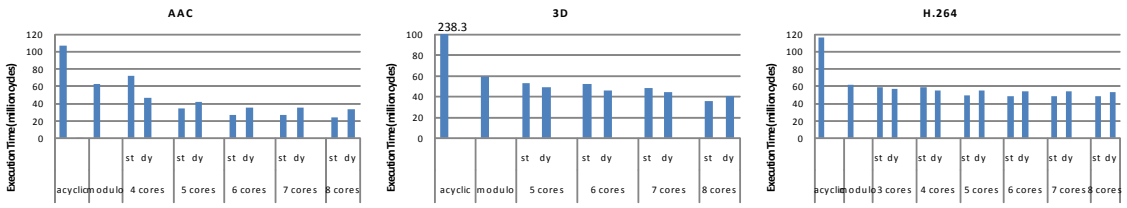


Figure 6.11: Performance Evaluation of PPA

for all three applications and shows reduced execution time over both *acyclic* and *modulo* configurations, with an exception of 4 cores with static partitioning for AAC. This is because AAC spends most of the execution time in the third stage in Figure 6.3(c). Extracting fine-grain pipeline parallelism is mostly important to improve the overall performance. However, static partitioning in 4 cores only allows a single core to be utilized for the third stage. As a result, coarse-grain pipelining parallelism does not gives benefit over *modulo* configuration. Dynamic partitioning with virtualized execution becomes quite useful in this case. With virtualization, the compute intensive kernels in the third stages can utilize additional resources from other stages when they are sitting idle, allowing 1.55 speedup over *static* configuration.

The same trend appears on all three applications; the dynamic partitioning out-performs the static partitioning when there is not enough resources to fully exploit the available fine-grain pipeline parallelism. However, the benefit of virtualization diminishes as more resources become available in the target PPA configurations. For AAC, the static partitioning out-performs the dynamic partitioning when 5 cores are utilized. For 3D and H.264, the reversion of performance appears later at 8 cores and 5 cores, respectively. This is because the amount of fine-grain pipeline

parallelism is richer in two applications than in AAC. To summarize, exploiting the coarse-grain pipelining parallelism does provide the overall performance improvement over *modulo* configuration with both static and dynamic partitioning. When there is a large number of resources available, static partitioning in 8 cores can achieve the speedup of 4.5 and 1.84 over *acyclic* and *modulo* respectively in average for three applications. The dynamic partitioning can maximally utilize the available resource in smaller arrays, out-performing the static partitioning.

Limitations on Virtualization

Virtualized Modulo Scheduling should be carefully applied to the kernel loops since it incurs run-time overhead and performance degradation. The run-time overhead includes the latency of hand-shaking between neighboring cores to check the resource availability and the code migration latency. Since this overhead is incurred in every invocation of a virtualized loop, virtualization can increase the overall execution time when the trip count is small. In this work, we virtualized all the modulo scheduled loops without considering the benefit. We believe selectively virtualization with profiling information can definitely improve the performance of dynamic partitioning results in Figure 6.11. Also, resource allocation at run-time is performed in a greedy way in our evaluation. More sophisticated allocation that considers the benefit of the additional resources can also improve the results. Attacking the current limitations of virtualization is on the list of our future work.

6.5.3 Power and Area Measurement

Area and power consumption was measured using the RTL Verilog model of the Polymorphic Pipeline Array (PPA) and synthesized using typical operation conditions in TSMC 90nm technology. The model contained both the datapath and control path and was targeted at 200MHz. Synthesizing higher frequencies was possible, but at 200MHz the target applications could be completed and more aggressive frequencies would generate a less energy efficiency design. The memories were generated using standard library models found in the Artisan SRAM memory compiler. Power consumption was calculated using Synopsys PrimTime PX. PrimTime calculates the total power consumption of the PPA using the synthesized netlist and parasitic data generated from Physical Compiler and activity files generated from behavioral simulations. The SRAM memory power was extracted from data generated by the Artisan Memory Compiler. The breakdown of average power when 8 PPA cores are executing the whole code region of H.264 is shown Figure 6.12(a). The major portion is in computation units like PEs and LRFs. The data memory power consumption is relatively low because H.264 has a high ratio of computation over memory operations. The average power for 8 cores running in pipelining mode is 255.06mW at 200Hz. The toal area of 8 core PPA is 3.37 mm^2.

Figure 6.12(b) plots the performance vs. power consumption of the PPA and other existing architectures. The numbers were obtained from a graph in [16]. On this plot, points on the same slope have roughly equivalent power efficiency in terms of MIPS/mW, with points towards the upper left having greater power efficiency. As

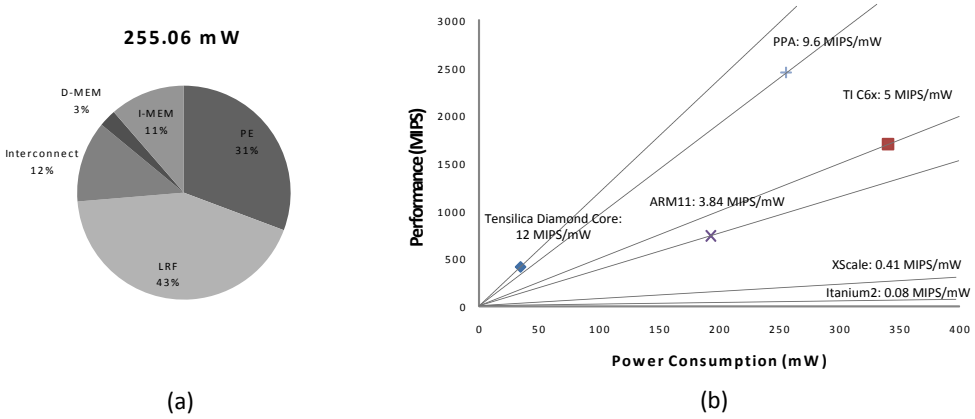


Figure 6.12: (a) Power breakdown of PPA: running H.264, (b) Power/performance comparison

can be seen from the plot, the PPA is able to achieve good power efficiency, only beaten by The Tensilica Diamond Core [53]. Embedded procoeessors like ARM11 and TI C6x show reasonable power efficiency, but their performance is significantly lower than PPA. Thus, they cannot meet the performance requirement of today's compute-intensive multimedia applications. The actual data points for XScale and Itanium2 are outside the range of the plot, but their efficiency lines are shown. As can be observed, the efficiency decreases significantly as the hardware becomes more general and less tailored for embedded applications.

6.6 Related Works

Architectures: Combining cores to create a bigger logical core is quite a new technique, recently proposed by Core fusion [22] and Composable Lightweight Processors [24]. Core Fusion is a CMP architecture that can dynamically allocate independent cores together for a single thread execution maintaining ISA compatibility. CLPs

also allows dynamic allocation of cores to form a larger and powerful single-threaded processors. It also keeps the binary compatibility for the special EDGE ISA. The major difference between [22] and [24] is the target environment. PPA is designed to exploit single thread performance in mobile environments where power consumption and hardware cost is a first-class constraint. The building blocks of PPA is simple in-order cores similar to clustered VLIW processors [59]. Also, the statically controlled point-to-point interconnect provides a fast inter-core communication, allowing PPA to exploit fine grain pipeline parallelism efficiently for multimedia applications.

The PE level view of PPA is similar to Coarse-Grained Reconfigurable Architectures. ADRES [38] is a reconfigurable architecture where PEs are connected to a mesh-style interconnect. Modulo scheduling using a simulated annealing is employed to exploit fine grain pipeline parallelism of nested loops. The top row in the array behaves as a VLIW processor with a multi-ported central register file. However, the non software pipelineable region of the application can only utilize the VLIW part of the array. So, it cannot pipeline the application in a coarser granularity as PPA. PipeRench [18] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining, but it has a fixed configuration of resource partitioning for pipelining while PPA can partition the array differently as to the characteristics of the target application. RaPiD [13] is another CGRA that consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

Exploiting Parallelism: Coarse-grained pipeline parallelism is becoming an

attractive approach to accelerate single thread performance as multicore architectures enter the mainstream. [20] and [27] proposed to exploit coarse-grained pipeline parallelism for StreamIt language. Even their target architectures(RAW architectures [30] and Cell processors [21]) are not an embedded system, their technique of task level software pipelining can be applied to our execution model in PPA. [55] has proposed a practical approach to extract a pipeline parallelism from legacy C code. With a help of the programmer, their static analysis tool can extract the potential for streaming execution.

Virtualization: There is much related work on virtualization for binary compatibility for different architectures in literature. Transmeta Code Morphing Software [9] dynamically converts x86 applications into VLIW programs. DynamoRIO [2], Daisy [11, 12], and DIF [41] are all examples that dynamically translate applications to target entirely different microarchitectures. Recent work [6] proposed dynamically binding to cyclic accelerators with modulo scheduling at run-time. The trace in a host processor is examined and run-time modulo scheduling is performed to map the kernel loops onto the cyclic accelerator. While this work focuses on acyclic-to-cyclic conversion, we propose dynamic conversion of modulo scheduled loops in homogeneous multi-core architectures.

6.7 Summary

In this chapter, we proposes a flexible multicore accelerator for mobile multimedia applications. Fine-grain and coarse-grain pipeline paralellism cooperatively improve

a single thread performance of computation-rich multimedia applications. To efficiently extract both forms of parallelism on the same piece of hardware, the PPA supports a flexible execution model where cores can be combined to create a powerful core that can effectively exploit fine-grain and and pipeline parallelism to achieve up to 8x speed up over a 4-wide VLIW processor. The array can be either statically or dynamically partitioned depending on the computation requirement of the application. For dynamic partitioning, we propose Virtualized Modulo Scheduling that can generate a single schedule of a target loop that can be easily converted to target different sub-arrays at run-time. With both static and dynamic partitioning, the 8-core PPA can achieve up to 4.5 speedup over a single core execution.

CHAPTER 7

Conclusion

7.1 Summary

Polymorphic Pipeline Array is a flexible multicore accelerator that improves upon Coarse-Grained Reconfigurable Architectures for future mobile multimedia applications. The acceleration of applications is extended beyond the innermost loops to the whole region of the applications. Both fine-grain and coarse-grain pipeline parallelism rich in today's multimedia applications can be effectively exploited by modulo scheduling and streaming execution with tightly coupled cores. Also, PPA shares the inherent power efficiency of CGRAs with fully distributed register files, nearest neighbor interconnect, and simple control. Thus, PPA provides an attractive solution that meets both high computing performance and tight power consumption budget requirements in future embedded systems.

In this dissertation, various compiler and hardware optimizations are presented for CGRAs that form the basic building block of PPA. Scheduling problem in highly

distributed architectures like CGRAs is quite different from conventional scheduling in that the routing of operands should be explicitly orchestrated by the compiler. Two modulo scheduling techniques are proposed to enhance the quality of schedules in a fraction of compilation time over the traditional approach.

Modulo graph embedding leverages classic graph embedding to draw loop bodies onto a three dimensional scheduling space. An affinity graph heuristic analyzes producer/consumer relations to place operations with common consumers closely. A skewed scheduling space allows dense packing of operations while ensuring operand routing paths are available. Edge-centric modulo scheduling focuses primarily on the routing problem, with placement being a by-product of the routing process. EMS categorizes the edges in dataflow graphs based on their characteristics and apply different strategy to routing them. EMS achieves 98% of a state-of-the-art simulated annealing approach, while reducing compilation time by 18x.

A novel control path design is proposed that adopts the concept of a token network in dataflow machines for fine-grain code compressions in CGRAs. The datapath is cloned to create a token network where resources observe the incoming tokens to determine the instruction format in fine-grain code compression. The resulting control path reduces the control power consumption by 56% while enabling a performance gain of 4%.

While these optimizations attack the bottlenecks for deploying CGRAs for mobile environments, CGRAs have two inherent weaknesses: scalability and accelerating only innermost loops. With increasing technology, more resources are likely to be

available in future embedded systems. PPA can utilize large number of resources by accelerating the whole region of applications with flexible execution model. By adopting streaming execution model, target applications can be further accelerated by running multiple tasks concurrently on partitioned sub-arrays of PPA. The most distinguishing feature of PPA is its ability to dynamically partition the resources as to the availability. Virtualized modulo scheduling generates a unified schedule that can be dynamically converted to target different sub-arrays. The 8-core PPA can achieve up to 4.5x speed up over a single core execution.

7.2 Future Directions

The research presented here can be extended in several directions. First, the performance of dynamic partitioning can be improved in various ways. The current VMS is in a initial development stage and can be improved with appropriate heuristics that reduce the sub-optimality of the unified schedule. Also, virtualization of loops should be carefully applied considering the run-time overhead and the scheduling quality. We noticed that the VMS scheduling results vary across different loops (ie some lose, some win). Selective application of virtualization combined with selective generation of multiple independent schedules will improve the results dramatically. More intelligent resource allocation at run-time is another feature that requires in-depth investigation to improve the performance.

Another issue that requires more attention is how to extract the task graphs of target applications for streaming execution. In this dissertation, all the applications

were manually analyzed and converted to streaming style applications. This definitely limits the number of applications that can be evaluated. It is well-known that legacy C codes are hard to parallelize due to their complicated memory dependencies. StreamIt [56] can be a good solution to obtain a wide range of applications. There are already a large number of multimedia applications written in StreamIt language. Replacing the front-end of the current framework with StreamIt compiler system is a feasible and attractive solution.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proc. of the 2006 Design, Automation and Test in Europe*, pages 363–368, Mar. 2006.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[3] K. Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal Applied Signal Processing*, 2005(1):2613–2625, 2005.

[4] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003.

[5] F. Bouwens, M. Berekovic, B. D. Sutter, and G. Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *Proc. of the 2008 International Conference on High Performance Embedded Architectures and Compilers*, pages 66–81, Jan. 2008.

[6] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008.

[7] K. Coons, X. Chen, S. Kushwaha, K. McKinley, and D. Burger. A spatial path scheduling algorithm for edge architectures. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, Oct. 2006.

[8] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[9] J. Dehnert et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of*

the 2003 International Symposium on Code Generation and Optimization, pages 15–24, Mar. 2003.

[10] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[11] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–38, June 1997.

[12] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, June 2001.

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

[16] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009.

[17] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *Proc. of the 2004 Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[18] 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.

[19] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[20] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[21] IBM. *Cell Broadband Engine Architecture*, Mar. 2006.

[22] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.

[23] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[24] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 381–393, Dec. 2007.

[25] Y. Kim, I. Park, K. Choi, and Y. Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proc. of the 2006 International Symposium on Low Power Electronics and Design*, Oct. 2006.

[26] G. Krishnamurthy, E. Granston, and E. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proc. of the 2002 International Conference on Supercomputing*, pages 107–116, June 2002.

[27] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008.

[28] A. Lambrechts, P. Raghavan, M. Jayapala, F. Catthoor, and D. Verkest. Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In *Proc. of the 2008 International Conference on VLSI Design*, pages 201–207, Jan. 2008.

[29] 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.

[30] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

[31] 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.

[32] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 194–203, Dec. 1997.

[33] W. Li and H. Kurata. A grid layout algorithm for automatic drawing of biochemical networks. *Bioinformatics*, 21(9):2036–2042, 2005.

[34] S. Liao et al. Code optimization techniques for embedded DSP microprocessors. In *Proc. of the 32nd Design Automation Conference*, pages 599–604, 1995.

[35] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.

[36] G. Lu et al. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[37] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 164–175, Nov. 2008.

[38] 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.

[39] B. Mei, A. Lambrechts, J. Y. Mignolet, D. Verkest, and R. Lauwereins. Architecure exploration for a reconfigurable architecture template. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 90–101, Mar. 2005.

[40] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 141–150, Oct. 2006.

[41] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[42] 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.

[43] H. Pan and K. Asanovic. Heads and tails: a variable-length instruction format supporting parallel fetch and decode. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 168–175, Nov. 2001.

[44] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[45] 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.

[46] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008.

[47] M. Quax, J. Huisken, and J. Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 230–235, Mar. 2004.

[48] 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.

[49] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Transactions on Architecture and Code Optimization*, 4(1):7, 2007.

[50] 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.

[51] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb and the armt7tdmi. *IEEE Micro*, 15(2):22–30, 1995.

[52] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

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

[54] Texas Instruments. *TMS320C55x DSP CPU Programmer's Guide*, Aug. 2001. http://focus.ti.com/lit/ug/spru376a/spru376a.pdf.

[55] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, Dec. 2007.

[56] 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.

[57] 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.

[58] M. Woh et al. From soda to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[59] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for VLIW processors. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Sept. 2005.