Offset Pipelining for Coarse Grain Reconfigurable Arrays

Aaron Wood

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Scott Hauck, Chair

Andrew Putnam

Visvesh Sathe

Program Authorized to Offer Degree:

Electrical Engineering

University of Washington

Abstract

Offset Pipelining for Coarse Grain Reconfigurable Arrays

Aaron Wood

Chair of the Supervisory Committee:
Professor Scott Hauck
Electrical Engineering

This dissertation presents an execution model and compilation algorithms to advance the utility of coarse-grained reconfigurable arrays (CGRAs). These time multiplexed, spatial architectures can provide improved energy efficiency and performance compared to FPGAs or commodity processor systems. Conventional CGRAs are generally modulo scheduled for efficiently pipelining computationally intensive code. However, as the control complexity of an application increases, the performance of modulo scheduled CGRAs is diminished. An application composed of a series of phases and complex control flow may yield poor device utilization and limited performance on conventional CGRAs. In this work, I present Offset Pipelining, a new execution model, along with supporting Offset Pipelined Scheduling and EveryTime routing algorithms for improved application mapping to CGRAs.

Offset Pipelining provides a mechanism to support different phases of application execution. The approach increases the flexibility of CGRAs by supporting independent initiation intervals for the different phases and interleaving loop iterations across the device, similar to modulo scheduling. This allows portions of an application to be optimized in isolation and also maximize resource sharing on the device. It is particularly effective in cases where an application requires infrequent setup or teardown steps around a shorter inner loop that executes for many iterations. I also introduce pipelined program counter CGRAs in order to support the proposed execution style.

The new algorithms presented perform the scheduling and routing for an Offset Pipelined CGRA. The Offset Pipelining Scheduling algorithm creates a schedule in the Offset Pipelining style. It takes an iterative approach to adjusting the schedule, balancing the needs of each phase by positioning operations and issue slots to maximize performance. The proposed EveryTime router manages new complexity resulting from the Offset Pipelining execution model. This includes signals with run time dependent paths and variable flight time. The Offset Pipelined Scheduling and EveryTime routing algorithms are joined by a more conventional simulated annealing placement phase to form a prototype tool chain to demonstrate the feasibility of Offset Pipelining for CGRAs.

These components provide improvements to the utility of CGRAs compared to existing techniques. The Offset Pipelining execution model increases the flexibility of CGRAs on a practical hardware architecture. It offers better performance by allowing each phase to execute independently rather than fused into a single monolithic schedule for the entire application. This work hopefully advances the development of CGRA architectures and tools.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The support I have received throughout my time as a graduate student has been tremendous. Most importantly, I want to thank my wife Kathy. Despite the seemingly perpetual timeline, her drive and love have sustained me through the years. Her encouragement was critical to my success.

It is difficult to express the scope of my appreciation for my advisor Professor Scott Hauck. My journey started when I took the introductory digital logic course in the Electrical Engineering department from him in 2003. I subsequently had the opportunity to work with him as an undergraduate researcher which sparked my interest in reconfigurable computing and design automation. I was incredibly fortunate to have the opportunity to work with Scott once again in graduate school in my area of interest. He has offered his guidance and mentorship through some of the most important events in my life, including getting married, moving across the country twice, and the birth of two daughters. I would not have completed this work without his steadfast support and patience.

My mother in particular deserves special thanks for her technical assistance, including discussions, paper and dissertation revisions, and benchmark development. In the latter stages of my time in graduate school, my parents and in-laws were instrumental in helping me find the time to complete my dissertation. Their help in caring for our older daughter gave me the time I needed to finish my work. My parents have been an inspiration, always interested and believing in my work.

I have been supported by the Electrical Engineering department, the National Science Foundation, and the Department of Energy at various points throughout graduate school. This financial support allowed me to remain in graduate school to pursue my research.

Lastly, I would like to thank everyone I had the pleasure of working with or alongside during my time in the program. My experience in graduate school, while long and challenging, has ultimately been a great experience within a fantastic community. Thank you!

DEDICATION

For my wife Kathy who supported me through this long endeavor from the beginning.

Chapter 1. INTRODUCTION

The potential of spatial parallel processing architectures is constrained by the lack of suitably mature programming models and compilation tools. This work introduces Offset Pipelining and an accompanying tool chain for multi-mode computations on coarse grain reconfigurable array (CGRA) architectures with pipelined program counters. Before getting to the details of the approach and device features, a brief overview of computing architectures provides context for this work. A more specific discussion found in Chapter 2 covers the benefits and limitations of modulo scheduling on CGRAs to provide a framework to discuss the contributions of the proposed toolchain. Understanding the broader computing landscape aids the reader in seeing the potential of this work to inform future architecture development and the necessary tool support for rapid, high performance application development.

1.1 Accelerating Signal Processing Applications

The scope of computing is wide and pervasive. It ranges from scientific computing to the smartphone applications. Such diversity is addressed by a correspondingly rich selection of devices and systems engineered to meet different performance requirements. This work focuses on mapping signal processing applications to CGRAs to evaluate a new CGRA execution model. Applications in this area are computationally demanding with performance typically resource limited, making them good candidates for acceleration with spatial architectures such as CGRAs.

Applications with strong modal behavior, exhibited as discrete phases of computation, are particularly interesting. These programs are generally more challenging to map to existing devices such as FPGAs due to difficulty in handling branching execution. CGRAs and tools in this work are designed to support modal behavior and provide new opportunities for developers to produce high performance solutions over a wider range of applications.

1.2 Options for Computation

The semiconductor industry fields a wide variety of commodity computing devices. Each is tailored to fill a specific role in the marketplace. They may be designed to excel at a particular type of computation, fit into a given power envelope, or meet some other combination of

performance metrics. There are many costs associated with designing, building, and deploying a device. While a custom application specific integrated circuit (ASIC) is an obvious choice without budget or time constraints, an ASIC is not practical for the majority of applications. The development time and cost of ASIC design makes it impractical for all but the highest volume and performance sensitive applications. The majority of devices avoid significant non-recurring overhead being prefabricated "off the shelf" components. These range from general purpose processors to Field Programmable Gate Arrays (FPGAs). All options gain flexibility at the expense of efficiency for a specific application compared to custom ASICs. While I focus on CGRAs as the target for the proposed Offset Pipelining execution model and tool chain, this is not to say that this work is limited to these devices. The overview here lays out the landscape of computing before focusing on CGRAs.

1.2.1 *Processors*

The most flexible computing platform is a general purpose processor. These devices execute a program by processing a sequence of instructions to perform a desired computation. A set of instructions configures the computation units to perform a function on a set of inputs. Modern systems with substantial memory resources at their disposal are not bounded by the number of instructions or even the amount of data that can be processed. However, the execution model of a processor sets limits on the performance. The serial nature of instruction execution remains a fundamental constraint limiting the scalability of a general purpose processor. Adding more processors to a system cannot alleviate this problem without significantly modifying a target application due to the serial nature of the execution model.

More closely related to this work, very long instruction word (VLIW) processor architectures inspire aspects of CGRAs and associated tool flows. VLIW machines eschew complex scheduling logic in favor of simpler hardware that relies on the compiler to schedule operations onto a set of compute resources at compile time. Chapter 2 introduces this scheduling problem and solutions through modulo scheduling as it applies to VLIW as well as CGRA architectures.

1.2.2 *Graphics Processing Units*

Graphics processing units (GPUs) compose a class of domain specific hardware that has been increasingly leveraged for accelerating applications beyond graphics. These devices excel where the same computation is replicated across large, independent data sets. This is single instruction, multiple data (SIMD) style processing. GPUs are particularly suited to the extensive use of floating point computations found in many scientific computing applications and also offer high memory bandwidth. If a single instruction stream describes the actions to be taken over a large collection of data with relatively few data dependencies, GPUs offer large performance improvements compared to conventional processors. However, efficient utilization requires keeping the compute units busy doing useful processing.

1.2.3 *Spatial Computing Architectures*

Another device category is the massively parallel processer array (MPPA). These devices integrate dozens to hundreds of relatively small independent processors which is an order of magnitude more than the number offered by commodity multicore processors. While a hierarchical organization can provide uniformity among resources, MPPAs are usually characterized by a grid oriented communication network to take advantage of the proximity of adjacent resources but still allowing communication across the device. The challenge in developing applications for these devices is to harness the individual processors in coordination around a single task or otherwise keep a large portion of the device performing useful computation for a given application. The much larger number of processors coupled with typically increased non-uniformity of memory access makes application development more challenging than for commodity multicore systems. While MPPAs offer a unique hybrid of computation density and flexibility, tools for development on these platforms remain largely limited to traditional software development flows.

FPGAs constitute a special class of ASICs. Manufactured in a generic fashion and configured using firmware, they bypass the expensive fabrication of an ASIC. An FPGA can implement arbitrary digital logic structures, but is limited by the capacity of the device and application performance requirements. FPGAs have significant area and power overhead compared to ASICs, but allow designs to be modified without the massive fabrication effort

required to produce an ASIC. The flexibility to tailor the hardware to specific applications is compelling with FPGAs offering significant parallelism despite the overhead. However, FPGA designs also face the additional challenges of physical design and explicitly managing memory resources.

1.2.4 *Coarse Grain Reconfigurable Arrays*

A coarse grained reconfigurable array (CGRA) contains a large number of word-wide functional units and small distributed memories connected in a style reminiscent of FPGA interconnect. This coarsened resource granularity mitigates some of the overheads seen in FPGA devices. Most CGRAs support time multiplexed execution to maximize computation density. While coarse grained resources are a boon when the granularity matches the application requirements, some applications may use these resources inefficiently. They generally lack expressive program counters, relying instead on static scheduling to map an application to the available resources. CGRA tool flows adopt ideas from traditional software compilation as well as CAD concepts found in FPGA or ASIC tools. Bridging the divide between a traditional processor execution model and a parallel computing substrate is an exciting and promising area of research across spatial computing. CGRA tool support compatible with familiar high level language development is a powerful prospect. The ability to leverage highly parallel hardware in this way offers a potential avenue for continued silicon performance improvements to supplement increasingly limited single thread performance gains.

1.3 RUNTIME RECONFIGURATION ON RECONFIGURABLE HARDWARE

Some FPGAs support changing their configurations at run time. A portion of the device may receive a new configuration to execute a different computation. This type of run time reconfiguration is limited in practice to sections that have been prepared to be swapped at compile time. Since even a partial bitstream is many times larger than a single processor instruction, reconfiguration is a costly operation and should happen infrequently. This cost may be worth paying for applications that have phases whose execution time dwarfs the reconfiguration time.

Most CGRA work focuses on statically mapped applications, comparable to a single program executing on a processor or a single configuration of an FPGA. For long running signal

processing applications, this is often preferred since greater effort at compile time to build the best application implementation benefits the application that runs for a very long time. There is an important distinction to make between a time multiplexed CGRA and a statically configured FPGA in terms of reconfiguration. An FPGA bitstream holds a single configuration of the available resources while a time multiplexed CGRA bitstream will include multiple contexts designed to be switched on a per cycle basis at run time.

In a time multiplexed CGRA, reconfiguration refers to the fine grain time multiplexing of resources as opposed to conventional FPGA reconfiguration. While execution is time multiplexed, applications on a CGRA are mapped in a static fashion; there is no dynamic instruction selection performed on the device itself.

1.4 Broadening the scope of CGRAs with Offset Pipelining

The limited control of conventional CGRAs constrains the utility of these devices to the inner loops of computationally intensive applications. This work introduces Offset Pipelining, an execution style that alleviates this limitation, broadening the scope of applications that can be efficiently mapped to CGRAs.

1.4.1 *Modal Computation*

The key constraint of modulo control is that there is only a single sequence of instructions available to perform the target application. If the application structure deviates from a single loop body, even with only minor conditional execution, overhead in the mapping will emerge. Operation predication is an effective tool to deal with operations whose results are not used on a given pass of the instruction sequence, but as the application control flow becomes more varied, the overhead may eliminate the potential gains of executing on parallel hardware. Offset Pipelining avoids much of this overhead by supporting instruction sequences that more closely match the application control flow.

1.4.2 *High Density Mapping*

K-means clustering provides an example application that exhibits modal behavior. The process logically breaks down into two phases. A data set is evaluated against a set of means and then

the means are updated. Such an application can benefit from an execution model and mapping tools that can effectively partition the phases of the computation. Thus, the different modes share the same resources in a multiplexed fashion with the understanding that, at run time, the modes are mutually exclusive. Enabling improved sharing over modulo scheduling makes a higher density mapping of the application possible. A multi-mode mapping may reduce the effective initiation interval compared to a modulo scheduled version. Higher density potentially benefits the back end placement and routing in the tool flow.

1.4.3 *Tradeoffs*

While Offset Pipelining is a promising approach for the modal signal processing applications evaluated in this work, it is certainly not without limitations. Applications without modal behavior might be mapped poorly compared to existing modulo scheduling techniques. The execution model also adds significant complexity to the routing problem which requires additional tool algorithm innovation discussed in Chapter 8.

1.5 THE BIG PICTURE

The Offset Pipelining prototype tool chain in this work explores an alternative approach to mapping multi-mode applications to CGRAs. The proposed execution model not only enables a broader range of applications for CGRAs, it does so by retaining an interface to conventional software front end compilation to fit into established application development flows. The proposed approach leverages an array of compute units and automatically tailors the application mapping to the available resources providing better performance and utilization than existing approaches for signal processing applications with significant conditional execution.

Chapter 2. SCHEDULING FOR MODULO COUNTER CGRAS

This chapter provides background on modulo scheduling as applied to modulo counter controlled CGRA architectures. Introducing these concepts sets the stage to present Offset Pipelining in the next chapter. CGRA architectures are discussed first with an emphasis on modulo counter based control for these systems. With an understanding of the hardware organization, the next subsection addresses mapping applications to the hardware using modulo scheduling for loop pipelining. Applying predication to manage control flow is discussed next to demonstrate how more complex practical applications are handled with modulo scheduling. The chapter provides a basis for comparison to the proposed Offset Pipelining execution model introduced in the subsequent chapter. Understanding modulo scheduling additionally provides corresponding background to discuss the Offset Pipelining Scheduling algorithm in a later chapter.

2.1 CONVENTIONAL CGRA ARCHITECTURES

CGRAs combine features of FPGA architectures and very long instruction word (VLIW) processors. There are many variations, but they are generally characterized by time multiplexed, word oriented functional units. Limited interconnectivity between the functional units is another important feature that simplifies scaling up device sizes. Further comparisons can be drawn with other architectures such as GPUs and MPPAs, but focusing on FPGAs and VLIW processors provides a construction that aligns with the mapping techniques for CGRAs covered later in this chapter.

Having adopted features of both, CGRAs represent a middle ground between FPGAs and VLIW machines in a variety of ways. A CGRA is composed of an array of functional units with interconnect structures similar to those found in FPGAs. A tile oriented organization coupled with configurable interconnect affords a scalable architecture for CGRAs and FPGAs. The functional units in VLIW machines and CGRAs are word oriented and time multiplexed leading to considerable savings in configuration overhead compared to the per bit configuration for FPGA resources.

From a configuration standpoint, a CGRA requires considerably less memory to configure its resources than an FPGA. A 6 input LUT requires a 64-bit RAM to produce a single bit result while an ALU might use the same memory to support 16 contexts using a 4 bit opcode and

producing a word-wide result. While time multiplexing resources multiplies the memory required by the number of contexts stored, this is offset by savings in logic and routing memory requirements. Even using the same amount of memory, a CGRA with time multiplexed resources would benefit from high density memory that does not need to expose every bit simultaneously for configurations as in an FPGA.

For interconnect, configuration memory reduction is due to the bundled organization and is on the order of the datapath width, since each bit of the word shares the same configuration. An equivalent path on an FPGA must configure each bit independently. Word wide functional units likewise reduce the amount of configuration memory needed for operations supported by these units, mainly arithmetic and logical functions on a data word. By contrast, while FPGAs include blocks for these operations, the dominant resources by far are look-up tables (LUTs) that use tens of bits to configure a single bitwise operation. Though ultimately flexible, this is a significant overhead for primarily word oriented applications. While there are advantages in favor of CGRAs from an overhead standpoint, coarsening resource granularity is only beneficial if the application is well matched. Applications that do not need or cannot use the full width of the available resources will have poor utilization.

As much as a CGRA reduces the configuration memory requirements compared to an FPGA, it is still massive compared to instructions for a VLIW machine. A CGRA with hundreds of execution units and supporting interconnect clearly requires more configuration than a VLIW machine with perhaps tens of execution units and a large multi-port register file acting as a highly flexible interconnect resource. While it is feasible for a VLIW machine to read instructions from memory and execute a very long program, this is not the case for a CGRA. Similar to an FPGA, a CGRA achieves its best efficiency when configured only once to execute an application. Unlike FPGAs, the relatively compact configurations for CGRAs make it feasible to store a number of resource configurations that may be selected at run time. While not the virtually unbounded program size of a VLIW machine, this set of instructions is suitable for the computationally intensive and performance limiting portions of signal processing applications that are the focus of this work. These instructions are generally executed in a repeating sequence at run time to time multiplex the resources.

The concept of time multiplexing is illustrated in Figure 2.1. In a device with a single configuration, such as an FPGA, each operation must be assigned to its own resource. A time

multiplexed system allows multiple operations to share the same physical resource by executing them at different times. The example shows resources with two time slots allowing the addition and shift operations to coexist on ALU0. Time multiplexing helps to increase the density of the computation, providing better device utilization and locality. The following section provides a brief overview of the control necessary to manage the instruction memory for a time multiplexed CGRA.

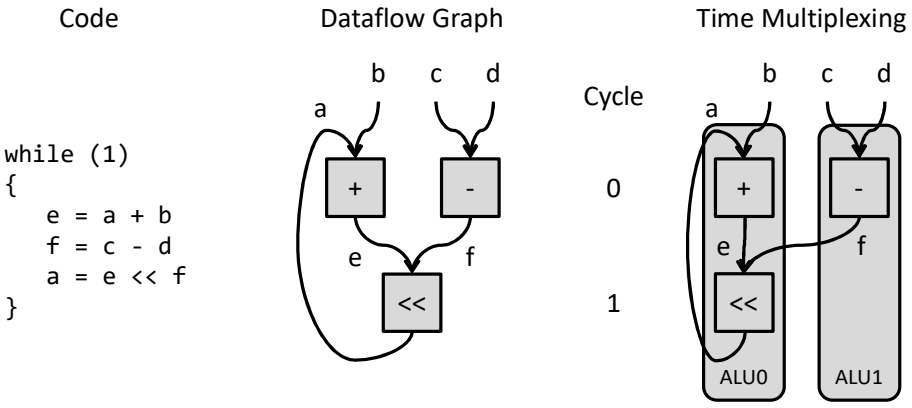

Figure 2.1. Source code (left) is converted to a dataflow graph (center) and mapped to a time multiplexed architecture (right).

2.2 Modulo Counter Control

For an FPGA, the configuration memory is usually unchanged after configuration for the lifetime of the application or for a long running phase of computation. In order to take advantage of multiple configurations on a CGRA, they must be accessed to continuously update the active instruction. While the next section discusses the execution of modulo scheduled applications, this section introduces the hardware features that facilitate it. The desired behavior of a modulo scheduled system is to repeatedly execute a loop of a sequence of instructions. This can be managed using a modulo counter which performs exactly this function. It may be configured to adjust the length of the schedule depending on the application mapping. The actual implementation in hardware may take a number of forms, but would most likely be distributed throughout the device to avoid distributing an address over a wide area unnecessarily when the execution sequence has been predefined during the application mapping process. Instruction memory is addressed using the modulo counter to look up the configurations for the available

computation units and interconnect resources on a cycle by cycle basis at run time. The repeated execution of a sequence of instructions is the desired behavior for a modulo scheduling introduced in the next section.

2.3 MODULO SCHEDULING

The introduction to modulo scheduling presented here focuses on scheduling for VLIW machines. Though the absence of large multi-port register files in CGRAs complicates the mapping process, a VLIW machine is a good analogue for CGRA scheduling. Software pipelining by modulo scheduling for VLIW compilation is a widely studied problem [RG81, Lam88, WBH+92, Huff93, Rau94, LGA+96]. As mentioned previously, a VLIW machine has a sizable advantage in terms of silicon area over a traditional out of order processor design. For VLIW machines, the compilation process defines precisely when an operation will be executed on each of the available functional units. Out of order machines devote considerable silicon real estate to perform dependency analysis and make scheduling determinations in hardware at run time.

Practical VLIW machines, along with most computing architectures, contain a variety of functional units to perform different operations including a mixture of integer units, floating point units and memory units. To simplify the following discussion, functional units will be generically called ALUs and are assumed to execute one operation per cycle in the following abstract examples. These simplifications are strictly for illustrative purposes.

```
while (1)
{
   a = readStream(stream1);
   b = readStream(stream2);
   c = (a + b) >> 1;
   writeStream(c, stream3);
}
```

Figure 2.2. Example of easily pipelined code.

The prototypical VLIW machine operates most efficiently when there is a large volume of processing that can be scheduled without branch instructions or dependencies that limit the amount of instruction level parallelism that can be exposed by the compiler. A simple example would be code such as that in Figure 2.2. This code is easily pipelined onto three ALUs as shown in Table 2.1. This scheduling represents a possible output of a modulo scheduler for this

code, the unrolled execution of which is shown in Table 2.2 with operations belonging to a single iteration outlined. In this example, operations from two different iterations are executing across the device simultaneously.

Table 2.1. Scheduled Operations

Cycle	ALU0	ALU1	ALU2
0	Read	Read	>>
1	+		write

In this example, each iteration of the loop contains a sequence of instructions that does not depend on operations from other iterations, making it easy to schedule the operations on the available ALUs over a period of two cycles (Table 2.1). This pair of cycles can then be executed repeatedly such that a single iteration of the loop executes over two repetitions of the schedule, with operations from two different iterations being executed simultaneously.

Table 2.2. Unrolled Schedule

Cycle	ALU0	ALU1	ALU2
0	Read	Read	>>
1	+		write
2	**Read**	**Read**	>>
3	**+**		write
4	Read	Read	**>>**
5	+		**write**
6	Read	Read	>>
7	+		write

Table 2.1 shows the issue slots of each ALU for each cycle in the schedule. The length of the repeating sequence of cycles is called the initiation interval (II), which indicates the time interval between the beginnings of successive iterations of the loop body. With no dependencies between iterations, this example is resource limited and could be executed faster, i.e., in only one cycle, if additional ALUs are available. With five ALUs, there are sufficient issue slots for an II of one cycle. More resources provide no further benefit for this example as it stands. Transformations such as unrolling the loop may provide additional opportunities for parallelism

by merging successive iterations, making it possible to leverage additional resources. For more complex code structures, loop fission or fusion, inversion or unswitching may help improve scheduling.

If the example is modified to add a dependency between operations in two different iterations, there is now a recurrence loop in the sequence of instructions. The length of the recurrence loop defines the minimum II into which a given set of instructions may be legally scheduled, regardless of the number of resources available. For example, if the writeStream operation were to change what would be read by one of the readStream operations of a subsequent iteration, the smallest II possible for that code is 4, in order to account for the dependency. While in the previous case additional resources allowed for more parallelism, the recurrence loop places a fundamental limit on the performance of the application.

Going to the other extreme of limiting the design to a single ALU, each operation is scheduled into its own cycle and the II is equal to the number of instructions in the target loop. A single ALU controlled by a program counter is a very simple processor that serves as a baseline for worst case performance. Such a processor exploits no parallelism with the single cycle operation assumption. Independent of modulo or other scheduling technique, application properties can be used to determine best or worst case performance based on resource constraints. This is independent of how the application is mapped to the target device.

2.3.1 *Iterative Modulo Scheduling Algorithm*

The basis of many modulo schedulers is the Iterative Modulo Scheduling (IMS) algorithm [Rau94]. The goal is to extract the most parallelism from the loop to keep the available ALU resources busy. To prepare for the main scheduling loop, the target code is analyzed to calculate the recurrence constrained minimum II (recII), which is the longest recurrence loop in the code. The resource constrained minimum II (resII) is calculated as the quotient of the total number of operations in the loop and the number of available ALUs for scheduling (Equation 2.1).

$$resII = \left\lceil \frac{TotalOps}{ALUs} \right\rceil \tag{2.1}$$

The maximum of these two minima is the minimum II that could possibly accommodate the target code and is the starting point for scheduling (Equation 2.2).

$$II = \max(resII, recII) \tag{2.2}$$

The primary data structure for scheduling is the modulo reservation table (MRT). For the previous example, this can be visualized as Table 2.1 on page 11. For each ALU, there is an issue slot for each of the II cycles in the schedule. All operations must have a valid position in the MRT for the scheduler to succeed.

The goal of the main scheduling loop is to find a legal scheduling of the operations at the given II. The loop starts by selecting operations one at a time from a priority queue. Operations are prioritized based on a notion of height in the dataflow graph of the target code. Higher priority operations intuitively have a greater impact on the overall length of the schedule. Therefore, they are scheduled as soon as possible in the MRT, respecting any constraints based on operations already scheduled.

If scheduling an operation makes another operation no longer legal at its current position, the operation is evicted and placed back in the priority queue. This can happen when there are insufficient resources available for all operations scheduled at a particular time. When revisiting an operation, it must be placed later in the schedule than it had been previously. This requirement helps ensure that the algorithm will continue to make progress and will not oscillate. If a heuristic cutoff is reached in terms of the number of attempts to schedule operations, the II is increased to reduce the difficulty of the scheduling problem and scheduling begins again. A successful schedule fulfills all data dependencies and resource constraints, creating an execution pattern that can overlap successive iterations of the target loop body.

Modulo scheduling can provide a significant benefit by optimizing loops to maximize resource utilization of the available hardware. However, this is not always the case since useful code is rarely as simple as the example presented above. The next section covers a technique to manage more complex control flow in the target code used in conjunction with modulo scheduling.

2.3.2 *Predication*

While an inner loop without any control divergence is preferable for modulo scheduling, the majority of signal processing applications contain conditional code. If the earlier example is modified as shown in Figure 2.3, the modulo schedule must accommodate the conditional statements and also respect the semantics of the code. In order to fit all of the target code into the schedule, both execution paths must be allocated issue slots in the schedule since either is

possible. At run time, both results for c are generated and the greater than comparison generates a predicate which is used to determine the value to store. This technique is called predication. The example is shown again in Figure 2.4 converted to static single assignment (SSA) form. This form eliminates the conditional statements allowing it to be modulo scheduled. The predicate value p controls the ternary operator result assigned to c3. This represents a 2:1 multiplexer function available in the target CGRA to perform the run time selection of the result.

```
while (1)
{
   a = readStream(stream1);
   b = readStream(stream2);
   if (a > b)
      c = (a + b) >> 1;
   else
      c = a - b;
   writeStream(c, stream3);
}
```

Figure 2.3. Code example requiring predication.

Predication is useful when sections of conditional code are short because the cost associated with allocating issue slots for both possible branches of the control flow are outweighed by the ability to maintain high throughput. It is especially useful for CGRAs executing a modulo schedule because it allows conditional statements within the target code and also because it would be very costly to stop the modulo scheduled execution to handle a conditional statement. Whereas a VLIW machine can transition via prologue and epilogue code between modulo scheduled blocks and other portions of an application, CGRAs are more restricted in this capacity due to significantly larger instruction size.

```
while (1)
{
   a = readStream(stream1);
   b = readStream(stream2);
   p = (a > b);
   c0 = a + b;
   c1 = c0 >> 1;
   c2 = a – b;
   c3 = p ? c1 : c2;
   writeStream(c3, stream3);
}
```

Figure 2.4. Example converted to single assignment form.

2.3.3 *Predicate Aware Mapping*

Additional work on CGRA tool chains [Fri11] further enhances the handling of predication on CGRAs. Predicate aware SPR (PA-SPR) is a state-of-the-art CGRA mapping tool used to evaluate the Offset Pipelining approach presented in this work. It can recognize the type of conditional structure shown in the Figure 2.3 example and allow the mutually exclusive statements to coexist on the target hardware. This requires additional hardware support, but mitigates some of the overhead associated with predication on CGRA devices. Instruction memory requirements can reduce the maximum program length by a power of 2 determined by the degree of sharing achieved.

The PA-SPR concept is introduced with the Figure 2.4 example, assuming that each statement consumes a generic resource in the target architecture. The eight operations mapped to three resources results in a three cycle schedule as shown in Table 2.3. The predicate aware variation recognizes that either the c0 and c1 assignments or the c2 assignment will be executed for a given loop iteration.

Table 2.3. Modulo scheduling of above predication example.

Cycle	ALU0	ALU1	ALU2
0	Read	Read	?:
1	>	+	Write
2	-	>>	

Armed with this information, PA-SPR allows operations that are known to execute in a mutually exclusive fashion to share resources in the schedule. Table 2.4 illustrates a possible schedule that takes advantage of this capability with the subtraction and shift occupying the same schedule position. More significant sharing scenarios may reduce the number of resources necessary to schedule the application at a given schedule length.

Table 2.4. Modulo schedule with predicate aware sharing support.

Cycle	ALU0	ALU1	ALU2
0	Read	Read	?:
1	>	+	Write
2		- or >>	

Modulo scheduling with predication as demonstrated by SPR is an effective scheduling approach for many applications on CGRAs. However, as application complexity increases, the limitations of predication and a single modulo schedule become apparent. The subsequent section introduces Offset Pipelining. This technique builds on the CGRA architecture and modulo scheduling concepts presented here to efficiently execute more complex applications on CGRAs.

Chapter 3. OFFSET PIPELINED EXECUTION ON PIPELINED PROGRAM COUNTER CGRAS

Offset Pipelining increases the efficiency of mapping signal processing applications with multi-mode control flow to CGRAs. Our model considers a computation divided into subsets of the target code which we call "modes," and then organizes the operation of the modes to efficiently run on a CGRA architectures that support a mechanism for program counter distribution. A mode is composed of one or more basic blocks as defined in compiler parlance. An algorithm implementation becomes the execution of a series of modes over time, with interleaving of different mode iterations in a manner similar to modulo scheduling. We introduce Offset Pipelining here in preparation for the discussion of the tool chain and algorithms that map applications using the execution style.

```
while (true)
{
  id = readVal(stream1);
  count = readVal(stream1);
  while (count > 0)
  {
    writeVal(stream2, count);
    count--;
  }
  writeVal(stream2, id);
}
```

Figure 3.1. Code example to motivate multi-mode applications.

Consider the example in Figure 3.1 with the three colored basic blocks corresponding to modes for this example. For a modulo scheduled implementation, the inner loop in green would need to be flattened as illustrated in Figure 3.2. The *if* statement conditions would then be converted to predicates to provide the necessary control to manage the execution at run time.

```
mode0 = 1; mode1 = 0; mode2 = 0;
while (true)
{
  if (mode0)
  {
    id = readVal(stream1);
    count = readVal(stream1);
    mode0 = 0;
    mode1 = count > 0;
    mode2 = count <= 0;
  }
  else if (mode1)
  {
    writeVal(stream2, count);
    count--;
    mode1 = count > 0;
    mode2 = count <= 0;
  }
  else if (mode2)
    writeVal(stream2, id);
    mode0 = 1;
    mode2 = 0;
  }
}
```

Figure 3.2. Multi-mode code flattened for modulo scheduling.

Although the example is quite small, it illustrates the mutually exclusive execution of the modes. A conceptual illustration for a corresponding modulo schedule is shown in Figure 3.3 on the left. This and subsequent figures in this section illustrate the modes with 12, 8, and 4 operations for the red, green and blue mode respectively to clearly illustrate the Offset Pipelining concept. The schedule includes all of the operations for the complete application executing during every iteration of the schedule. This illustrates a disadvantage of modulo scheduling as the control complexity of the target application increases. We know that only one mode is executing at a time so only operations of one color are useful on any given iteration. Trying to issue instructions for only one mode would be impossible, since the modes have been spread across time and space to form the overall schedule. In fact, it is even worse than this diagram implies; under modulo scheduling the instructions for an iteration in row 1 may be intended to execute at time 1, 1+II, 1+2*II, etc.

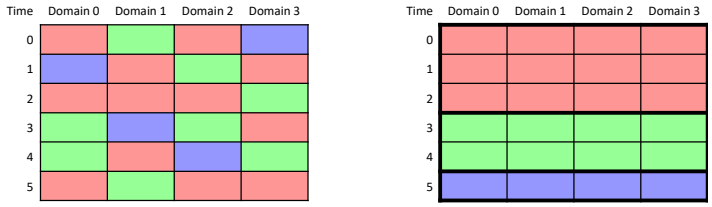

Figure 3.3. Example modulo schedule after flattening and IMS scheduling (left) and modulo schedule organized by mode (right).

However, imagine we grouped all instructions from a mode together into a specific set of rows, and an iteration completes within II cycles, as shown on the right in Figure 3.3. In this organization we could choose to run only specific cycles of the schedule, and avoid issuing instructions that are not needed at this phase of the execution. Unfortunately, this organization is impossible for most applications due to data dependencies – any specific mode iteration will have instructions that depend upon other instructions, meaning they cannot all be issued simultaneously. For example, a value read in may be transformed through a set of mathematical operations before the result is written. Performing a single iteration of a mode may require issuing a string of dependent instructions.

An alternative is to offset the start times of different ALUs, as shown in Figure 3.4. This skewing allows the system to support modes with deep sequences of instructions. The concept of a "domain" will be introduced in 3.3.4. Briefly, a domain is a set of resources that must be skewed as a unit. In these examples, a domain is composed of a single ALU. We can execute only those instructions needed for a given mode by telling each ALU which mode should be executed. In fact, in Offset Pipelining we consider the modes as separate sets of instructions, with the only requirement being that the start time of instructions in a given ALU be offset by the same amount (Figure 3.5), inspiring the name of the execution style.

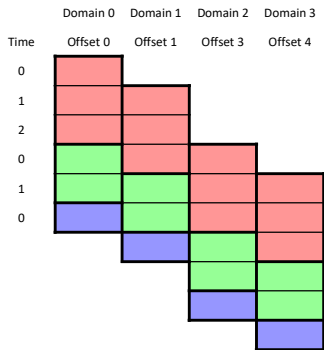

Figure 3.4. Staggering iteration start times across resources.

From an execution perspective, the lead ALU, which has the earliest issue slot, determines the mode of the next iteration to start once the current iteration ends. The following ALUs execute the same sequence with *offset* delay from the lead. The ability to stack up the different mode iterations eliminates wasted issue slots. Figure 3.6 shows an example execution trace for *count* of 2, 0, and at least 1 for 3 iterations of the outermost loop. Instead of an II of 6 for the modulo schedule in Figure 3.3, the Offset Pipelined approach has an effective II of 2. Rather than a monolithic modulo schedule for the entire application, Offset Pipelining executes each mode as required in a run time data dependent fashion. This matches the control flow of the application without the flattening and predication required for modulo scheduling. Both versions require the same total instruction storage, 6 operations for each ALU.

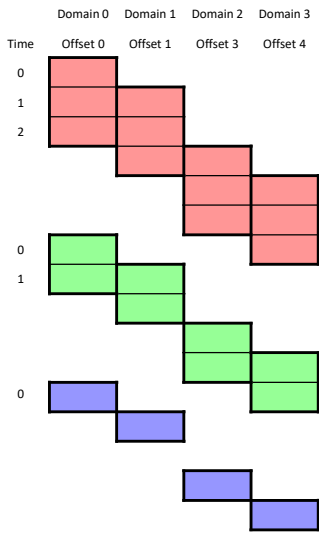

Figure 3.5. Independent mode schedules with shared staggering between resources.

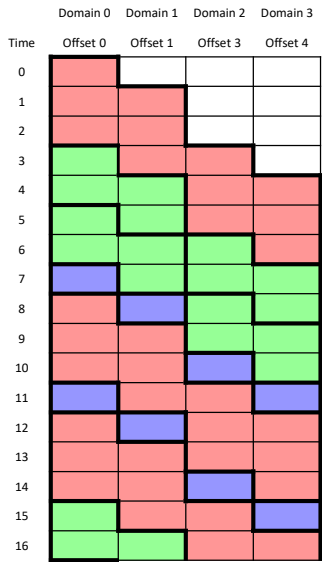

Figure 3.6. Sample Offset Pipelining execution trace of the Figure 3.5 schedule where *count* is 2, 0 and at least 1 over three iterations of the outmost loop.

Under Offset Pipelining the device is broken into domains, where a domain is a set of both logic and routing resources controlled by one program counter. This will typically be the basic tile of the spatial architecture, though several tiles could be combined into a single domain. The evaluation of Offset Pipelining assumes an architecture based on the baseline Mosaic CGRA [VE10].

A mapping is composed of a lead domain, which determines the series of modes to execute, while the rest of the domains are followers. Each follower domain has an offset, which is a statically programmed constant specifying the latency between the program counter of the lead domain and this domain. If a domain has an offset of 5, then its program counter is identical to the program counter of the lead domain 5 clock cycles previously. This offset allows for the pipelined distribution of the program counter throughout the array and supports nearly arbitrary chains of dependencies within a given mapping. Note that the offsets are programmable and are set based on the requirements of a specific application.

Each mode of an algorithm has its own II, which is the number of clock cycles each domain will spend issuing instructions for a given iteration of that mode. Thus, if the lead domain starts mode M at time T, it will spend cycles T, T + 1, … T + II_M - 1 on that iteration. Each follower domain I, with offset O_I, will spend cycles T + O_I, T + O_I + 1, … T + O_I + II_M - 1 on that iteration, as shown in Figure 3.6. In this way, the instructions for a given iteration form a

pipeline of computation through the array, pipelining both the program counter changes and instruction issuing for an iteration. These features allow Offset Pipelining to provide a scalable mechanism for executing branches in a spatial architecture.

Note that the assignment of instructions to issue slots within a mode, and the domain offsets, are statically configured during scheduling, placement, and routing of an application to an Offset Pipelined device. Compared to modulo scheduling, Offset Pipelining eliminates the overhead of issuing instructions for modes that are not currently active, which is important for a wide range of applications. However, the assignment of instructions to issue slots is more constrained due to the requirement of a single offset per domain for all modes.

3.1 PHI NODE ELIMINATION

Offset Pipelining provides a further benefit for optimizing application execution compared to modulo scheduling. Consider the *for* loop on the left side of Figure 3.7. For a standard modulo counter based architecture, the loop execution would be predicated as shown to the right. This leads to a minimum II of at least 2, since there is a recurrence loop from the increment operation to the phi node and back as seen in the top of Figure 3.8. Note that the ternary *?:* operator represents the phi node in question.

In contrast, Offset Pipelining can eliminate phi nodes whereby the invocation of modes implicitly perform the phi node functions. For example, the Offset Pipelined version of the same example is shown in Figure 3.8 bottom, showing two iterations of the inner loop (blue) within two iterations of the loop initialization mode (red). Each mode has an II of 1. Consider the input to the increment and comparison operations: when they are part of the first iteration of the loop, they receive the constant 0 from the preceding red iteration; when they are part of any other loop iteration, they receive the output of the previous increment operation on the same routing resources. In effect, this means that the phi nodes are handled implicitly by the routing fabric, where the loading of instructions for each mode automatically performs the selection function. Eliminating these operations from the dataflow graph can make an Offset Pipelined implementation more compact and can improve the recurrence limit.

```
while (1) {
  ...
  for (x = 0; x < C; x++) {
    ...
  }
  ...
}
```

```
while (1) {
  ...
  x0 = 0;
  x1 = x_loop + 1;
  p = x_loop < C;
  x_loop = p ? x1 : x0;
  ...
}
```

Figure 3.7. Inner loop predication (left) produces a ternary operator phi node in predicated version (right).

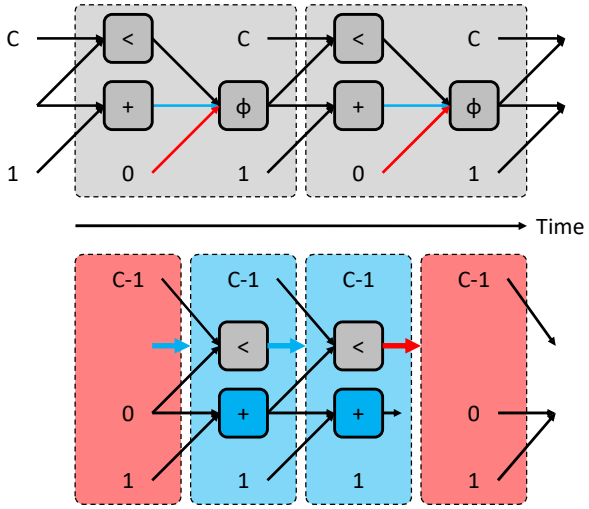

Figure 3.8. Phi nodes in modulo schedule (top), eliminated with Offset Pipelining (bottom).

3.2 Scheduling Example

The scheduling algorithm will be introduced in Chapter 5, but to provide a basis for understanding, consider manually scheduling the example of a single mode application in Figure 3.9a. The grey boxes are operations, with dataflow constraints depicted by arrows. The application has a minimum II of 3 on a 5-ALU architecture. ALUs are shown as oblongs to indicate their issue slots, with the offsets listed above. The initial offset assignments are shown in Figure 3.9a, and are set to architecture-defined minimums. This example does not represent a real dataflow graph but was instead constructed specifically to illustrate the major steps of the

scheduling process. It highlights various decisions the scheduler must make in a compact unified example.

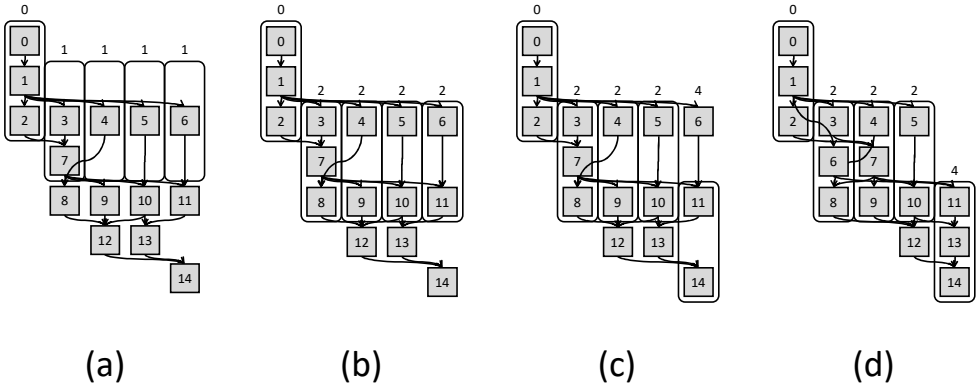

Figure 3.9. (a) Initial schedule, (b) front end shaping, (c) back end shaping, (d) rescheduling.

We employ list scheduling [ACD74], similar to IMS. We also set the offsets as small as possible and then strictly increase them as needed. Our offset increases are designed to be conservative and we only make heuristic choices if no conservative move is possible.

In Figure 3.9a, consider the earliest instructions in the application. In this example, there is only one instruction that can issue in cycle 0 and one instruction that can issue in cycle 1, there is no reason to have so many domains with offsets of 1. In fact, whenever we have a domain with unused issue slots in its earliest slots, that domain can be shifted later without degrading the schedule quality. This is a process unique to my scheduling algorithm we call front-end shaping. Figure 3.9b shows the result of front-end shaping, with the offsets now set to 0, 2, 2, 2, 2 with an II of 3.

The next thing to consider is any instruction that cannot be legally scheduled at the bottom of the diagram, e.g., numbers 12, 13 and 14. There are two possible solutions: Have a domain with offset 3 and one at 4 so instructions 13 and 14 are in the last issue slot of these two domains, or set one domain to an offset of 5 so instructions 12-14 can be scheduled, with 13 and 14 each moved 1 cycle later. It is unclear which solution is better, since this move will impact the rest of the scheduling by moving the available issue slots forcing operation 6 to move later. However, for this example we can prove that there must be at least one ALU with an offset of at least 4, and we make this clearly conservative change in Figure 3.9c. This step is a second innovation in

the scheduling process called back-end shaping. After front-end and back-end shaping, the application is rescheduled in Figure 3.9d which moves operation 6 a cycle later to an available issue slot.

Since front-end and back-end shaping are conservative transformations, there will be cases where a heuristic choice must be made, such as in Figure 3.9d. In such a case, we do offset exploration, which involves adding 1 to one domain at each offset individually to check improvement in schedule quality, measured by the number of nodes that cannot be scheduled. In this case, 0, 2, 2, 3, 4 and 0, 2, 2, 2, 5 are the two possibilities, and are shown in Figure 3.10. These two cases are generated from the 0, 2, 2, 2, 4 collection of offsets. Offset 0 is fixed and cannot be adjusted. Incrementing one offset at 2 generates the 0, 2, 2, 3, 4 case while incrementing offset 4 generates 0, 2, 2, 2, 5. The 0, 2, 2, 3, 4 case provides a complete schedule for the application and is therefore chosen. In the sections that follow we take this intuitive example of scheduling and transform it into a complete scheduling algorithm for applying Offset Pipelining.

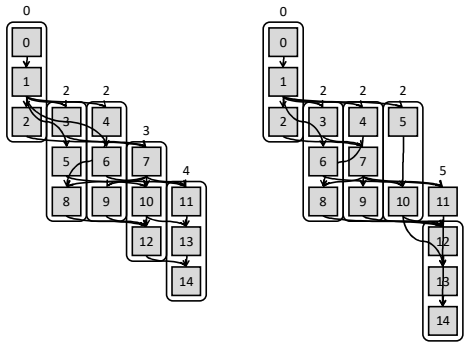

Figure 3.10. Offset Exploration with 0, 2, 2, 3, 4 and 0, 2, 2, 2, 5.

3.3 Features Related to Offset Pipelining

This section introduces concepts that describe an Offset Pipelined system. Previously introduced features of the CGRA architecture and applications are covered to fill in additional details.

3.3.1 *Pipelined Program Counters*

While this work focuses on tool chain innovations for multi-mode CGRA applications, the tools go hand in hand with features of the target architecture. The central component of CGRA's leveraging Offset Pipelining is the pipelined program counter control mechanism, which enables a scalable approach to managing control flow changes at run time. Unlike conventional multicore devices, a pipelined program counter CGRA can be configured to pipeline program counter values across the array. The device is divided into regions called domains that each contain a disjoint set of resources controlled by a single program counter. Each domain is assigned an offset that defines a window of time that operations may be scheduled on the available resources. The offset describes the relative delay between the program counter values of the lead domain and a given domain propagating through the device.

3.3.2 *Modes*

A mode is a partition of the target application code mapped to the CGRA comprised of one or more basic blocks. Consider two successive loop bodies which logically execute in sequence. Each may be considered a mode which executes in isolation. Such a division matters little in the context of a conventional processor since branching to move between basic blocks is trivial. For a CGRA, where each cycle of a schedule is essentially a very long instruction word, efficient utilization of the hardware and program length become important considerations. The added overhead of predication for complex control flow further increases the challenge. In a conventional CGRA, control logic must execute by predicating operations with side effects across the entire target code body. Modes delineate portions of the target code that are mutually exclusive and are selected for execution at run time. Offset Pipelining supports this selection to avoid executing unused portions of code by initiating an iteration of only the operations of one mode at a time.

The decomposition of the target application into modes does not limit a particular basic block to one mode, in fact it may be preferable to replicate a basic block into multiple modes where this can eliminate predicated logic. For example, a mode might make a data dependent branch to different modes following completion of a loop execution. If it is possible to eliminate this steering code by handling the final iteration in its own mode, overall performance may be

improved at the expense of replicating logic to execute the late iteration of the loop in a separate mode.

While it may be tempting to aggressively decompose an application into many modes, a practical device will be limited by the total number of instructions supported on the device across all modes as well as how branching is handled in the program counter.

3.3.3 *Scalable Mode Transition*

The pipelined program counter architecture was developed to improve execution efficiency of multi-mode applications on CGRA hardware. It also addresses practical concerns in device architecture. Moving to a multi-mode execution style requires distributing control information throughout the device. In practice, this broadcast is subject to appreciable latency. The pipelined program counter organization enables a multi-mode execution style while distributing control across the device for a scalable approach. In contrast, CGRAs geared toward modulo scheduling do not require control distribution other than a mechanism to start and stop the entire system. However, CGRAs that rely on modulo scheduled execution are not as well suited to multi-mode applications which will be explored in 5.3.3.

A significant challenge in developing parallel architectures is managing the scalability of a proposed solution. For a single mode, characteristic of an inner loop of a computation, modulo counter based control scales fairly easily to larger devices since control of the counters can be managed locally and the sequence of operations repeats and never changes. For a multi-mode application that requires the control flow to branch, there must be a mechanism available to steer the computation. In a modulo counter based system, predicated execution can provide a means to reflect control flow changes but is only practical for lightweight branches. The alternative is to adjust instruction fetching to reflect the desired execution sequence at run time. The following discussion outlines the progression towards a pipelined program counter architecture that represents the minimum set of features an architecture should support to take advantage of Offset Pipelining. The proposed architecture is practical to build and supports branching control flow to enable a wider range of applications on CGRAs.

3.3.4 *Domains*

CGRAs, conventional or those including pipelined program counters, are generally logically organized in an array of tiles. A domain corresponds to a tile in the architecture and encompasses the set of resources controlled by a single program counter, a critical concept for Offset Pipelining. A domain can be thought of as a processor in an MPPA. A program counter manages specific resources, controlling both computational and interconnect resources, and allowing configurations to change on a per cycle basis. The provision of a program counter for each domain allows skewing iteration times for Offset Pipelining. The flexibility of individual program counters also enables partitioning so that independent tasks can coexist on the device. Modulo scheduled applications can likewise be mapped to pipelined program counter CGRAs with the domain program counters configured to behave as modulo counters allowing a pipelined program counter CGRA to directly replace the modulo counter variety.

3.3.5 *Basic Device Organization*

The target device architecture for this work is based on earlier research exploring the resource composition for modulo scheduled CGRAs [VE10] as part of the Mosaic project [CFV+07]. Resources that compose a domain include a pair of ALUs, a pair of LUTs, and memory blocks. A feature of a domain is that the enclosed resources can communicate quickly, generally through a crossbar. In order to communicate between domains, signals must traverse an island style registered interconnect structure. A program counter in each domain replaces a modulo counter to provide configuration sequencing at run time. All resources within a domain are controlled by the single program counter contained within, which is an important feature of the execution model and scheduling described later.

Interconnect resources are partitioned into three separate networks to match the different types of data used by applications. The first is the word-wide interconnect that handles ALU operands, memory, and stream data. LUT inputs and outputs are served by a 1-bit interconnect network which also connects to memory write enable and ALU control signals. The last type of interconnect network is used to propagate program counter values between domains.

3.3.6 *Offsets*

The offset is a property of each domain determined during scheduling. The collection of offsets on the device sets the staggered execution of resources and provides the time needed to send changes in control flow. One domain is assigned as the leader and is set to an offset of 0 by definition. This domain computes loop and branch conditions to determine the next mode to initiate during execution. Other units follow the leader, receiving the program counter value from the leader, delayed by a number of cycles defined by the offset. Organized this way, the resources across the device form a pipeline. The leader can repeatedly execute multiple iterations of a given loop and then transition to another portion of the computation. The follower domains execute their own portions of these modes, in the same order, delayed by the fixed offset. This allows the fill and drain periods of subsequent modes to overlap. Decomposing the application into modes that execute independently with their own IIs helps eliminate wasted issue slots.

Domains in the device are organized in a tree, with the leader at offset 0 at the root. Control information propagates from the root to all other domains. The offset is the cycle latency of receiving a program counter value from the lead domain. The organization of the tree of domains factors into the placement of operations on the device and will be discussed in the scheduling and placement sections. Each domain offset must be greater than the offset of its parent, as this represents a constraint on the legal arrangement of resources. Furthermore, each domain is assigned one offset that is a constant for all modes, which allows iterations of different modes to be interleaved without wasting issue slots.

The domain offset organization provides the additional benefit of eliminating the need to broadcast control information across the device. The program counter in each domain controls the enclosed resources. Control across the device is pipelined with latency measured by the offsets relative to the leader at offset 0. A given domain can receive its control information from any domain with a smaller offset since all domains follow the same execution sequence, delayed by the offset. The arrangement of domain offsets becomes an important part of the mapping quality. The pipelined nature of the control simplifies the device architecture and aids in scalability by avoiding a complex broadcast mechanism.

The cascading control of a pipelined program counter CGRA mitigates the cost of prologue and epilogue specific code by effectively merging it with the steady state loop behavior. Each

domain executes II operations for the current mode iteration and then proceeds to the subsequent iteration determined by the lead domain. For each mode iteration M, the domains execute II_M operations for the current mode, staggered based on offset assignments. The overall program length is the sum of the IIs of all modes. In contrast, modulo scheduled CGRAs require significant extra control logic to represent prologue and epilogue behavior. The tradeoff between these two architectural styles is ultimately highly dependent on the target application. For the target signal processing applications in this work, results in Chapter 10 will demonstrate that Offset Pipelining offers a compelling execution model.

3.4 ARCHITECTURE SUPPORT

Based on the Mosaic architecture [VE10], this sections describes the changes made to facilitate Offset Pipelined execution on these CGRAs. Other architectures could be modified to add the necessary support or may already include the required mechanisms. The features outlined here are a minimum set to manage the staggered execution described in this chapter.

There are two related features that must be added to support Offset Pipelining on Mosaic devices. The first is replacement of the modulo counter controller with program counters capable of changing the program sequence. Other than the lead domain, all other program counters in the device are delayed versions of that controlling program counter so these units must also be able to receive a program counter value to retrieve domain configurations during execution. The individual program counters allow each domain to be assigned a unique offset to execute in the staggered Offset Pipelined style. The other feature needed for Offset Pipelining is a mechanism to propagate the program counter value from the lead domain out to all other domains in the device. This can be accomplished with a single dedicated interconnect channel for program counter values. The program counters and associated interconnect must support deterministic distribution of the lead program counter to ensure the Offset Pipelined schedule is respected as it relies on the fixed offset relationship and associated issue slot windows.

Chapter 4. OFFSET PIPELINING TOOL CHAIN OVERVIEW

This chapter provides an overview of and ancillary information about the tool chain infrastructure. The Offset Pipelining tool chain is organized like SPR [FCV+09] in that the flow consists of scheduling, placement, and routing to perform application mapping to the target CGRA. Additional aspects of the tool chain include the device database representation and the preparation and form of the application netlist consumed by the tools specific to Offset Pipelining.

4.1 PHASE OVERVIEW

Placement and routing are well understood components of traditional FPGA and ASIC implementation flows, while scheduling is more commonly associated with general purpose or VLIW compilation. By incorporating features of both tool flow styles, applications targeting Offset Pipelined CGRAs in this work are mapped with a tool chain composed of scheduling, placement and routing. Each phase consumes and generates a Xilinx Design Language (XDL) [SWS+11] netlist. The scheduler takes a device as the target for mapping.

The Offset Pipelining tool chain developed for this work is illustrated in Figure 4.1. While each phase builds on previous work in the literature, the implementation was written from scratch to accommodate extensive modifications required for Offset Pipelining, particularly for scheduling and routing. The tools are built on container data structures for the target device and application netlist from the Torc [SWS+11] project.

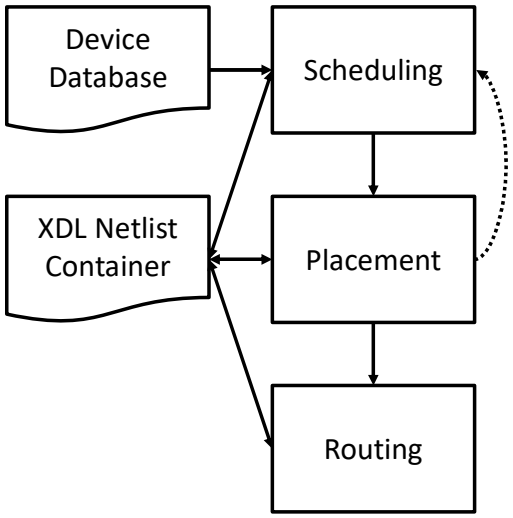

Figure 4.1. Tool chain overview.

4.1.1 *Scheduling*

Offset Pipelined Scheduling is introduced in Chapter 5. The scheduling phase of the implementation consumes the application netlist and device parameters as input. The output of scheduling consists of three interdependent items. Each mode is assigned an II value that defines the schedule length for the mode. A collection of domain offsets determines the degree of staggering among the available domains for execution based on the target device parameters. The primary output of scheduling is time slot assignment for each operation in the netlist. The mode IIs, offsets and target device are annotated at the top level of the XDL netlist, while each operation receives a time slot assignment. These annotations describe the complete scheduling of the netlist on the target device resources, but without determining physical locations on the device. While scheduling adopts concepts from previous work such as a list scheduler at its core and a height based priority scheme, fixed issue slot windows across multiple modes and the flexibility from independent IIs makes this a novel approach.

4.1.2 *Placement*

The placement phase, introduced in Chapter 6, assigns domain offsets to specific physical domains on the target device. It also places netlist operations on the domains as well. The placement always reflects the offset assignment and makes no changes to the schedule. Domain

assignments for operations and offset assignments for the domains are the only additions to the netlist made by this phase. This is largely based on the SPR placer though includes modifications to reflect the fixed issue slot windows tied to domains used in Offset Pipelining.

4.1.3 *Routing*

Introduced in Chapter 8, the routing phase adds interconnect configuration data to the netlist. Resources necessary to connect net terminals are added to the design at this stage. The complete netlist captures all the information necessary to describe the mapping of the application to the device. The router is based on QuickRoute likewise used in SPR. However, it is extensively modified to support different possible paths a signal may traverse as a byproduct of Offset Pipelining. Static routing in this style is unique to Offset Pipelining requiring a novel approach.

4.1.4 *Feedback Between Phases*

While the tool chain is usually considered a sequence from beginning to end, situations arise where a particular stage cannot make progress. The placement phase may be unable to find a solution that is even possible to route. In this situation, extra information is added to the netlist to capture problematic components of the design and the netlist is passed back to the scheduler. The scheduler provides a new solution reflecting the feedback from the placer in order to address the problem.

Similarly, feedback could be provided by the router to either placement or scheduling in order to help resolve congestion. Routing feedback is not included in the prototype tool chain because part of its evaluation is based on the channel width necessary to route designs. Targeting a specific device with a fixed channel width would benefit from the additional feedback from routing in order to complete a successful mapping of the target application.

4.2 APPLICATION PREPARATION

The Offset Pipelining tool chain, starting with the scheduler, consumes an XDL netlist representing the application dataflow graph. The tool chain represents the back end of a compilation framework for CGRAs. A complete tool chain for Offset Pipelining would include front end compilation, which is not explored in this work. In order to develop application

benchmarks to evaluate the system and provide realistic tests, code is written in stylized C. A conversion utility produces the XDL netlist from the source code and also adds annotations based on execution frequency of the modes.

4.2.1 *Single Assignment Form*

The benchmark C code is written in a static single assignment form in order to facilitate conversion to the XDL netlist. This is a manual effort that provides effective control over the resulting netlist. The process could be readily automated since single assignment form is commonly used in compilers as an intermediate representation for optimization. The C implementation results can also be compiled and executed for convenient comparison with reference code to ensure correctness.

Most operations correspond to C operators that map to an ALU or LUT. Memory and stream operations are represented as function calls that provide the appropriate behavior when executing the C program, and are mapped to the appropriate resource by the tool chain. Applications must be written to conform to the physical memory limits of the architecture. This means that any array must fit within a physical memory block available in the device. Larger memories must be constructed from a collection of physical blocks with this composition managed explicitly by the application. The ternary operator, *? :,* is used to represent phi nodes in the dataflow graph.

The variables in the C implementation represent nets in the dataflow graph. Most nets are transient, passing data between operations in a mode; others are loop carried nets, used to hold loop indices, accumulator values, or control information. Loop carried nets may have multiple source operations depending on the dataflow graph. Constants are converted into nets that do not have an explicit source. The location of a constant is determined by the router.

4.2.2 *Netlist Creation Utility*

The netlist consumed by the main tool chain is generated by an auxiliary tool that converts the C implementation to an XDL file. This tool, written specifically for this work, parses the code and performs error checking to ensure that the code follows the stylized rules expected for netlist conversion. Given the single assignment form of the C implementation, the conversion is trivial, with each line becoming an operation in the resulting dataflow graph.

Modes are annotated using C labels by the programmer. All assignment statements which become schedulable operations must appear after a label in the code. Operations are assigned to the mode corresponding to the last label parsed. The C "goto" statement is used to provide the necessary loop control. Conditional goto statements become branch operations for the lead program counter and represent transitions to other modes. An unconditional goto can be used to handle default mode transition behavior.

The variables converted to nets used by several operations correspond to nets with multiple sinks. In addition, the single assignment rule is slightly relaxed, allowing a variable to be assigned once per mode. This means that some nets may have multiple sources as well.

A net is often composed of more than a single pair of source and sink. Multi-sink nets are handled in conventional routing by seeding the search for subsequent sinks with the currently routed portion of the net. In the multi-mode routing needed for Offset Pipelining, this approach presents a challenge because a given point on an existing partial net may not be guaranteed to be live or complete with respect to a new sink being added to the net. A simpler approach is to decompose multi-terminal nets into two terminal nets for routing. This simplifies reasoning about mode transition complexity at the expense of requiring the router to allow the sharing of resources for nets with common terminals.

Limitations of the XDL format are also bypassed through multi-terminal net decomposition. Feedback from the placer to scheduler is accomplished with net annotations indicating where additional scheduling slack is necessary. Since the XDL format only allows annotations of nets and not individual terminals, a net annotation is a blunt tool. Two terminal nets allow for more targeted annotations.

One complexity introduced by Offset Pipelining is that a given path from a source to sink may be contingent on a particular run time execution sequence. This makes the traditional FPGA routing approach of extending a net from previously routed sinks difficult. The portion of a net already routed may not even be accessible to a sink being routed or an incomplete subset of possible paths to a given sink may be part of the existing path. For these reasons, all nets are decomposed into two terminal pairs for routing. The routing problem is explored in section 8.2.

4.2.3 *Mode Execution Frequency Annotations*

In moving to a multi-mode representation of the target applications, understanding the relative performance impact of different adjustments to the application is important. Some scheduling decisions are driven by mode execution frequency discussed in section 5.2.4.1. This helps to minimize the performance impact when an II needs to be increased to complete the mapping. The netlist includes annotations of how many initiations of a given mode occur during a sample execution of the code. For applications that do not have data dependent loop bounds, this provides an exact ratio for the mode execution frequencies. In other cases, the quality of the annotations depends on the data set upon which the compiled code executes. The designer may adjust these values directly to influence how the application is handled by the tools. This type of adjustment can make it easy to bias the mapping results based on expected properties of the input at runtime.

The profiling information provided to the Offset Pipelining tool chain controls per mode II increment decisions made during scheduling in order to minimize the impact of increasing the II on the overall application. While this information is an advantage for Offset Pipelining, it is not an unfair one in comparison to a modulo scheduling approach because the execution counts in question would not change the modulo scheduling. In a monolithic scheduling, there is only one II for the entire body of code.

4.2.4 *Desirable Application Features*

In order to convert a baseline single mode implementation into multiple modes, the control logic to manage the transitions between modes must be added. An application might be decomposed into modes in different ways. Modes should ideally be as independent of one another as possible. Modes are beneficial when they reduce the amount of unused code executing. However, adding a mode adds additional control logic, so it should be done judiciously, with awareness of the control implications. Modes should be compact and efficient, but adding too many will increase the control logic needed to orchestrate the execution.

Another issue is increasing II due to adding conditional branch operations necessary for mode transitions. For loops with constant bounds, pre-calculating the branch in previous loop

iterations avoids lengthening recurrence loops. The overhead of mode transition appears as more resources are added to the target device and the implementation becomes recurrence limited.

A sequence of loop bodies is a great candidate for mode decomposition. Excluding compiler transformations that may merge or interleave these neighboring loops, it is clear that while one loop is executing, others are idle. In a modulo scheduled regime targeting sequenced loops, any idle loop code consumes issue slots, potentially increasing the II. SPR supports mutual exclusion [Fri11] to alleviate this issue somewhat, but it has two drawbacks: the application must still be mapped into a single II and it consumes additional configuration memory which may add unnecessary cost. Offset Pipelining allows loop bodies to coexist on the same physical resources with the recognition that their execution is mutually exclusive, but provides for separate IIs and consumes instruction memory for exactly the number of operations needed. The overhead of Offset Pipelining is a more capable program counter unit and network to propagate control information across the device.

4.3 Architecture Generation

The Offset Pipelining tool chain targets devices represented as Torc [SWS+11] DDB objects. The Torc device database, DDB, presents an API designed to interact with the resources in commercial Xilinx FPGAs. Custom Mosaic CGRA devices were developed using a tool to generate files consumed by the Torc database build scripts. While the physical resources in the CGRAs targeted by Offset Pipelining use the DDB interface, additional infrastructure was developed to enable the necessary mode and issue slot tracking to reflect the time multiplexed nature of the CGRA architecture. The application netlist contained in XDL and the DDB object for the target device together capture all information necessary for input to the tool chain.

The device database is used in all stages of the tool chain to query the device composition. The scheduler and placer rely on the logic site resource information to build the data structures to track utilization while they execute. The router takes advantage of the site pin information to determine the source and sink locations. It then relies on interconnect connectivity information to perform the routing.

4.3.1 *Domain Composition*

The domain composition is based on previous work on modulo scheduled CGRA architectures as part of the Mosaic project [VE10]. This section provides details on the available resources. Table 4.5 summarizes the resources available in each domain.

Table 4.5. Domain resource composition.

Resource	Quantity
32-bit ALU	2
4-LUT	2
4 KB 2 port memory	1
Register file	1
Stream port	1
Program counter	1

4.3.1.1 32-bit ALU

The 32-bit arithmetic logic unit (ALU) is the primary computation element in the architecture. The interface is shown in Table 4.6 for ports connected to the general purpose interconnect. Other inputs, such as the opcode, are driven by the configuration plane set by the tools. The unit performs two input arithmetic, bitwise and comparison operations. The predicate input S allows the ALU to act as a multiplexor for selecting one of the inputs to pass to the output. Comparison operations produce output on port P which can be configured to produce a carry or overflow bit. All operations are single cycle with the exception of the multiply operation which is pipelined with two cycle latency.

Table 4.6. ALU Interface.

Port	Width	Description
A	32	Operand input
B	32	Operand input
O	32	Function output
P	1	Flag output
S	1	Predicate input

4.3.1.2 4-LUT

Two conventional four input lookup tables (LUTs) are available for handling single bit logic often used for managing control in signal processing applications. All LUT inputs and output are single bit, with the LUT configuration provided by the configuration plane. The pair of LUTs per domain is inherited from the Mosaic architecture though in that architecture they were 3-LUTs.

4.3.1.3 2 Port Memory

The 4 KB memories are available in the domains, similar to block RAM in conventional FPGAs. The port interface is shown in Table 4.7. A write enable allows data dependent control of the write port. The read port is always active without risk of unintended side effects.

Table 4.7. Memory Interface.

Port	Width	Description
A	32	Write port address
B	32	Read port address
D	32	Write port input
O	32	Read port output
W	1	Write enable

4.3.1.4 Register File

While represented as a site type in the device database, a register file is treated as a special type of routing resource rather than a logic resource, since no operations are explicitly mapped to a register file. Register files have only 32 entries and do not expose address ports because this is part of the configuration plane managed by the tools rather than the application. The write enable values are bundled with the data as a valid bit which will be discussed in the next section. While Mosaic used rotating register files which were well matched to a modulo scheduling mapping, this work uses conventional register files.

Table 4.8. Register File Interface.

Port	Width	Description
D	32	Write port input
O	32	Read port output

4.3.1.5 Stream Port

The stream port provides a first-in-first-out (FIFO) style streaming interface to and from the CGRA. It may also be used as a mechanism to communicate between partitions of the CGRA depending on the application. The read portion provides an empty flag E that may be read to ensure data is valid. The full flag F may be used to avoid overwriting data on the write half. Applications are responsible for properly reading the empty and full signals and correctly stalling execution such as described in [PH12]. Full and empty flags can be set to thresholds to allow stall signals to propagate across the device in a manner similar to program counter distribution. The stream port and memory would likely be combined in a practical device.

Table 4.9. Stream Port Interface.

Port	Width	Description
D	32	Write port input
E	1	Read port empty output
F	1	Write port full output
O	32	Read port output
R	1	Read port read input
W	1	Write port write input

4.3.2 *Interconnect Organization*

The pipeline program counter CGRAs employ a register rich, island style interconnect fabric. Within a domain, signals can be routed to be consumed in the next cycle via a crossbar as used in Mosaic CGRAs. Moving to an adjacent domain requires at least two cycles. Each multiplexor in the interconnect has a registered output, leading to the heavily pipelined execution style on which Offset Pipelining relies.

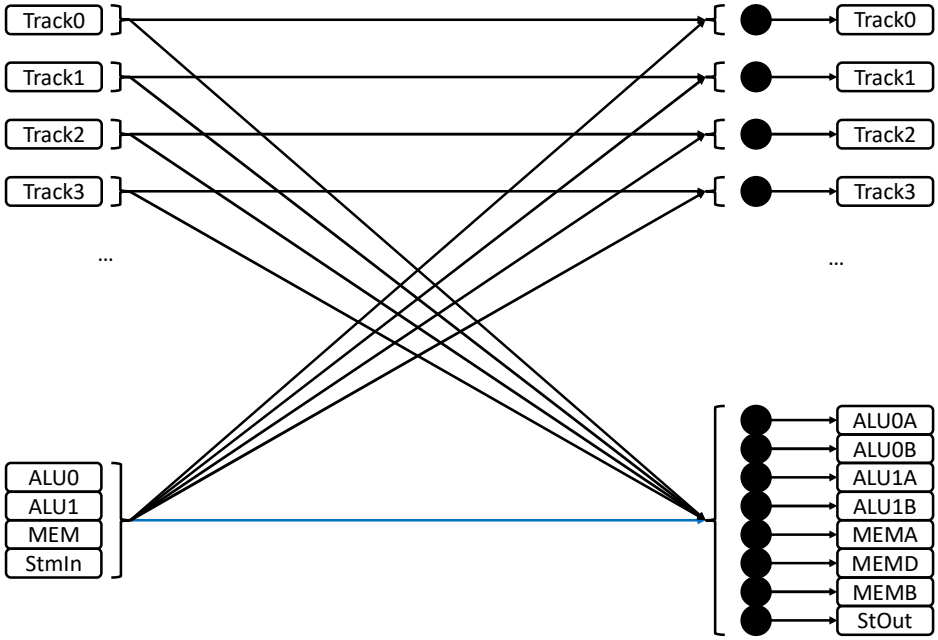

Figure 4.2. Domain interconnect organization.

4.3.2.1 Valid Bit

All 32-bit interconnect resources include an additional bit that is routed along with the data. This extra value is used as a write enable for values routed to a register file. This is necessary when a value is routed to a register file in a run time dependent manner the details of which will be introduced in Chapter 8. At run time, only one path will be valid. The valid bit ensures that the correct data is written to the register file based on the run time execution sequence.

Chapter 5. OFFSET PIPELINED SCHEDULING

In order to realize the potential of Offset Pipelining, applications need to be scheduled to leverage the staggered execution model introduced in Chapter 3. The Offset Pipelined Scheduling (OPS) algorithm must not only efficiently schedule operations but also produce per mode II values and domain offset assignments that act as constraints on the scheduling.

In this chapter, we present the details of the novel scheduling algorithm. The algorithm consists primarily of an outer loop that sets per mode IIs and an inner loop that combines operation scheduling and offset adjustment to achieve a schedule within those IIs. The core of this approach is the Offset Reservation Table, which tracks the available issue slots of all domains for all modes. The scheduling example in section 3.2 introduced the intuitive process of scheduling detailed in this chapter.

5.1 OFFSET RESERVATION TABLE

The issue slot window provided by each resource is the basis for constructing the offset reservation table (ORT) used by OPS. The ORT is analogous to the modulo reservation table (MRT) used in iterative modulo scheduling [Rau94] and is likewise used to track resource utilization during scheduling. It is constructed using the collection of domain offsets, per mode II information, and the composition of resources in each control domain. Each logic resource in the device provides II issue slots, starting at the domain offset to which it belongs. Issue slot times are measured relative to the first issue slot in a domain with offset 0. There are separate issue slots offered by a resource for each mode. An ORT is defined by the set of domain offsets and mode IIs.

An example ORT is shown in Figure 5.1. The application for this example has three modes with IIs 2, 1, and 3, respectively. There are two domains shown with offsets of 0 and 2. Note that in this simplified example, the table shows a single issue slot for each domain which assumes that the domain contains only a single resource. An ORT for the target architecture in this work includes issue slots for each schedulable resource in the domain, further differentiated by the resource type. While each mode has its own set of issue slots, the domain offset is fixed across all modes. This is a key feature of the execution model that allows different mode iterations to be effectively interleaved. Modes provide a significant benefit but must also be

applied judiciously to avoid underutilized modes that leave unused issue slots throughout the reservation table.

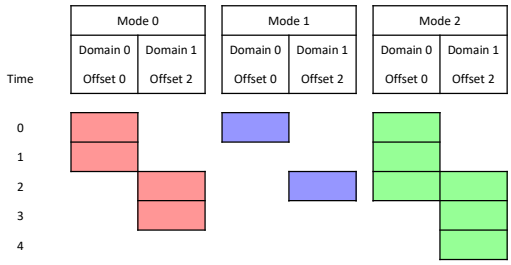

Figure 5.1. Example Offset Reservation Table.

Traditional modulo scheduling fits all operations in the target application into II cycles by placing the operations into time slots modulo the maximum schedule length of II cycles. In contrast, OPS domain offsets are set to provide issue slots at the necessary times for the operations in the dataflow graph.

5.2 Algorithm Overview

Before discussing the details of the scheduling algorithm, a brief outline is presented here to guide the subsequent discussion. A pseudocode representation of OPS is shown in Figure 5.2. OPS consists of two nested loops. The inner loop alternates between a scheduling pass and adjusting domain offsets. The outer loop controls II increments when the inner loop cannot find a legal schedule. The basic operation scheduling is a greedy as-soon-as-possible approach with a prioritization scheme based on IMS. With this basic organization in mind, the following subsections cover the different facets of the algorithm.

```
 1 initializeIIs()
 2 do {
 3    initializeOffsets()
 4    do {
 5       buildORT()
 6       if ASAPschedule(tight)
 7          return SUCCESS
 8       offsetsUpdated = offsetAdjustment()
 9    } while (offsetsUpdated)
10    IIsUpdated = incrementIIs()
11 } while (IIsUpdated)
```

Figure 5.2. Top level Offset Pipelined Scheduling algorithm.

5.2.1 *Operation Scheduling*

Operation scheduling occurs in the context of an ORT. The central concept for scheduling is an as-soon-as-possible approach that attempts to schedule the target netlist into the ORT built from the current IIs and offsets. Figure 5.3 illustrates scheduling the dataflow graph on the left into the ORT on the right. Operations are ordered by height in the dataflow graph. An operation may only be scheduled after all of its predecessors.

The core list scheduling function is modified to operate in a loose or tight mode. In tight mode, operations must be scheduled into legal issue slots available in the ORT. The loose mode relaxes this constraint by allowing operations to be scheduled without a legal issue slot if there is an unused time slot available from an earlier cycle; however, each such loosely scheduled operation consumes one of these issues slots. Loose scheduling is performed within the offset adjustment phase (Figure 5.2 line 8) because a subsequent increase in a domain offset may move unused issue slots later in order to handle the loosely scheduled operations. During the offset adjustment phase, loose scheduling primarily provides feedback. The inner loop body attempts a tight scheduling (Figure 5.2 line 6). When successful, this terminates the algorithm with a completed schedule. Failure leads to offset adjustment.

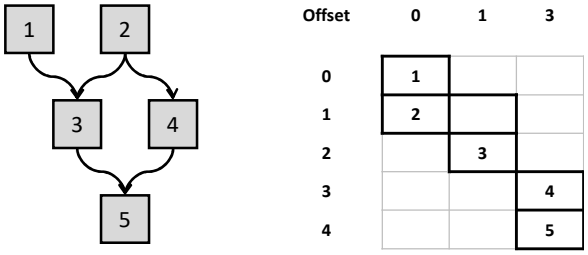

Figure 5.3. As-soon-as-possible operation scheduling.

5.2.2 *Delay Calculation and Operation Prioritization*

The delay calculation extends the technique used by IMS [Rau94] for ensuring that operation dependencies are enforced during scheduling. Dependencies within an iteration are simple; an operation must be scheduled after all of its predecessors. In modulo scheduling, the delay calculation includes an expression for dependencies that cross iteration boundaries to reflect the earliest schedule position in the next iteration. This relationship is shown in Equation 5.3 where the iteration distance multiplied by the II is subtracted from the operation delay of the source operation which corresponds to transition across the iteration boundary. Note that for intra-iteration dependencies, the iteration distance is 0.

$$Delay = SourceOpDelay - (II * IterationDistance) \tag{5.3}$$

For Offset Pipelining, the delay calculation between two operations, as measured between their inputs, is more complex than the IMS case since each mode has its own II. This delay is described by Equation 5.4 with parameters described in Table 5.10 and handles both intra-iteration and inter-iteration dependencies.

$$Delay = SourceOpDelay - SumShortestPathIIs \tag{5.4}$$

With multiple IIs, OPS replaces the II adjustment by evaluating the sequence of possible mode transitions between two operations in order to determine the smallest number of cycles between them. The sum of mode IIs of the intervening iterations between source and sink, including the source mode, but excluding the sink mode, provides the same conceptual behavior for the delay calculation as the IMS approach. However, it has been augmented to support the multi-mode execution model. This minimum distance ensures that the schedule will be correct for any run time distance between the operations.

Table 5.10. Delay calculation parameters.

Parameter	Description
SourceOpDelay	Delay of the operation driving the net
SumShortestPathIIs	Sum of mode IIs for shortest sequence of iterations from source to destination, excluding the destination mode.

When scheduling a new operation, the delay expression is added to the time slot of each operation driving the new operation. The maximum value among all inputs driving the new operation determines the earliest time the operation may be scheduled in order to ensure all inputs are available.

Operation ordering for OPS is accomplished with a height based priority scheme similar to IMS. The difference again reflects the multi-mode nature of the target netlists and leverages the delay calculation discussed to provide the criticality heuristic. The algorithm is otherwise the same as IMS; a temporary predecessor node is added to all operations in the dataflow graph and a depth first traversal labels operation priorities.

5.2.3 *Offset Adjustment*

With a scheduling mechanism in hand, the next aspect to address is setting the domain offsets. The concept of offset adjustment is to judiciously increase offsets to provide issue slots for operations not currently assigned. The offset adjustment process is done in two phases, illustrated in Figure 5.4. The first deterministically increments offsets in search of a legal scheduling via front-end and back-end shaping. If the deterministic process fails, then a heuristic offset exploration phase continues the search for a legal scheduling described in 5.2.3.3.

The interplay between as-soon-as-possible scheduling and the deterministic offset adjustment ensures that offsets are adjusted conservatively. This means that an offset will never be set later than necessary to provide issue slots to the current loose scheduling incarnation. The algorithm resorts to a heuristic adjustment only when the deterministic phase exhausts all possibilities.

```
 1 offsetsChanged = false
 2 do {
 3    ASAPschedule(loose)
 4    updated = shapeOffsets()
 5    if (updated) offsetsChanged = true
 6 } while (updated)
 7 if (!offsetsChanged) {
 8    generateOffsetCandidates()
 9    foreach (offsetCandidate)
10       ASAPschedule(loose)
11    offsetsChanged = pickOffsetCandidate()
12 }
13 return offsetsChanged
```

Figure 5.4. Offset adjustment pseudo code.

Offsets start at their architectural minimums, reflecting the earliest a control flow change calculated in the lead domain can reach each of the other domains in the architecture. Offset adjustment continues until a legal scheduling is found or if one of two failure conditions is reached. It may be that no offsets remain at 0, indicating that inter-mode dependencies cannot be met at the current II assignments. Alternatively, the latest issue slot may exceed the length of a fully sequential schedule, which also indicates the current II settings are too small.

5.2.3.1 Front-end Shaping

The shapeOffsets function (Figure 5.4 line 4) is broken into two phases, front-end shaping and back-end shaping. Front-end shaping takes the loosely scheduled netlist and attempts to increase the offsets based on the time slots assigned to operations at the beginning of the schedule. The idea is to shift offsets later when the early issue slots are unused. Since the scheduling is as-soon-as-possible, any unused issue slots occurring before the earliest scheduled operations are not needed, so the associated domain offsets can be increased without changing the schedule. This may also improve later portions of the schedule by freeing these issues slots. As with back-end shaping discussed in the next section, this process is guaranteed to be conservative.

Front-end shaping starts with a set of offsets and a loose schedule of the application onto those offsets; this guarantees that there is an issue slot for each operation either at or before that operation's current schedule. Note that this step will not change the scheduled time of any operation, which happens during ASAP scheduling only. Front-end shaping iterates through the

domains from smallest to largest current offset and assigns operations to domains. For each domain, if there is no free operation in any mode at the time of the first issue slot of that domain, that offset is increased until there is one. As many operations as possible are assigned to the issue slots of this domain. This process continues through all of the domains.

An example of front-end shaping can be found in Figure 5.5. Grey boxes represent operations scheduled at the listed times and the rectangular outlines represent the issue slots available on the domains, with offsets noted above them. The horizontal positions of the operations in the figure have no special significance other than to provide visual intuition about where operations and issue slots exist in the schedule and domain offset assignments. Since the scheduling algorithm places operations as soon as possible, there are extra issue slots at cycle 1 that can never be used by the target netlist. The front-end shaping moves these offsets later based on the earliest operations that could use these issue slots.

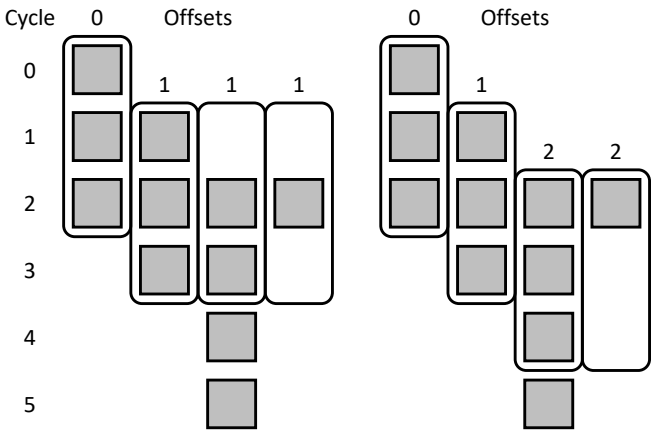

Figure 5.5. Before and after front-end shaping of a single mode with II = 3.

5.2.3.2 Back-end Shaping

Back-end shaping also increments offsets, but focuses on the operations scheduled latest rather than earliest. The process begins with all domains marked as unadjusted with their current offset settings from previous adjustment steps and all operations are marked as unassigned. The domain with the largest offset is moved, if necessary, late enough so that it offers an issue slot at the scheduled time of the latest unassigned operation. Then, for each mode M, the II_M latest

unassigned operations are assigned to this domain. The algorithm iterates until unassigned operations are exhausted.

An example of this shaping is shown in Figure 5.6: the rightmost ALU is moved to offset 4 to provide an issue slot for 14, and operations 12-14 are handled by this ALU. Though it is impossible to actually schedule operations 12-14 into a domain with offset 4, since none of these instructions can issue in cycle 4, this is a conservative, lower-bound assignment of offsets that can be heuristically improved during offset exploration described in the next section. Operation 11 and those preceding can be addressed by the other domains without further offset adjustment.

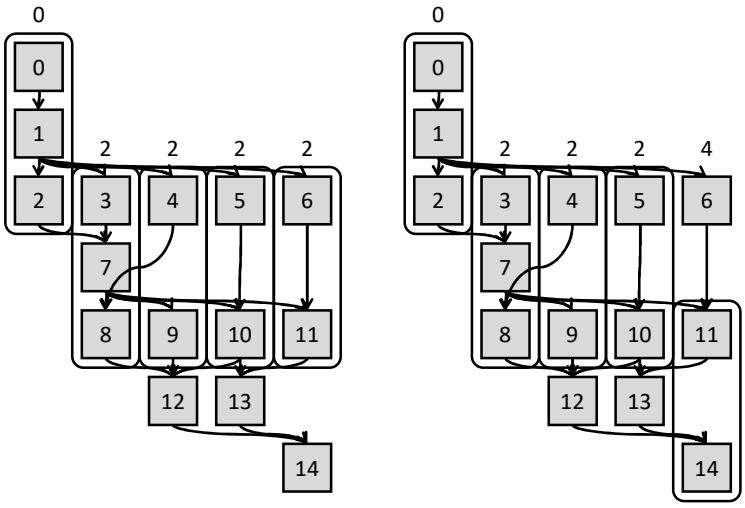

Figure 5.6. Before and after back end shaping. The initial schedule is infeasible. Subsequent scheduling passes will move operation 6 to a later cycle.

As with front-end shaping, back-end shaping makes conservative adjustments to the offsets. Consider the scenario in Figure 5.7 with a single mode application with an II of 2. This example is the simplest situation that illustrates the conservative nature of back-end shaping. There are 2 unassigned operations that are both scheduled for time *t*, all other operations have earlier time slots. The algorithm will assign a domain to offset *t-1*, which provides an issue slot at times *t-1* and *t*. Both of the two unassigned operations are assigned to this single domain, even though neither is ready to issue at time *t-1*. Recall that this is as soon as possible scheduling. This choice is made because the optimal answer is unclear. Assigning an offset of *t* to the domain forces one operation to be moved later. On the other hand, two domains with offset *t-1* provides each operation with a valid issue slot. The conservative offset assignment is taken during the

shaping portion of offset adjustment and a subsequent offset exploration step resolves the ambiguity.

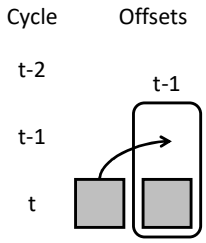

Figure 5.7. Allocating offsets at the back end.

5.2.3.3 Offset Exploration

Front-end and back-end shaping are conservative processes that will never increase offsets beyond the bounds of the as-soon-as-possible scheduling. When this deterministic shaping makes no further adjustments to the offsets, the offset exploration routine takes over. This process is unique to the Offset Pipelined Scheduling algorithm. The process evaluates possible incremental changes to the offset assignment and heuristically selects one for further shaping passes. The possible candidates are generated by incrementing each offset individually, ignoring duplicate offset configurations; e.g., if there were 3 domains with offset 6, we would only try incrementing one to offset 7. Each candidate configuration is scheduled in loose mode and the numbers of operations without issue slots (dangling operations) are tallied. The offset candidate set with the fewest dangling operations is selected. In the event of a tie, we select the candidate that increments the lowest offset. The dangling operations heuristic intuitively provides a measure of how close to a complete schedule each offset candidate set is.

5.2.3.4 Offset Initialization

At the beginning of the offset adjustment loop (line 3 of Figure 5.2), offsets are initialized to the lowest values that can be legally placed. The example in Figure 5.8 illustrates how these minimums are set for a 4x4 device with nearest neighbor program counter communication. Device sizes range from 1x2 to 5x5 for evaluating Offset Pipelining in this work with this example providing a convenient visual sample. For the devices in this work, the lead domain is placed in the middle of the device with offset set to 0. The minimum offset of all other domains

is the Manhattan distance from the lead, corresponding to the inter-domain interconnect organization of the target devices based on the Mosaic project. These values represent the number of cycles necessary to send control information to the domains in order to form the execution pipeline.

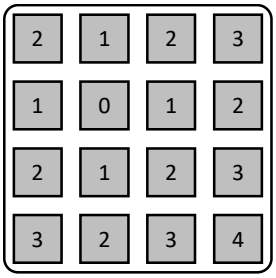

Figure 5.8. Minimum offsets for a 4x4 domain device.

5.2.4 *II Adjustment*

OPS resorts to II adjustment when the offset shaping and exploration do not yield a successful scheduling. While an application targeted with IMS has a single II that is incremented when scheduling fails, OPS must manage multiple modes with independent IIs, complicating the adjustment process. Rather than pessimistically increment all modes, OPS selects a single mode for II increment to provide more flexibility for the scheduling and offset adjustment phase of the algorithm.

When an IMS pass fails to find a legal schedule, the only option is to increment the II. In Offset Pipelining, the ability to increment the II of individual modes lets performance of the overall application degrade more gracefully. For example, an IMS application that increments the II from 2 to 3 reduces throughput to two thirds. Assuming that the same application can map to two modes for Offset Pipelining, each with an II of 2, if the application can be scheduled with one mode at an II of 2 and the other at 3, then the effective II for the application is less than 3. The value depends on the relative execution frequencies of the two modes making it clearly preferable to increment the II of the less frequently executed mode.

5.2.4.1 Selecting a Mode for II Increment

Applications scheduled with OPS include mode priority information derived from profiling the runtime behavior of the application. The frequency of execution of each mode captures how many iterations of each mode is executed for a sample data set or may also be derived from fixed loop bounds. These annotations might also be set manually by the developer such as for a PET scanner where the likelihood of events may be carefully characterized. A higher priority corresponds to a more frequently executed mode. It is desirable to minimize the IIs of higher priority modes in order to provide the best overall application performance.

To select a mode for II increment, the priority information is used to calculate an overhead value for each mode. The overhead is the product of mode priority and the ratio of current mode II to minimum mode II calculated at initialization. This calculation serves to track how much the II has been changed by the II increment process, scaled by the priority. The OPS algorithm prefers to increment the lowest overhead mode since this will minimize the impact on overall application performance. However, this preference is capped at two times the overhead of any other mode, heuristically selected to avoid skewing the IIs too far out of concern for situations where a lower overhead mode might have its II bloated but ultimately not provide additional slack where necessary during scheduling. The cap was never triggered across the benchmarks evaluated.

5.2.4.2 II Initialization

The initial IIs for OPS are calculated in a manner similar to IMS, beginning with resource limited IIs calculated for each mode. However, recurrence limited IIs require a different approach in OPS than in IMS. Each mode is first isolated by considering nets that are only connected to operations in that mode. A recurrence II is calculated for each isolated mode using a maximum cycle ratio routine as used by IMS.

In order to resolve inter-mode recurrence loops, the entire netlist is processed with the maximum cycle ratio algorithm. A positive cycle indicates that there is not enough time around the loop to accommodate the operations that comprise it. All modes that have an operation involved in a positive cycle become candidates for II increment using the II increment priority scheme discussed above.

5.2.5 *Iterating to a Solution*

The three main components of the OPS algorithm work together to search for IIs and offsets that can accommodate a legal scheduling of the target application. OPS takes inspiration from IMS in its iterative approach. The algorithm attempts to schedule the application and then adjusts the offsets to improve the fit of the netlist operations on the available issue slots. Iteration of this inner loop explores the offset assignment space. If no legal scheduling is found, the outer loop increments an II to make more issue slots available, adding flexibility to the scheduling at the expense of some application performance. This combination of scheduling, offset, and II adjustment embodies the iterative approach of OPS.

5.2.6 *Accepting feedback from placement*

As an application moves through the tool chain, it may be discovered that the original scheduling is infeasible. The scheduler can be provided feedback in the form of net annotations to insert additional delay between operations. Nets in the XDL netlist can include a property for extra delay which forces the operations to be scheduled further apart, akin to latency padding found in SPR. This allows the scheduler to address placement or routing constraints that could not be resolved by these later stages, creating a new schedule with more flexibility to work around these issues.

5.2.7 *Spare Domains*

A device may provide more resources than the application dataflow strictly needs from an issue slot perspective, perhaps due to a large recurrence II. These spare domains can be useful in the later stages of the tool chain if assigned reasonable values where the domains provide issues slots to increase the flexibility for the subsequent placement and routing phases. This is done after the main scheduling algorithm has successfully completed. The strategy employed to assign offsets to these domains starts by generating histograms of operation schedule times and available issue slots for occupied domains. Taking the square of the number of operations divided by the available issue slots (Equation 5.5) at each time provides a score that represents the occupancy and number of operations scheduled at each time. Spare domain offsets are assigned one at a time to provide the maximum cumulative score reduction. This heuristic prioritizes assigning

spare domain offsets to cover schedule times with the most operations and the fewest unused issue slots. In the event of a tie, the offset value that occurs least frequently in the collection of offset assignments is selected to more evenly distribute offset values to facilitate placement. The example in Figure 5.9 illustrates a spare domain assignment which adds a domain with offset 1. Note that an offset of 2 would provide the same score reduction, but the tie breaker favors an offset of 1. No dependency information is shown as it has no bearing on the spare domain assignment. The process evaluates a histogram of operation schedule times.

$$\frac{ops^2}{slots} \tag{5.5}$$

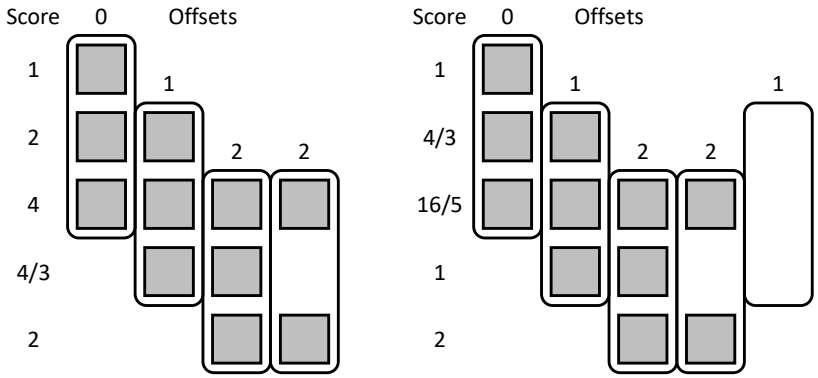

Figure 5.9. Spare domain offset assignment.

5.2.8 *Memory and Stream Operations*

Operations involving reading and writing to memory blocks in a domain require special consideration in the scheduler. While most operations are stateless, memory and stream operations do have side effects. In particular, a pair of memory operations that write and read a particular memory must reside on the same physical memory block. While the scheduler does not manage placement directly, it must still guarantee that such a placement is possible. Note that the domain memory block access latencies are fixed, accessing off chip memory must go through a stream.

In order to ensure memory operations will have a legal placement, the scheduler checks that a domain exists that can support these operations on the same resource. The first check is illustrated in Figure 5.10 on the left. The cases shown here have write (W) and read (R)

operations associated with the same memory block. If the operations reside in the same mode and are separated by more than II cycles, the scheduling will fail and require an II increment of the mode in question due to the fixed issue slot windows of Offset Pipelining. The second case on the right can be resolved by offset adjustment to provide a domain that covers both operations in the schedule.

Scheduling stream operations uses the same checks for cases where a stream is both written to and read from within the target application. This may be used for multi-kernel applications on the same device [WKY+12]. In this case, a stream is effectively a FIFO in the application. However, when used as top level IO for the application, these tests do not apply since such streams will be either read from or written to, but not both within the scope of the application.

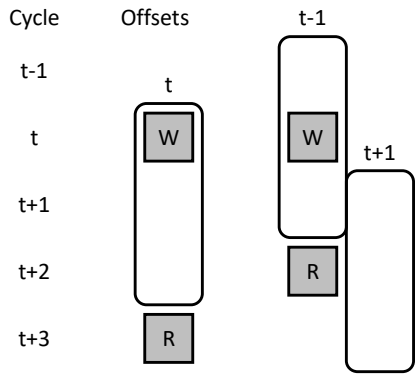

Figure 5.10. Memory operation scheduling cases.

5.3 EVALUATION

Offset Pipelined Scheduling is evaluated in comparison to IMS. The target architecture is based on work that explored resource composition for modulo scheduled CGRAs [VE10] in the Mosaic project. While the overall tool chain will be compared to SPR and PA-SPR in Chapter 10, here we focus exclusively on scheduling. Without access to the PA-SPR scheduler, the baseline SPR scheduling implementation of IMS serves as a comparison point to illustrate the advantages of Offset Pipelining for modal signal processing applications.

5.3.1 *Benchmarks*

The benchmark applications used in this evaluation are summarized in Table 5.11. These applications represent a set of signal processing algorithms typical for CGRAs. In order to compare performance between the OPS and IMS implementations, the numbers of cycles needed to execute a given benchmark are normalized to the recurrence limited cycle count of the corresponding IMS implementation. This provides insight into the performance of OPS relative to IMS and allows the applications to be compared to each other. Figure 5.11 shows results for the five benchmarks when scheduled onto four different device sizes.

Table 5.11. Applications for OPS Evaluation.

Application	Description
Bayer	Bayer filtering, includes threshold and black level adjustment
DCT	8x8 discrete cosine transform
DWT	Jpeg2000 discrete wavelet transform
K-means	K-means clustering with three channels and eight clusters
PET	Positron emission tomography event detection and normalization
RabinKarp	Hash based string matching
RSA	Encryption and decryption with 32-bit key

The applications represent a cross section of signal processing algorithms typical for CGRAs. In order to compare performance between the OPS and IMS implementations, the numbers of cycles needed to execute a given benchmark are normalized to the recurrence limited cycle count of the corresponding IMS implementation. This provides insight into the performance of OPS relative to IMS and allows the applications to be compared to each other. Figure 5.11 shows results for the five benchmarks when scheduled onto four different device sizes measured in domains.

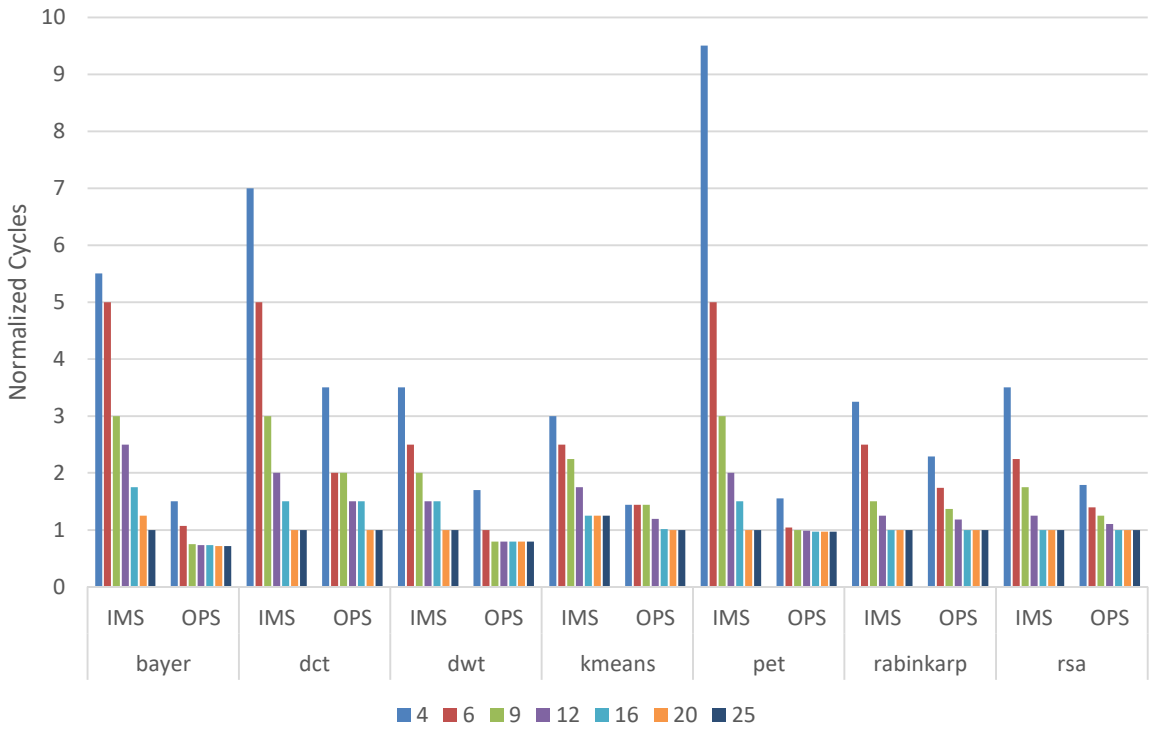

Figure 5.11. OPS vs IMS execution cycles normalized to recurrence limited II across various device sizes.

The IMS implementation reaches the recurrence limit for all applications. The DWT and Bayer implementations outperform IMS for all resource quantities because the multi-mode implementation cleanly separates a sequence of loop bodies into modes and has the mode transitions pre-calculated and pipelined. These features lower the effective recurrence limit of this implementation compared to the complex control required to coordinate predicated execution of the same code using IMS. On small devices, applications show significant improvement over IMS due to avoiding issuing unused operations from inactive modes on every iteration, making targeting more than a single loop body much less efficient for IMS. PET also sees a substantial benefit particularly on smaller devices due to unbalanced mode execution frequencies. Of the two modes in this application, the larger mode executes rarely while the smaller mode benefits from a small II and is executed very frequently.

From an architecture perspective, OPS relies on relatively small control domains rather than fewer large ones. Setting the domain offsets provides the flexibility to map the target application

efficiently and to strike a balance between mutually exclusive mode execution and more limited issue slot windows.

Note that the Bayer, DCT and DWT applications have deterministic loop bounds throughout and therefore do not depend on the actual data set. K-means and PET are data dependent. The sample data for K-means converges in three iterations and the PET dataset contains events on average every 25 samples. Despite the data dependent convergence, the modes of K-means execute with the same relative frequency, there is also cluster assignment and then cluster update. The PET application performance is data dependent and would yield different results as the ratio of samples to events changes.

5.3.2 *Scheduling Behavior*

To help demonstrate how the OPS algorithm executes, Figure 5.12 shows the offset assignment of the domains for each tight scheduling pass during scheduling. Each line represents the progression of a domain offset assignment while the algorithm is searching for a legal schedule. Recalling Figure 5.2, in this example there are two II increments that occur before the 4th and 8th tight scheduling passes indicated by the re-initialization of the domain offsets. The final offset progression finishes with a successful scheduling on the 13th iteration. From a netlist perspective, the inner loops of the DCT were unrolled to provide more computational work relative to the control overhead. The final DCT offset spacing is relatively uniform, spreading out a single iteration over a long latency, while allowing adjacent iterations to coexist in time on the CGRA.

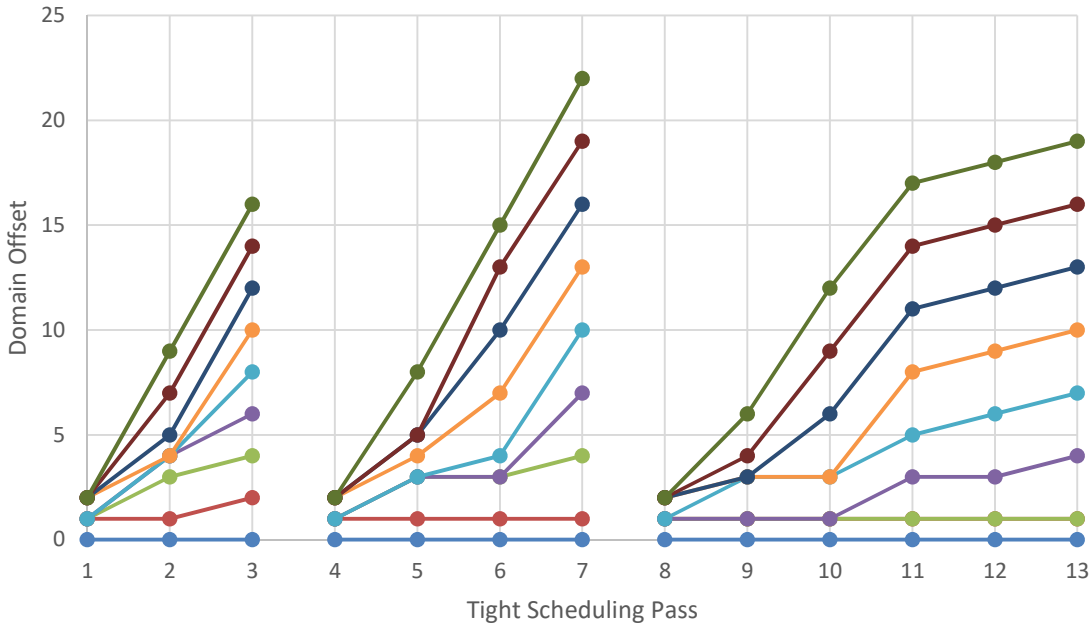

Figure 5.12. DCT offset progression example. Each line represents a domain offset assignment as it is adjusted over the course of the scheduling algorithm.

Next consider the 8-mode DWT. Figure 5.13 shows the individual mode IIs for scheduling on 1 to 10 domains. The sum of the IIs of all modes represents the total program length needed at each device size in order to hold the instructions for all modes. The line overlay is the length of the schedule for an IMS implementation. While the OPS implementation has a larger overall program size, it outperforms the IMS implementation due to the benefit of reduced execution overhead. While larger program size may seem problematic, particularly on a CGRA, the actual values for the benchmarks are modest, particularly for larger devices. A practical architecture will also have a maximum supported instruction memory size which left unused is wasted. In this case, OPS provides better performance of the application leverages available resources.

OPS runtime is fast, not exceeding approximately 10 seconds for any of the schedules generated in this evaluation running on modest commodity hardware. This is negligible compared to the runtimes of the placement and routing portions of the tool chain.

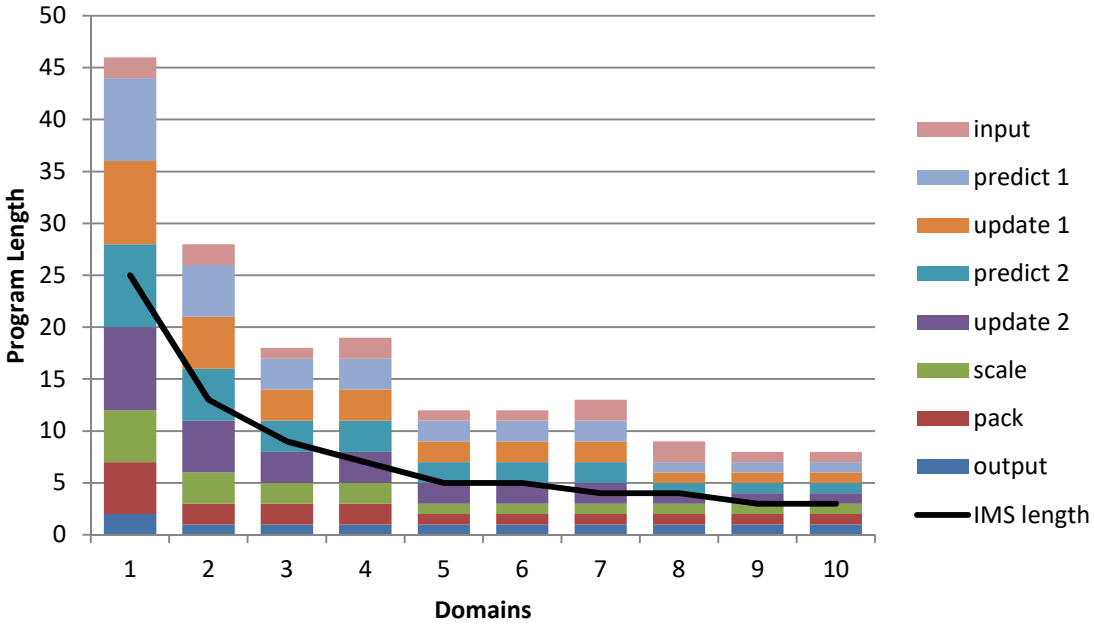

Figure 5.13. DWT mode IIs. Stacked bars are OPS mode IIs. The line is the IMS II.

5.3.3 *Results*

Figure 5.14 presents the ratio of OPS to IMS cycles to provide a speed-up metric. The geometric mean is also included, aggregating across all applications. As more resources are provided, the IMS implementation eventually reaches its recurrence limit and OPS provides no additional benefit in terms of performance. The multi-mode DWT application is an exception with the OPS version outperforming IMS due to a fundamentally lower recurrence II. Much of the predication necessary in the control logic of the IMS version is eliminated in OPS. There is a twofold advantage of Offset Pipelining. OPS can achieve the same performance as an IMS implementation with fewer resources or provide better performance with the same number of resources when resource limited.

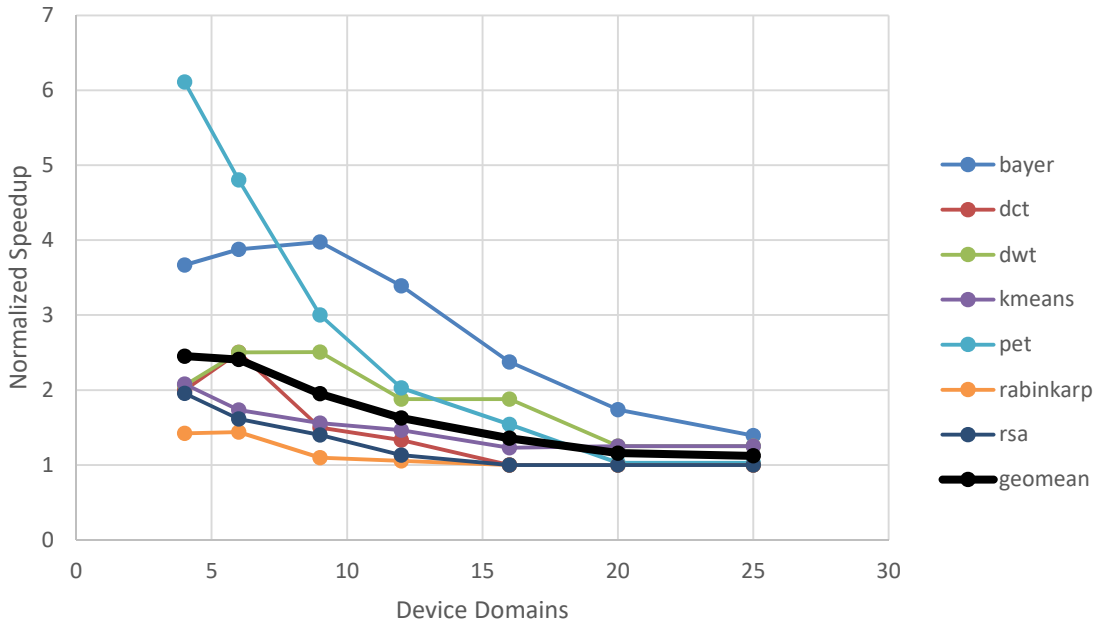

Figure 5.14. OPS vs IMS performance summary.

When provided with enough resources, the applications eventually attain their recurrence limit. In this case, there are enough resources available that, despite wasted issue slots, the application can still be scheduled for maximum performance. However, for the various device sizes where applications are resource limited, OPS provides an average speed up of 1.94 times over a modulo scheduled solution. The results presented so far represent only the scheduled applications. The following chapters detail the placement and routing phases of mapping culminating in a complete tool chain evaluation in Chapter 10.

OPS provides significantly better performance when resources are limited, but when only considering scheduling there comes a point where IMS can reach the same performance as OPS at the intrinsic recurrence loops. Since this IMS solution would be significantly larger than the corresponding OPS solution, IMS will likely have greater challenges when placement and routing are also considered. Chapter 6 and Chapter 8 introduce corresponding placement and routing phases for an Offset Pipelining tool chain.

The integration of conditional branch support for complex control flow operations significantly increases the computational density achievable on pipelined program counter CGRAs while also broadening the range of applications supportable by these systems. The OPS

algorithm automatically schedules operations, sets mode IIs, and assigns domain offsets to achieve a high performance and dense implementation.

Chapter 6. PLACEMENT

The placement phase provides the next step after scheduling in mapping an application to the device. Placement assigns operations and domain offsets to the available physical resources. This work adopts a simulated annealing approach to placement developed to support the scheduling constraints of Offset Pipelining. The scheduler guarantees that the domain offsets and per mode IIs will provide sufficient issue slots for the scheduled operations. The placer must assign offsets to the physical domains and also assign operations to issue slots in those domains. An initial placement assigns domains and operations randomly while respecting the scheduled time slots. This is followed by the annealing phase to improve the quality of the placement and follows the VPR [BR97] cooling schedule. Different move types and the cost function formulation are the major features of the placement phase specifically developed for, and unique to, Offset Pipelining.

6.1 MOVE TYPES

The move function is responsible for making changes to the placed design in order to explore the space of possible placements during the annealing process. There are two types of moves made for placement: an operation move and an offset move. The move type is selected randomly, weighted by the proportion of movable items in the design of each type. By selecting the move type randomly, all moves made during annealing are managed by the VPR cooling schedule which avoids introducing additional heuristic decisions in the placement process. Both of these move types preserve the schedule constraints while exploring possible placements for the scheduled netlist.

6.1.1 *Operation Move*

An operation move procedure randomly selects an operation and then selects a legal destination based on the schedule constraints. An operation may be moved to an issue slot at the same time in the appropriate mode. For example, operation A in Figure 6.1 can move to domain 0 or 2 while remaining at time 1. This corresponds to moving along the row at its scheduled time. Some operations will be more constrained than others depending on the number of issue slots

available at a given time, a result of offset assignments made during scheduling. Operation B only has two legal positions while operation C cannot be moved.

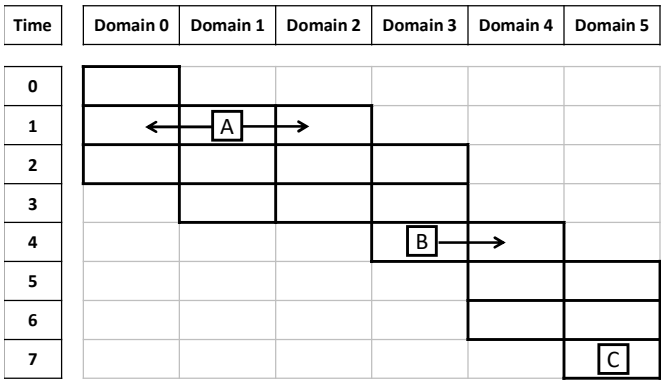

Figure 6.1. Offset Reservation Table demonstrating operation mobility during placement.

If the selected destination is occupied, the operations are swapped. This approach guarantees that the scheduling is respected after any move and operations remain legally scheduled. Note that while operation C cannot be moved through an operation move, it may be moved through an offset move discussed next.

6.1.2 *Offset Move*

The offset move swaps the entire contents of a domain, including the offset and all operations. The pair of domains are selected randomly, the essence of simulated annealing, allowing for full exploration of the design space. Moving the offsets alone would not be feasible because the offset defines the specific issue slot times available on the domain. The example in Figure 6.2 illustrates the effect of swapping domain offsets on a simple linear architecture. In this case, the offset reservation tables as seen in Chapter 5 become an explicit representation of the physical arrangement of resources and the associated issue slots. Moving offsets is an important piece of the placement optimization because it allows a larger block of operations to move as a cluster. This helps the placer avoid local minima due to highly connected groups of operations, where moving a single operation would never be favorable from a cost perspective. For cases where the issue slots on a domain are the only ones at a particular time, the offset move is the only way to move these operations. Note that in the example, domain 3 is now the lead domain since its

offset is zero. Offsets are initialized as described in 5.2.3.4 to reflect the minimum offsets possible in the architecture.

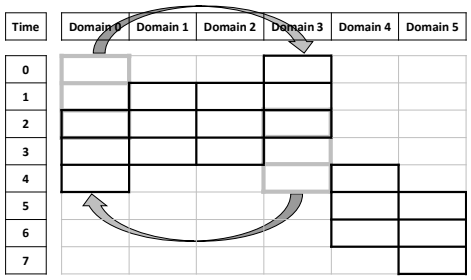

Figure 6.2. Illustration of a domain swap.

6.2 COST FUNCTION

The cost function distills the quality of the placement to a value for evaluating the progress of the algorithm. In order for the placement to be viable for routing, it must be possible to route each signal in the application. Dealing with congestion is left to the router, but the placer will not complete successfully until all signals can individually be routed. The cost function aggregates over each source/sink pair the difference between the best case route latency and the required schedule latency, illustrated in Figure 6.3. When the slack term is positive, the pair of terminals cannot be routed within the required latency. For example, if the minimum latency of the placement is 3 but the schedule requires a latency of 2, the net cannot be routed so the cost is multiplied by ten to encourage further annealing improvement. The factor of ten cost adjustment is empirically selected to severely penalize illegal nets while still allowing them to be probabilistically accepted during the annealing process. Scaling by ten ensures that a net that is short even a single cycle will outweigh the negative slack of a net that has sufficient latency to transit corner to corner in a 5x5 domain device in nine cycles. A successful placement minimizes the cost function with no net violating its required latency such that each source and sink pair in the design can be routed. Negative slack terms minimize wire length as a secondary goal, favoring nets shorter than the latency requires.

```
cost = 0;
foreach (source:sink pair) {
  slack = MinPlacedLatency(source, sink) - ScheduleLatency(source, sink);
  if (slack > 0) slack *= 10;
  cost += slack;
}
```

Figure 6.3. Placement cost function applied to each source/sink pair.

The cost function also includes a term that drives organization of the domain offsets. A legal placement requires that each domain, other than the leader at offset 0, must be adjacent to a domain with a smaller offset in order to propagate program counter values for Offset Pipelining. Figure 6.4 illustrated an example domain offset arrangement following placement. Note that each domain receives the program counter value from an adjacent domain with a smaller offset creating a rooted tree from the leader at offset 0. Figure 6.5 shows the portion of the cost function that allows the annealing to optimize the offset arrangement. If the smallest adjacent offset is less than the domain in question, the cost contributed is just the difference between the offsets to encourage a sequential arrangement. Otherwise, the domain does not have an adjacent domain it can receive the program counter value from. This situation is penalized by the offset difference multiplied by the total number of issue slots available on the domain representing the need to move the entire domain contents to improve the placement.

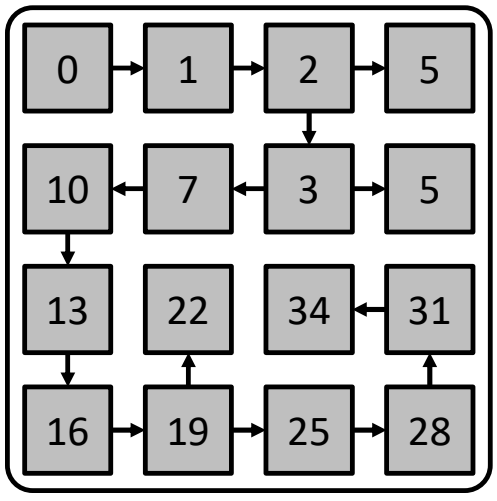

Figure 6.4. Example domain offset assignments after placement showing program counter propagation.

```
foreach (domain) {
  if (domain.offset == 0) continue;
  smallestNeighbor = getSmallestAdjacentOffset(domain);
  offsetDifference = domain.offset – smallestNeighbor;
  if (offsetDifference <= 0)
    cost += (1 - offsetDifference) * domain.totalIssueSlots;
  else
    cost += offsetDifference;
}
```

Figure 6.5. Placement cost function for domain offset arrangement.

6.3 Feedback to Scheduling

If the placer cannot find a routable placement, there is no use in running the router since it would fail. In this case, nets that cannot meet their latency requirement are annotated with the amount of extra delay needed along that path and the design is scheduled again as noted in 5.2.6. The scheduler produces a new solution recognizing the added latency requirements for the previously failing nets to provide more flexibility in placement in order to find a routable solution. The design is passed to the router when a suitable placement is achieved.

The initial scheduling pass often is too optimistic in terms of offset assignments. For the benchmark applications in this work, an average of four scheduling and placement passes execute before a routable placement is achieved. The number of passes tends to increase when IIs are small or when the application is recurrence limited with an abundance of resources.

6.4 Placement Examples and Behavior

The effectiveness of the placer is seen by observing that initial placements are not routable while the final successful placements are routable assuming sufficient interconnect resources. In this section we qualitatively examine placer behavior. It is difficult to visualize scheduled application dataflow graphs for Offset Pipelining. A detailed visualization must illustrate the spatial dimensions of the architecture, the temporal dimension of the schedule, and the additional temporal dimension of modes. We instead take a high level view by looking at the resulting offset arrangement after placer execution.

The example in Figure 6.4 shows a domain offset placement for a Bayer filtering benchmark on a 4x4 device. An initial random offset arrangement is organized in the final result. Offsets are generally placed adjacent to others in the sequence of a sorted list of the offsets. This means that the offsets can be visited in increasing order starting from offset 0 constructing a rooted tree of domains in the device as shown. The placer occasionally even produces an offset assignment in a spiral or zigzag arrangement though this is not the case in the example here.

A path formed by the offset sequence in the resulting placements makes sense from a routing standpoint. Dependencies are most often among operations that are scheduled close together. Keeping sequential offsets close together provides issue slots at a given time that are physically close together to facilitate routing as modeled by the cost function. This can be seen in the scheduling examples in Figure 6.6. If we consider a one-dimensional architecture, the diagram on the left represents a placement where offsets are arranged in an increasing sequence whereas the right shows a random ordering. Operation 12 depends on 9 and we can see that the random placement will not allow the signal to be routed with that scheduling and placement since the signal would need to traverse the entire device in a single cycle. However, the placement on the left places these issue slots close together making routing for that signal possible. The cost function drives both the arrangement of domain offsets and operations on the available issue slots in order to optimize the overall placement through the simulated annealing framework.

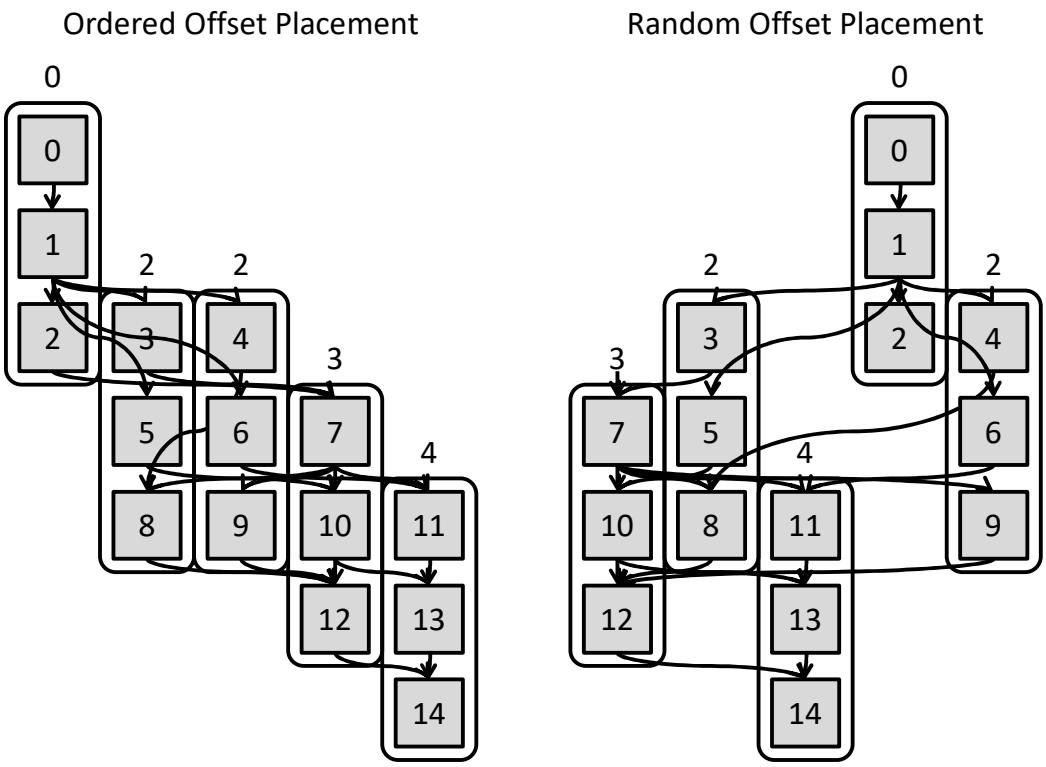

Figure 6.6. Routing considerations for a placed netlist.

Chapter 7. PIPELINED ROUTING

It is important to understand the routing problem for conventional modulo scheduled CGRAs before introducing EveryTime routing for Offset Pipelined systems in Chapter 8. This chapter provides an overview of existing techniques providing background information exclusively. It is assumed that the reader has a basic understanding of applying shortest path algorithms to conventional routing. Relying on this foundation, we introduce existing work on negotiated congestion for global routing alongside pipelined routing.

The PathFinder [ME95] negotiated congestion algorithm for global routing is reviewed first before modifications are introduced to support the Offset Pipelining execution model in Chapter 8. QuickRoute [LE04] provides the main framework for the EveryTime signal router. Lastly, an overview of PipeRoute [SEH03] introduces the phased search concept used by the EveryTime router to support certain net types in an application.

The entire collection of routed nets for an application must coexist on the available resources. The negotiated congestion approach pioneered by PathFinder resolves resource contention through incremental cost adjustments and iterative re-routing of the design. This technique is particularly well suited to optimization across multiple competing objectives.

Conventional routing for an FPGA involves determining the specific path each signal in the design will take to connect source and destination. The pipelined routing problem adds a latency requirement for each path. Whereas an FPGA route is evaluated based on the delay along the path, a pipelined route visits registered interconnect points that extend a path across multiple cycles in the resulting design. This problem cannot be solved with efficient shortest path algorithms. The QuickRoute and PipeRoute algorithms provide solutions that have addressed this problem.

7.1 PATHFINDER

In addition to routing individual nets, a mechanism is needed to orchestrate the global routing of the design. Trying to route each net in turn and allowing no resource reuse is problematic. Since the earlier nets effectively have priority, the later nets must avoid conflicting with those previously routed. This leads to the issue of determining the best order to route the nets of the

designs to achieve a successful routing. PathFinder addresses the problem by allowing the conflicts to occur and systematically increasing resource costs throughout the process.

$$c_n = (b_n + h_n) * p_n \tag{7.6}$$

PathFinder defines the cost of a node as shown in Equation 7.5 where b_n is the base cost of using the resource n, h_n is a history cost incremented during routing, and p_n represents the number of nets occupying the resource n. The base cost for routing is often a delay term, while in pipelined routing the latency of the resource is generally prioritized. The p_n and h_n terms will be explained with two examples described in the PathFinder paper [ME95].

The example in Figure 7.1 has three sources S_i and corresponding sinks D_i. There are three routing nodes, *A*, *B*, and *C,* with the base costs labeled on the graph edges. Individually, the minimum cost path for each signal would route through *B*. If the router is only capable of avoiding conflicts with paths already routed, then the order that the signals are routed will determine which signal will use *B*. This may lead to a globally more expensive solution or even fail to route all the nets. Only when signal S_2 is routed first is the optimal cost solution achieved. Consider three scenarios routing the three signals S_x to the corresponding destinations D_x, with each choosing the lowest cost route available:

- Routing order S_1, S_2, S_3 leads to S_1 on B, S_2 on A, and S_3 on C with a total cost of 14.
- Routing order S_3, S_2, S_1 leads to S_3 on B, S_2 on A and S_1 is unroutable.
- Routing order S_2, S_1, S_3 leads to S_2 on B, S_1 on A and S_3 on C with a total cost of 12.

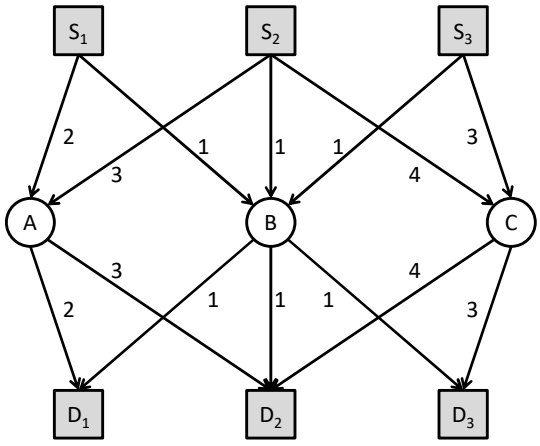

Figure 7.1. Example of first order congestion.

The p_n term in the cost function alleviates this *first order* congestion, providing the ability to resolve any ordering of the signals during routing. For the first routing pass, all signals use *B*. With contention on this resource, signals see an increased p_B value, which promotes exploration of alternative paths. There is no explicit conflict between nets during routing; instead, the cost function is adjusted to encourage nets to avoid a congested resource.

A second problem the PathFinder formulation addresses is *second order* congestion. In these cases, congestion resolution requires nets not involved in a conflict to move to different resources. Such an example is illustrated in Figure 7.2. If the signals are initially routed in order 1, 2, 3, S_1 will occupy *B*, and both S_2 and S_3 will pass through *C*. Since S_1 is not using a contested node, it will never be rerouted away from *B* to make room for S_2 to use *B*. S_3 would need to be routed first in order for the routing of S_1 to find the alternate route through *A* in a conflict avoidance style.

The h_n term is increased during each routing pass that a resource *n* is contested. As this added cost builds over multiple passes, S_2 will eventually take the less expensive path through *B* and subsequently S_1 will move to *A,* making room for S_3 to use *C*. The impact of h_n is a permanent increase in the cost of the node to encourage the router to explore other paths.

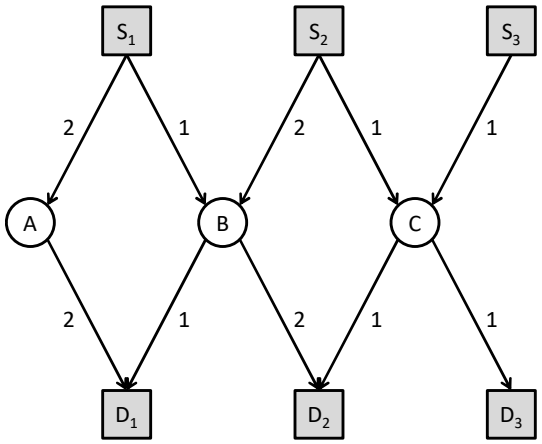

Figure 7.2. Example of second order congestion.

Negotiated congestion resolution is a powerful tool helping ensure that resources go to the nets that need them most. It is also an architecture adaptive solution that avoids the pitfalls of a global routing approach that relies on ordering the nets.

7.2 QUICKROUTE

The difficulty with conventional shortest path algorithms for pipelined routing is that once a node is visited by the search, it is assumed that it was found by the shortest path and can only be visited once by the algorithm. In pipelined routing, the first time a node is visited may not be part of a path that reaches the destination with the correct latency. Even if it does reach the destination at the right time, the signal may not have traversed the shortest path to get there.

The example in Figure 7.3 illustrates the limitations of a conventional routing approach when applied to pipelined routing. The net being routed has a source *S* and destination *D* with a signal flight time of 2 cycles. In this case, we assume that resources *A* and *B* each have a latency of one cycle, so both must be visited to reach *D* at the correct time. In a conventional shortest path algorithm, both *A* and *B* would be visited directly from *S* and would not be revisited. This would not explore the path from *A* to *B,* which must be used to meet the 2 cycle flight time requirement. A brute force approach to addressing this shortcoming lets the search revisit nodes to ensure that all possible paths are explored. The shortest path with the correct latency will be found using a brute force search but at the cost of exploring an exponential number of paths.

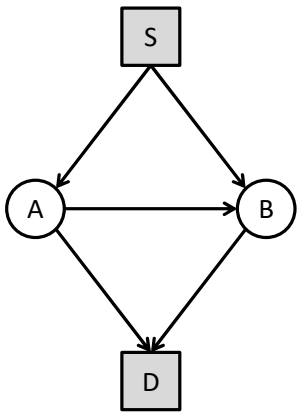

Figure 7.3. The pipelined routing problem

Instead of tracking every possible path, the QuickRoute algorithm caps the number of times a node may be visited during the search. A node may be visited *k* times at each latency *l*. For example, with $k = 2$, a node may only be visited twice at $l = 1$, may be visited up to two times at $l = 2$, and so on. Limiting the number of times a node can be revisited significantly prunes the

search space. Even with a heavily pruned search, the authors found that limiting QuickRoute to $k = 1$ produced superior results to prior work on PipeRoute.

7.3 PIPEROUTE

The predecessor to QuickRoute, PipeRoute introduced an optimal routing approach for finding a path with one cycle of latency from source to sink. This 1-Delay router was further generalized for arbitrary delay paths and pipelined multi-terminal nets. We focus here only on an aspect of the 1-Delay router as it inspires a feature of the EveryTime router.

The phased search is the key concept used in PipeRoute to visit a registered resource along the path. Consider the example shown in Figure 7.4 routing from S to K. There are two registers available, R0 and R1, and A through H are other interconnect resources. The router must find the shortest path that visits R0 or R1 before reaching the destination. Routing begins with a phase 0 search at S. The 0 above S indicates that the node was visited during the phase 0 search. In Figure 7.5, the neighbors of S are visited during phase 0.

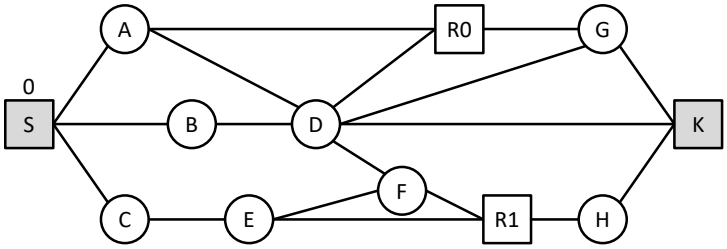

Figure 7.4. Beginning route from S to K with phase 0 search.

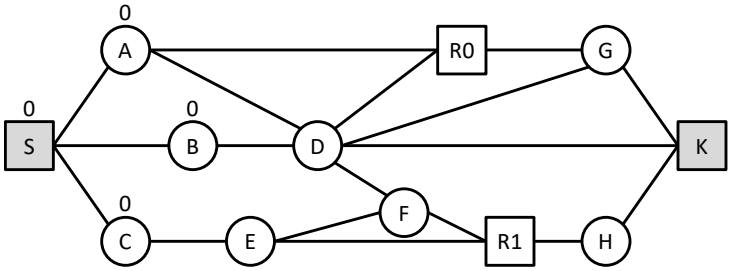

Figure 7.5. Neighbors of S explored during phase 0.

At the next expansion, register R0 is visited in Figure 7.6. The “fromA” annotation indicates that the node was visited from A. This is important, assuming an undirected graph, to

avoid reusing the edge that reached the register along the explored path. Visiting a register begins a phase 1 search from that point. Figure 7.7 shows the expansion from R0 visiting G during phase 1 while G is also visited from D in phase 0. The destination K is visited from D as well, but since it is from a phase 0 search, no register exists along the path and the search continues.

While PipeRoute describes the pre and post register portions of the search as phases, this is somewhat confusing. Both phase 0 and phase 1 portions of the search proceed simultaneously depending on whether a register has been visited along the explored path. This is evident in Figure 7.7 where G is visited by both phase 0 from D and phase 1 from R0.

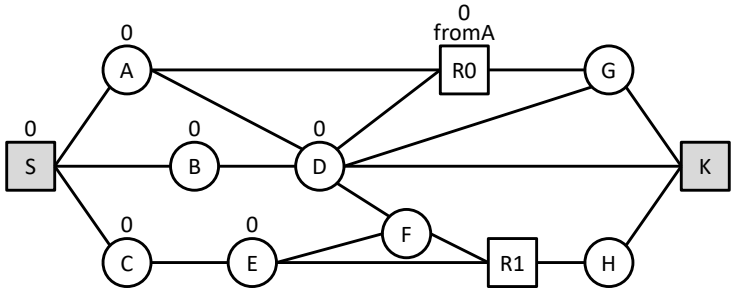

Figure 7.6. Next level expansion at phase 0 discovers register R0.

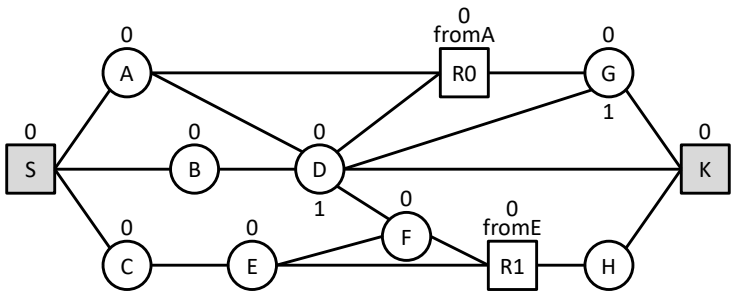

Figure 7.7. Phase 1 search begins from R0 while phase 0 search continues, finding R1.

The final step for the example is shown in Figure 7.8 with the path via A, R0, and G highlighted. The first time the destination is visited during the phase 1 search guarantees that the path is the shortest and also includes a single register. The EveryTime router adapts this phased search concept to QuickRoute to visit register file resources along a path for particular net types and is described in section 8.7.1.

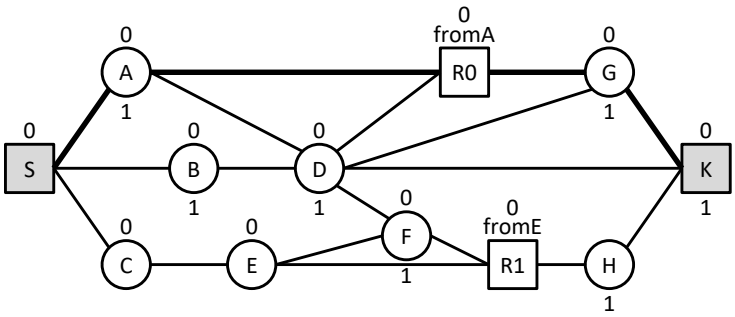

Figure 7.8. K is visited during phase 1 search that started from R0.

PipeRoute also assumes that registers can be bypassed. In conventional FPGA pipelined routing, this is important since the router is usually trying to meet a particular timing specification. This means that the phase 1 search can continue through subsequent registers visited on a path and still meet the 1-delay requirement. Unlike FPGA routing, Mosaic CGRA routing is strictly pipelined and registers cannot be bypassed.

Chapter 8. EVERYTIME ROUTING

In Chapter 6 we developed a complete placement approach for Offset Pipelined devices. We now turn to the challenge of routing in these devices. As done for an FPGA, routing must be precomputed with only one signal allowed to use a given resource at a time. However, the new requirements of an Offset Pipelined system introduce complexities that require special handling in the routing algorithm.

To aid in this discussion, we first present two styles of diagrams that will be used to illustrate the challenges of these systems and the algorithmic innovations we have developed to solve them. We then provide an overview of the EveryTime router before going into the full details of the algorithm.

8.1 ROUTING ABSTRACTIONS

During the placement discussion, we presented tables with domains given as columns and timeslots as rows. For routing, we will extend these with a simplified routing structure used to help illustrate points throughout our routing discussion. As shown in Figure 8.1, we consider a simple one dimensional architecture with single cycle routing available between adjacent domains. Signals that travel longer distances must do so over multiple clock cycles. The real architectures we consider are more complex, but this abstraction is sufficient for discussion purposes. A line traversing a box in the figure represents a mux configuration within the enclosing domain. Registers are at the horizontal boundaries between boxes, making the cycle boundaries visually obvious.

The other concept that will be important for subsequent discussions is the sequencing of mode execution on the device. That is, what are the possible execution sequences for a given application? This is not a single fixed trace, since the lead domain in an Offset Pipelined application can dynamically determine the next mode to execute. Instead, it is an execution graph that indicates the potential orderings of modes for a specific application. An example of a mode transition diagram is shown in Figure 8.2, which is representative of a simple loop with preamble mode A of II = 3, loop body B of II = 1 executing zero or more times, and epilogue C of II = 1.

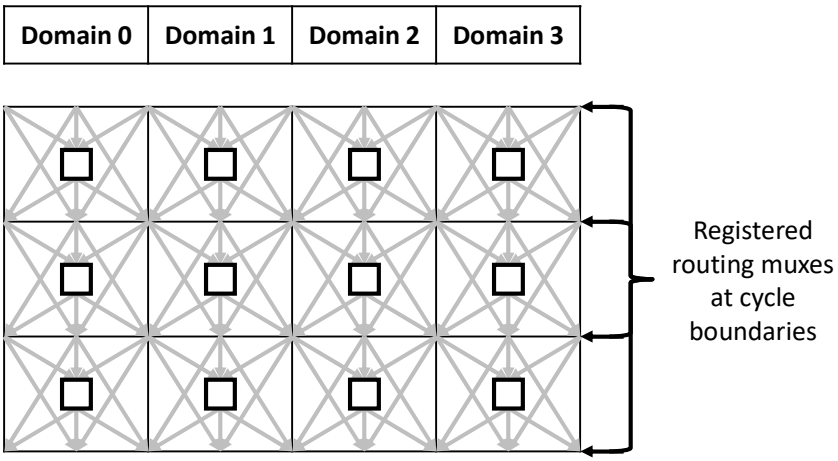

Figure 8.1. Simplified routing architecture

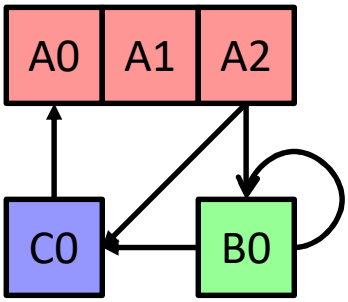

Figure 8.2. Example mode transition diagram.

8.2 The Offset Pipelined Routing Problem

In many ways, the challenge of routing for an Offset Pipelined device is similar to the challenge of routing for an FPGA or a modulo counter based CGRA: signals must be sent from source to sink in an efficient manner through a predefined interconnect, and congestion between signals must be resolved. However, there are several unique features of Offset Pipelined systems that require careful consideration and innovation to solve.

To illustrate each of these issues, consider the example in Figure 8.3, the same example used for introducing the Offset Pipelining execution model in Chapter 3. The code includes a preamble and a fast inner loop. Note that the IIs in the mode transition diagram were selected for illustration purposes. Although the code looks fairly straightforward, it raises several complex issues for a routing algorithm to solve for Offset Pipelined execution:

- Nets with multiple sources
- Routing resources that exist in multiple modes
- Nets traversing distant portions of the iteration space
- Nets with unknown flight time

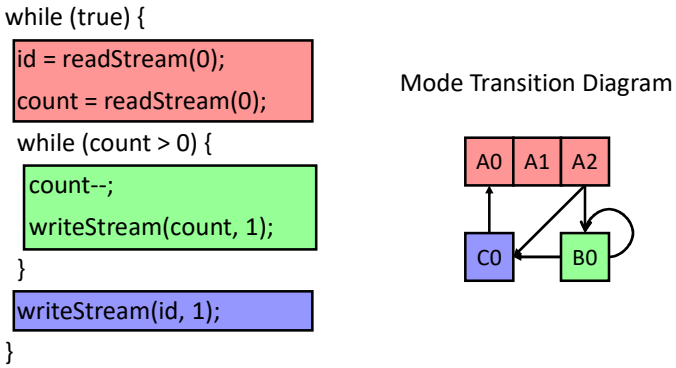

Figure 8.3. Example code and mode transition diagram.

8.2.1 *Nets with Multiple Sources*

Consider the signal *count* in the example code. The value of *count* is created both in the preamble via the read of a stream and in the loop body via the decrement operation. This means that during routing, the net actually has two sources. One could simplify this by inserting explicit phi nodes, which would become multiplexor functions computed in the functional units. However, since the while loop would then contain a recurrence loop from the phi node to the decrement and back, this would inevitably increase the II of the inner loop. We instead allow for multiple sources and leverage the mode invocation order of Offset Pipelining to handle the path selection implicitly via the routing. This situation is illustrated in Figure 8.4a, depicting an iteration of mode A followed by two iterations of B, and so forth.

8.2.2 *Routing Resources May Have to Exist in Multiple Modes*

In Figure 8.4b, we show what happens when mode B executes twice and highlight the routing of the *count* signal from the decrement operation to the writeStm operation within each of the two

iterations. Notice that the second *count* routing stays completely within the green mode B, while the first *count* goes through a domain executing the last cycle of the red mode A. Even if we re-routed the *count* signal to go down first, it would still sometimes go through mode B and other times go through mode C in blue. That the routing of a single signal may require configuring resources in multiple modes simultaneously is a fundamental requirement of the Offset Pipelining execution strategy and requires careful design of the routing algorithm.

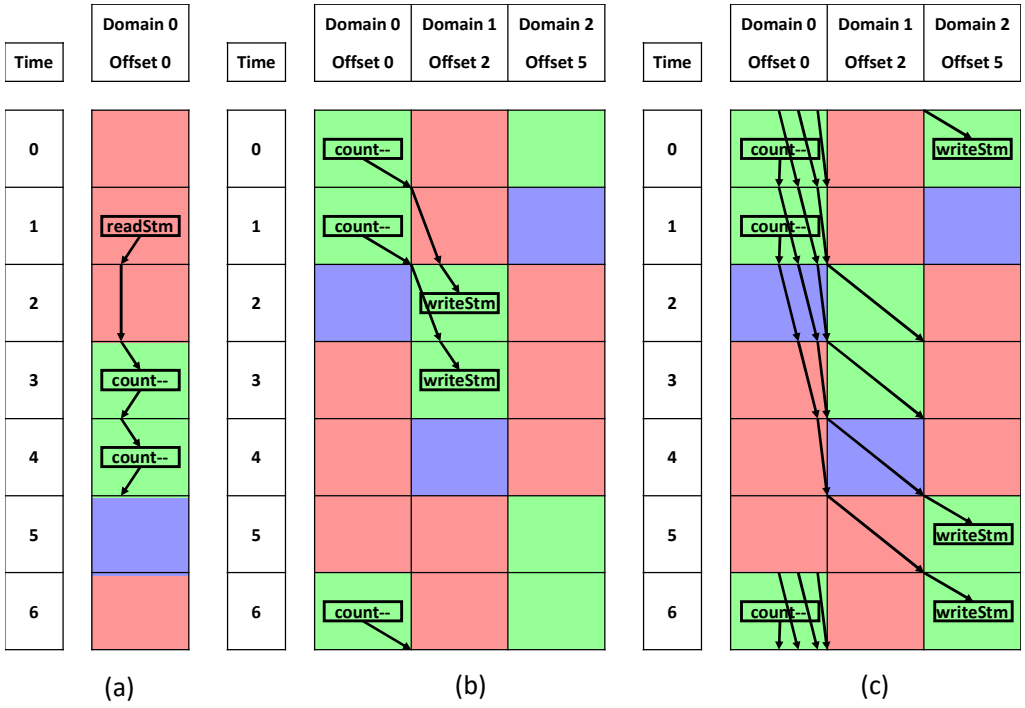

Figure 8.4. Example execution traces for variable *count*: (a) multiple source net; (b) traversing resources in multiple modes; (c) moving through resources several iterations away from source and sink.

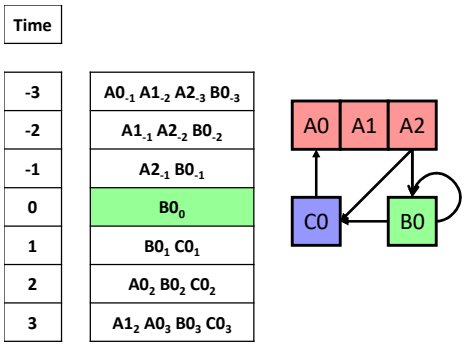

Figure 8.5. Possible active cycles relative to a known mode iteration.

8.2.3 *Nets Traversing Distance Portions of the Iteration Space*

The example in Figure 8.4c moves the writeStm to the rightmost domain. Consider the path for the last green mode iteration starting at time 1. At time 4, this path is transiting the second cycle of the red mode two iterations later than the source. However, for a given green iteration, there are actually four different domain configurations that could be active at that point on the path: A1, A0, C0, or B0, depending on whether the source iteration was the last, 2nd to last, 3rd to last, or that at least 4 more green iterations occurred. Routing in Offset Pipelining often requires us to consider very different positions in mode iteration space. Figure 8.5 introduces a table used for capturing this information. Each box contains the modes and issue slots which could be active relative to a green mode B iteration, with the subscripts denoting iteration distance. Note that the four entries at time 3 in the table correspond to the list above. To route on an Offset Pipelined device, this information must be maintained in order to consider all possible execution sequences.

8.2.4 *Nets with Unknown Flightime*

Consider signal *id*, whose value is read during mode A and is written in mode C, meaning this signal must be "live" during any and all intervening iterations of mode B. However, we would only know the number of B iterations at runtime. Thus, the signal must travel fast enough to get from the read to the write in the case where *count* is zero and mode B never executes, but must maintain the value during any intervening iterations of B. This situation is illustrated in Figure 8.6, with *count* equal to 0 on the left and 2 on the right.

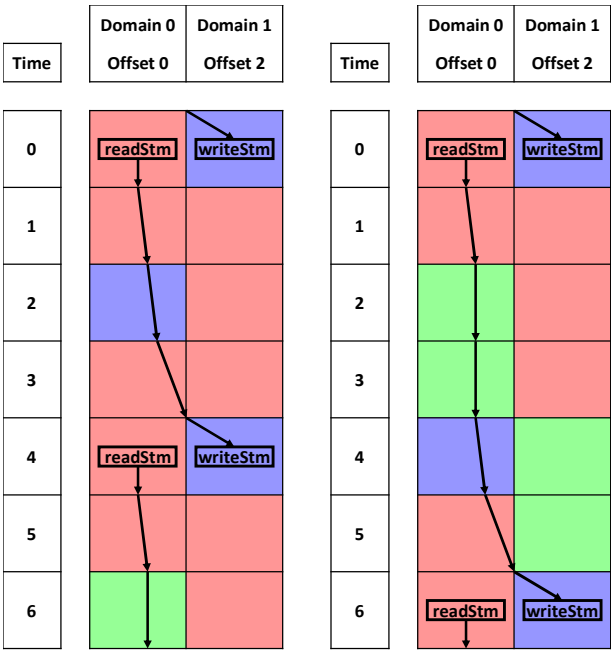

Figure 8.6. Net with a run time defined flight time.

Given the complexities of these scenarios, it is clear that Offset Pipelined routing must manage constraints which have not been studied previously. By careful application of existing techniques and the introduction of new approaches, we have developed a novel, efficient routing algorithm for these systems which is presented in the following sections.

8.3 Signal Router Costs in Different Device Styles

Routers for many styles of reconfigurable devices use the negotiated congestion cost model pioneered by PathFinder, where signal routes allow resource sharing, but the cost of congested resources are gradually increased until the congestion is resolved. This formulation is also at the heart of our EveryTime router. However, the question of the costs of resources and how those costs are incurred during routing requires careful consideration. We will start by reviewing how routing is done on standard devices and then extend this to Offset Pipelined devices.

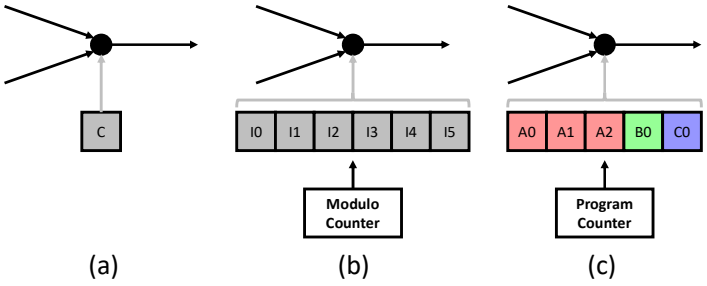

Figure 8.7. Configuration styles for (a) FPGAs, (b) modulo counter CGRAs, and (c) Offset Pipelined CGRAs.

A representation of a 2:1 FPGA routing mux is shown in Figure 8.7a. During routing, there may be two signals that both wish to route through this mux, but because an FPGA is statically configured, only one of the signals can actually use this resource. To deal with this, Pathfinder associates a cost with the use of this mux, and all routes that wish to traverse this mux pay that cost.

Figure 8.7b shows the case for routing resources in a modulo scheduled CGRA, such as those targeted by SPR [FCV+09], which uses an extension of PathFinder. The programming of the mux in this case is actually handled by II different programming bits, each at a different issue slot of the modulo schedule. Multiple signals can share the same mux, as long as they do so during different time slots. Thus, if we are attempting to route signals S0, S1, and S2 through this mux and S0 needs it at time 0, and S1 and S2 at time 1, only S1 and S2 are conflicting and see an added congestion cost. We consider the routes contending for the programming bits of the mux, rather than for the mux itself. As such, the routing costs are maintained for all issue slots of a mux and signal routes only see the costs for time slots on a mux that they are actually using. This method for negotiation is how SPR handles modulo counter pipelined routing.

The Offset Pipelined routing problem is similar to modulo counter routing in SPR: The control of each routing resource is handled by multiple programming bits and signals can share the mux if they use it at different times. Consider the example in Figure 8.8, where we are routing three intra-iteration routes, i.e., signal SA from A2 to A2 between the two domains, signal SB from B0 to B0, and signal SC from C0 to C0. We will focus on a routing mux at the boundary connecting the two domains. For these paths, signal SA traverses the mux at timeslot B0, and signal SC traverses the mux at timeslot A0, and thus do not conflict. However, what

about signal SB? The signal routing must be the same no matter which B iteration we are in, so the successful routing of SB requires use of the mux at both timeslot B0 (for B iterations followed by a subsequent B iteration) and timeslot C0 (for the final B iteration before C). Thus, the routing of signal SB requires the proper setting of the mux in two time slots, and therefore congestion minimization must happen for each of the time slots it uses. For this specific example, signals SA and SB would both see the costs of using the mux at time slot B0, SB also sees the costs of timeslot C0, and SC sees the costs of timeslot A0. Note that in this example there may not actually be a conflict between SA and SB if they both call for the same configuration of the mux in cycle B0. This is similar to the static resource sharing in SPR.

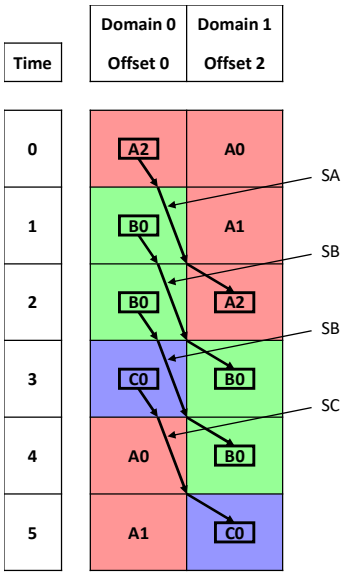

Figure 8.8. Resource costs for routing.

8.4 EVERYTIME ROUTER OVERVIEW

The EveryTime approach provides solutions to the aforementioned challenges faced in routing multi-mode Offset Pipelined systems. At a high level, routing a net using the EveryTime concept creates a single path that consumes all resources across every iteration that could be active at each node along the path. This guarantees that the path is complete for all possible run time mode sequences. While this seems like a costly approach, only one path is actually active at run time for the actual execution sequence. The single path concept implicitly encapsulates any run time behavior.

The proposed solution is based on two observations. The first is that, even with the multi-mode execution style, each signal is generated at a particular time and location and must arrive at the destination time and location regardless of what may happen along the way. The routing cost for a resource is based on the use at a given issue slot. The cost of a route using a resource is the sum of the costs of all issue slots that could be active at that point on the path.

The second observation is that not all nets have a fixed flight time despite the statically scheduled nature of Offset Pipelining. The router must be able to reconcile different paths among possible execution sequences. EveryTime routing takes advantage of register file resources in a new way to synchronize these paths. By breaking these signals into fixed delay paths from source to a register file and from the register file to the sink, the variable delay portion of the path is confined to the register file.

There are several advantages of the EveryTime approach. The core routing is straightforward with no complex multi-path handling. It can handle arbitrary mode transition diagrams and uses negotiated congestion to resolve resource contention. The use of register files to handle variable signal flight time maintains the single path nature of the routing.

By limiting the router to a single physical path, possibly better solutions may be overlooked that involve merging different, independent paths rather than the unified EveryTime path. There may then be a channel width penalty for the EveryTime approach; however, the benefit of avoiding merge and synchronization issues favors a simplified routing approach to multi-mode routing. The Chapter 10 evaluation will also demonstrate that any such overhead is small in practice by comparing EveryTime routing to SPR and a flattened FPGA-like architecture.

8.5 EVERYTIME TABLES

The central issue in routing in an Offset Pipelined system is tracking the active mode and iteration at a given time and place. This section introduces the concept of the EveryTime table that is used to determine which modes and times are active at a given distance from the source and/or sink while routing. These tables are calculated based on the scheduled and placed application, since they are dependent on the mode IIs and domain offset assignments. During routing, these tables are constant and provide reference for the active set of modes and times. EveryTime tables represent the iteration space relative to a particular mode iteration, allowing the router to track the set of possible active resources anywhere and anytime on the device

relative to an anchor point. An anchor point is usually the mode containing the source of the net, though in cases where the signal flight time is not fixed, the sink serves as a second anchor.

8.5.1 *Dealing with Iteration Space*

Routing on a modulo scheduled architecture requires tracking use of physical resources for each time slot in the schedule, effectively unrolling the architecture graph in time to represent the available resources. Adding the dimension of independent modes means that physical resources must be tracked by mode as well as time within each mode. The router must understand how to traverse the possible mode transitions and track the utilization of a resource in multiple modes and times simultaneously.

For modulo scheduling, the next cycle is always known through an increment and modulo operation. In an Offset Pipelined system, moving forward or backward in time relative to a known point can lead to one of multiple possible modes and times in different iterations as illustrated in Figure 8.5. In the most basic case, within a domain, moving forward in time one cycle has two possibilities: either the next cycle is still within the II cycles of the current iteration or the next cycle is in a new iteration. The new iteration can be found through traversal of the mode transition diagram. Consider an iteration of mode B in the example shown in Figure 8.9 on the left. We place the iteration of mode B at time 0 and then construct new entries by examining the mode transition diagram. At time 2, a new iteration of either mode B or mode C begins, as shown in the table. By time 4, there are three possibilities: a second iteration of mode B, an iteration of C following the iteration of B, or the last cycle of an iteration of C that immediately followed the initial mode B iteration that anchors the table. When we move between domains (Figure 8.9 right), we must shift the entries in the table by the difference in offsets between the two domains.

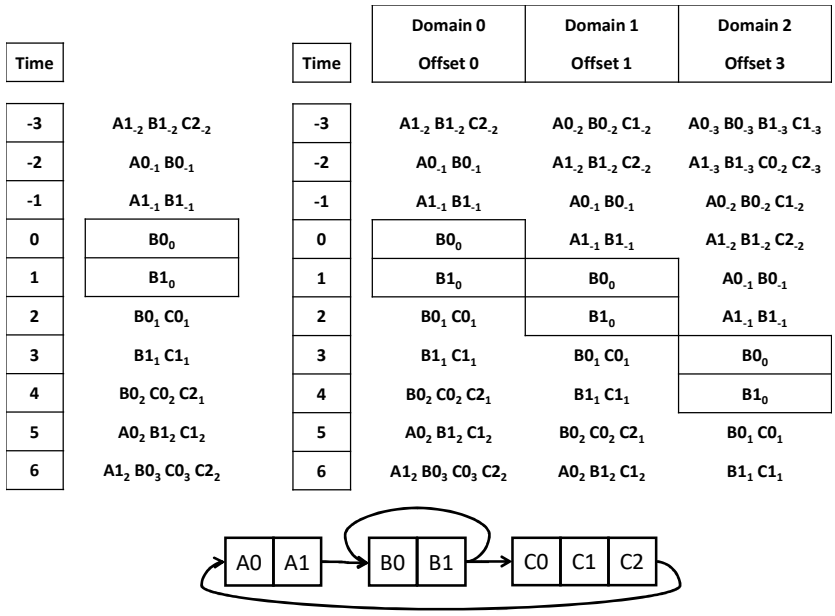

Figure 8.9. EveryTime table for mode B (left). EveryTime tables set to different offsets (right). Mode transition diagram for the EveryTime tables (bottom).

Entries in the table represent possible mode/times that could be active at run time. A letter denotes the mode and a number the cycle of the associated mode iteration from the mode transition diagram. This table captures all possible run time mode execution sequences and provides some intuition about the cost of using a resource for routing a net. Subscripts track the iteration offset relative to the anchor iteration with subscript 0.

We can see that moving further away from the anchor increases the uncertainty of determining which mode and iteration is executing. From a routing cost perspective, moving away from the anchor generally becomes increasingly costly corresponding to this run time uncertainty. The router will try to avoid large sets of active modes and times by preferring cheaper paths through resources with less run time uncertainty, but costly routes are handled seamlessly through the EveryTime formulation.

8.5.2 *Fused Source and Destination Relative Timing*

While many nets in a design will remain within a single iteration, nets also connect different iterations and modes in order to move data. In this case, two EveryTime tables, one associated with the source and the other with the sink can be combined to prune the space of active mode/time combinations required to connect source and sink. This will be explored further in

later sections, but the high level idea is to intersect a source relative table and a sink relative table with the appropriate shift in time to provide the set of mode/times that will complete the path under any runtime scenario. Note that for an intra-iteration net, the EveryTime table for the source and sink is the same, so no further pruning would be possible.

8.5.3 *Reachability*

The EveryTime tables can also be pruned through analysis of modes that can be legally reached from the source enroute to the sink. The basic idea is that the router should not visit resources that cannot be active with the given source and sink pair. This is a more detailed analysis compared to simply applying an EveryTime table to determine the active set.

An example of this situation in Figure 8.9 concerns a variable that is updated in loop B, and the last version of the value is required for an operation in mode C. It is important that the correct value be passed to C, which can be handled during routing by not allowing the path to traverse resources in a subsequent iteration of B. A second example can be found if an application might branch to different modes, an example of which is shown in the Figure 8.10 mode transition diagram. A net with a source in A1 and a sink in the immediately following iteration at B0 would never traverse the alternate iteration of mode C following A. These entries would be pruned from the EveryTime table when routing this net.

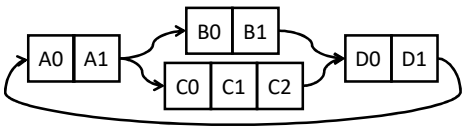

Figure 8.10. Mode transition graph with variable distance between modes A and D.

8.6 LOCKED NETS

We consider nets with a constant flight time to be “time locked,” having a flight time independent of the run time mode execution sequence. The following examples demonstrate EveryTime routing using EveryTime tables to account for the possible execution sequences that may arise at run time.

8.6.1 *Nets with No Iteration Delay*

A net whose source and sink are contained within one iteration is the most basic case. Imagine a net in Figure 8.9 (right) has a source in $B0_0$ of the left domain and a sink in $B1_0$ on the right domain. As the router explores the available resources at a given distance from the source, the modes and times these resources will be active can be found in the table. Ignoring congestion for the moment, in order to use the minimum number of resources to route this net, it is clearly desirable to remain within the active iteration if possible, otherwise the net will exist in multiple iterations at some point along the path.

However, this isn't always possible for two reasons. An intervening offset, such as domain 1 in the example, may have a much larger or smaller offset. This would lead to the table for the domain being shifted up or down such that, in order to traverse that domain, there would necessarily be several active modes and times. The second issue faced in routing is simply congestion; the net may have to find an alternate route, possibly through less desirable resources that have additional mode/times active.

8.6.2 *Iteration Delayed Nets*

An iteration delayed net differs from the intra-iteration net in that the source and sink use different tables anchored by their respective modes. Figure 8.11 shows an example of the source relative table on the left for $B1_0$ on domain 1 and the sink relative table on the right for mode $C0_1$ on domain 2. Note that the sink has a subscript of 1 representing the iteration delay relative to the source. What is unique about an iteration delayed net is that these two tables can be intersected to prune some of the active mode/times for routing. This helps to ignore resources that are not necessary for a given scenario, thereby avoiding overpaying for the path and without repeated calculation of the active set of mode/times.

Figure 8.12 shows the merged tables that prune the space between the source and sink iterations. Note that at times 3, 4 and 5 on domain 1, only one mode is active since we know the sink exists in this iteration. This technique works for all cases where the time of flight is known. Thus, for the mode transition diagram in Figure 8.11 we can handle single iteration delayed nets from A to B, B to A, B to B, B to C, C to A and C to B. Iteration delayed nets from A to A, A to C or C to C cannot be handled using merged tables alone since there is an unknown number of B

iterations leading to variable signal flight times. Choice in execution paths is even supported as long as the flight time is fixed, as seen in Figure 8.13 where mode B or mode C follows A.

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
0	$B0_0$	$A1_{-1}$ $B1_{-1}$	$A1_{-2}$ $B1_{-2}$ $C2_{-2}$
1	$B1_0$	$B0_0$	$A0_{-1}$ $B0_{-1}$
2	$B0_1$ $C0_1$	$B1_0$ (source)	$A1_{-1}$ $B1_{-1}$
3	$B1_1$ $C1_1$	$B0_1$ $C0_1$	$B0_0$
4	$B0_2$ $C0_2$ $C2_1$	$B1_1$ $C1_1$	$B1_0$
5	$A0_2$ $B1_2$ $C1_2$	$B0_2$ $C0_2$ $C2_1$	$B0_1$ $C0_1$
6	$A1_2$ $B0_3$ $C0_3$ $C2_2$	$A0_2$ $B1_2$ $C1_2$	$B1_1$ $C1_1$
7	$A0_3$ $B0_3$ $B1_3$ $C1_3$	$A1_2$ $B0_3$ $C0_3$ $C2_2$	$B0_2$ $C0_2$ $C2_1$

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
0	$B0_0$	$A1_{-1}$ $B1_{-1}$	$A1_{-2}$ $B1_{-2}$ $C2_{-2}$
1	$B1_0$	$B0_0$	$A0_{-1}$ $B0_{-1}$
2	$C0_1$	$B1_0$	$A1_{-1}$ $B1_{-1}$
3	$C1_1$	$C0_1$	$B0_0$
4	$C2_1$	$C1_1$	$B1_0$
5	$A0_2$	$C2_1$	$C0_1$ (sink)
6	$A1_2$	$A0_2$	$C1_1$
7	$B0_3$	$A1_2$	$C2_1$

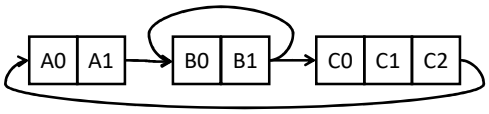

Figure 8.11. Source and sink relative routing tables for a net from mode B to mode C.

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
0	$B0_0$	$A1_{-1}$ $B1_{-1}$	$A1_{-2}$ $B1_{-2}$ $C2_{-2}$
1	$B1_0$	$B0_0$	$A0_{-1}$ $B0_{-1}$
2	$C0_1$	$B1_0$ (source)	$A1_{-1}$ $B1_{-1}$
3	$C1_1$	$C0_1$	$B0_0$
4	$C2_1$	$C1_1$	$B1_0$
5	$A0_2$	$C2_1$	$C0_1$ (sink)
6	$A1_2$	$A0_2$	$C1_1$
7	$B0_3$	$A1_2$	$C2_1$

Figure 8.12. Merged routing table.

8.7 UNLOCKED NETS

The main limitation for EveryTime routing as described so far is that the signal flight time must be known. For conventional pipelined routing on an FPGA, this is always the case. However, for Offset Pipelining, we must also be able handle nets whose flight time isn't known until run time. To solve this problem, the EveryTime router has the net visit a register file along the path. The register file can hold a value as long as necessary before the net proceeds to the sink. The

approach involves tracking net flight time first from the source until a register file is visited and then to the sink after departing the register file. This allows the signal to propagate in a run time dependent way while still being routed statically. The identification of an appropriate register file is handled automatically during routing as part of the EveryTime approach for unlocked nets. The additional support for unlocked nets treats register files as a special type of routing resource in order to eliminate the aspect of the unknown flight time. However, register files are still valid routing resources for other signals. PathFinder negotiation manages access to all routing resources including the register files.

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
-3	$B1_{-2}$ $C1_{-2}$	$B0_{-2}$ $C0_{-2}$	$A0_{-3}$
-2	$D0_{-1}$	$B1_{-2}$ $C1_{-2}$	$A1_{-3}$
-1	$D1_{-1}$	$D0_{-1}$	$B0_{-2}$ $C0_{-2}$
0	$A0_0$ (source)	$D1_{-1}$	$B1_{-2}$ $C1_{-2}$
1	$A1_0$	$A0_0$	$D0_{-1}$
2	$B0_1$ $C0_1$	$A1_0$	$D1_{-1}$
3	$B1_1$ $C1_1$	$B0_1$ $C0_1$	$A0_0$
4	$D0_2$	$B1_1$ $C1_1$	$A1_0$
5	$D1_2$	$D0_2$ (sink)	$B0_1$ $C0_1$
6	$A0_3$	$D1_2$	$B1_1$ $C1_1$
7	$A1_3$	$A0_3$	$D0_2$

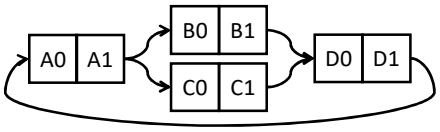

Figure 8.13. EveryTime table for fixed flight time multi-path net.

8.7.1 *Decoupled Source and Destination Relative Timing*

The previous discussion of EveryTime tables for locked nets was based on knowing how far the signal had propagated from the source and therefore how many cycles were left before it would reach the sink. For a net that does not have a fixed flight time, this is not possible. The problem of guaranteeing a register file along the path that can hold the unlocked net value is solved using a series of steps. All routing from the source to the register file is source-relative, meaning that the set of possible executing modes and times is computed relative to the source mode. All routing from the register file to the sink is sink-relative, where we compute the set of possible executing modes and times with an EveryTime table anchored to the sink mode. In this way we essentially convert the time unlocked route into two time locked signals stitched together via a

register file. Note that the register file used is dynamically determined via a phased search concept adapted from PipeRoute [SEH06].

Although the exact time allowed to send the signal from source to sink is unknown, since there are many possible run time execution sequences, we can use the mode transition diagram to find the minimum such delay. Thus, the path must travel from source to register file and register file to sink within the minimum delay. In this way, the communication will complete no matter which mode sequence executes at run time.

Figure 8.14 illustrates the scenario for an unlocked net routing from $A1_0$ in domain 0 to $C0_2$ in domain 1. The left table is anchored to the source while the right is anchored to the sink. In the diagram, the tables are placed relative to each other based upon the shortest flight time, a single B iteration, but a route must support a dynamically determined number of B mode iterations.

The PipeRoute phased search is adapted in the EveryTime router to guarantee a register file waypoint for unlocked nets. Routing begins in the first phase using a source relative EveryTime table. Upon visiting a register file, the search begins a second phase switching to a sink relative EveryTime table to find the destination. The lowest cost route to the destination implicitly selects a register file along the way by requiring a second phase search to discover the sink.

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
-1	$C2_{-1}$	$C1_{-1}$	$B1_{-2}$
0	$A0_0$	$C2_{-1}$	$C0_{-1}$
1	$A1_0$ (source)	$A0_0$	$C1_{-1}$
2	$B0_1$	$A1_0$	$C2_{-1}$
3	$B1_1$	$B0_1$	$A0_0$
4	$B0_2$ $C0_2$	$B1_1$	$A1_0$
5	$B1_2$ $C1_2$	$B0_2$ $C0_2$	$B0_1$
6	$B0_3$ $C0_3$ $C2_2$	$B1_2$ $C1_2$	$B1_1$
7	$A0_3$ $B1_3$ $C1_3$	$B0_3$ $C0_3$ $C2_2$	$B0_2$ $C0_2$
8	$A1_3$ $B0_4$ $C0_4$ $C2_3$	$A0_3$ $B1_3$ $C1_3$	$B1_2$ $C1_2$
9	$A0_4$ $B0_4$ $B1_4$ $C1_4$	$A1_3$ $B0_4$ $C0_4$ $C2_3$	$B0_3$ $C0_3$ $C2_2$

Time	Domain 0 Offset 0	Domain 1 Offset 1	Domain 2 Offset 3
-5	$A1_{-3}$ $B1_{-3}$ $C2_{-3}$	$A0_{-3}$ $B0_{-3}$ $C1_{-3}$	$A0_{-4}$ $B0_{-4}$ $B1_{-4}$ $C1_{-4}$
-4	$A0_{-2}$ $B0_{-2}$	$A1_{-3}$ $B1_{-3}$ $C2_{-3}$	$A1_{-4}$ $B1_{-4}$ $C0_{-3}$ $C2_{-4}$
-3	$A1_{-2}$ $B1_{-2}$	$A0_{-2}$ $B0_{-2}$	$A0_{-3}$ $B0_{-3}$ $C1_{-3}$
-2	$B0_{-1}$	$A1_{-2}$ $B1_{-2}$	$A1_{-3}$ $B1_{-3}$ $C2_{-3}$
-1	$B1_{-1}$	$B0_{-1}$	$A0_{-2}$ $B0_{-2}$
0	$C0_2$	$B1_{-1}$	$A1_{-2}$ $B1_{-2}$
1	$C1_2$	$C0_2$ (sink)	$B0_{-1}$
2	$C2_2$	$C1_2$	$B1_{-1}$
3	$A0_3$	$C2_2$	$C0_2$
4	$A1_3$	$A0_3$	$C1_2$
5	$B0_4$	$A1_3$	$C2_2$

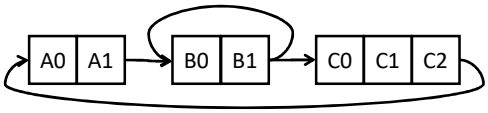

Figure 8.14. Unlocked net routed with a register file.

8.7.2 *Architecture Considerations*

While the concept of a net passing through a register file waypoint is straightforward, the net is nevertheless being routed using the EveryTime concept. This means that the register file might be visited when multiple modes and times are active relative to the source. However, only one sequence actually occurs at run time. In order to ensure that the correct value is written to the register file, the architecture must include a valid bit with the data to enable writing to the register file. The enable signal is therefore asserted in a runtime dependent way when the desired value should be written. Figure 8.15 revisits our earlier example from Figure 8.3 showing two different execution traces. On the left, no green iterations execute between red and blue while two execute on the right. For this example, we assume the register file has write through so the value is both written and read at time 2 for the left example. With the possibility of either a blue iteration or a green iteration executing at time 2 relative to the source, the valid bit provides the mechanism to ensure the correct value is written to the register file. In the example on the right, the valid bit is set at time 2, but not at times 3 or 4. Only register file writes need to be protected this way since the write is a stateful operation while a speculative read operation is not.

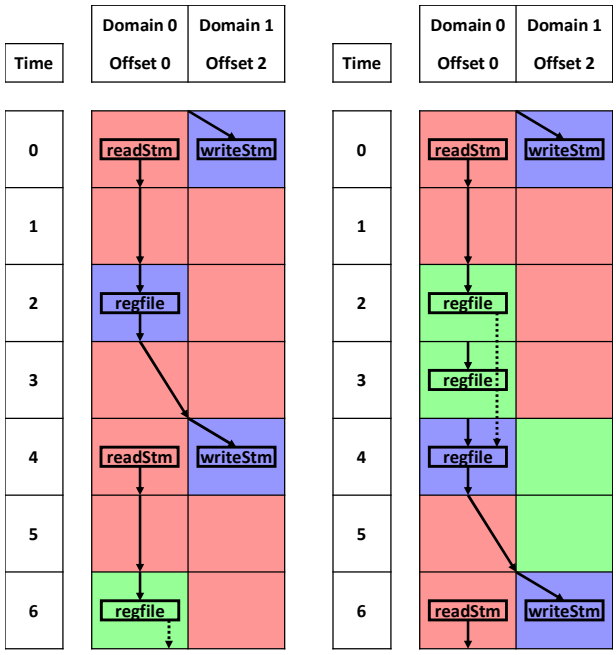

Figure 8.15. Example demonstrating the valid bit write enable for register files.

8.8 EVERYTIME ROUTER

The EveryTime router is based on QuickRoute for pipelined routing. Rather than operating directly on an architecture graph, the routing process is augmented with EveryTime tables as previously described. For unlocked nets, the EveryTime router also ensures that a register file waypoint is included on the path. This is accomplished by changing from a source relative to sink relative search when a register file is visited and then requiring that the sink be visited at the appropriate latency during the sink relative phase to successfully route the net.

The PathFinder cost metrics rely purely on the available time slots provided by each mode. This is akin to the SPR unrolled datapath graph that represents the time slots available for each physical resource in the device. These data structures provide convenient accounting of the PathFinder metrics, but are not used directly for routing. They are instead populated based on the EveryTime table data for a given path. From a cost perspective, a path must pay for the use of all the mode/times that could be active along the path. For example, if a resource is used where six different mode/times are possible, the cost of this resource is the sum of the costs of each of the six mode/time possibilities. This will encourage routes to use paths that are less uncertain, but allow paths to use whatever resources are necessary to achieve the required signal connectivity.

The EveryTime tables are pre-calculated based on mode IIs and assigned offsets. Preparing to route a given net involves aligning and merging tables if the net is locked. In order to expand a node in the architecture during routing, the domains of the wires in question are used to index into the EveryTime table to determine the active mode/times at the given distance from the source or sink. The tables allow movement among domains and through time relative to the source and/or sink. There is no need to traverse the mode transition diagram to calculate when the signal exists.

8.9 Resolving Congestion

PathFinder provides the mechanism to resolve congestion. A given net consumes whatever mode/times are part of the active set for each node in the path. PathFinder evaluates the mode/time occupancy information to address present and history sharing costs.

While conventional routing algorithms support nets with a single source but multiple sinks, offset pipelined netlists also involve nets with multiple sources in certain situations. For example, in Figure 8.4 left, the signal is initialized in one mode and updated in another and, therefore, has two possible sources. This is reasonable since the dynamic execution pattern will determine which source actually generates a given signal at run time.

Our EveryTime router handles this by decomposing all nets into two-terminal source-sink pairs, which are routed independently. However, we must now resolve the merging of the two sources: Once an iteration of a loop begins, the two sources of the loop index must enter this loop body mapping at the same point. We use PathFinder to negotiate this shared join point by tracking the configuration of muxes in the architecture. The separate source-sink routes of the signal are routed independently and can share resources between the paths freely since they represent the same signal, but an incompatible mux configuration between the two paths is penalized. Thus, if the two routes join at the entrance point to the loop body, there is no penalty, but any other join is penalized and negotiated by PathFinder. Our EveryTime router creates an implicit phi node to join the paths, created as a side-effect of which mode precedes the loop body iteration in the run time execution. Formulating multiple source merging and the EveryTime concept as PathFinder negotiation provides global routing based on proven techniques

8.10 Routing Constants

Routing constants calls for another type of router. In this case, a sink is known but no source is assigned. Here we perform a reverse search from the sink using EveryTime expansion following the same cost metrics as QuickRoute to find an available register file read port. PathFinder's present and history sharing values also apply to these paths and ensure they are negotiated on the same footing as all other nets. In practice, register file ports are plentiful across the benchmark set and are usually found in the same domain as the sink. Even in highly utilized scenarios, constant routing is unlikely to become a major source of conflict because unlike the majority of

nets, including unlocked net, constants do not have a specific flight time that must be met. This flexibility means they can use less contested resources and take circuitous paths to an available register file port.

8.11 Feedback to Scheduling or Placement

The prototype tool chain does not include feedback to scheduling and placement from the router. In order to evaluate routing performance, the architecture channel width is swept to determine the minimum channel width necessary to route designs. This stresses the performance of the router in order to focus on evaluation rather than meeting constraints of a specific target architecture.

For use with a fixed architecture, a practical tool chain would include feedback to the scheduler and placer to loosen constraints in these phases to provide enough flexibility to complete routing. One possibility would be to annotate nets that remain in a conflicted state after a certain number of PathFinder iterations. These nets could be assigned additional slack in scheduling to stretch out the overall schedule to make it easier for placement and routing to find a solution. A second alternative would include analysis of channel utilization which the placer could use to search for a better resource arrangement in advance of routing. These metrics might be used to recognize possible congestion in advance of routing to guide scheduling and placement in order to map applications to specific architectures with fixed routing resources. This type of feedback to the scheduler is akin to the feedback from placement discussed in 6.3 and gives the tool chain the opportunity to continue searching for a successful mapping rather than failing.

8.12 Evaluation

The placement and routing phases of the tool chain are evaluated in two ways. The first compares against a hypothetical flattened architecture while the second comparison is made to an SPR implementation for modulo scheduled CGRAs. The target architecture is based on Mosaic, work that explores resource composition for modulo scheduled CGRAs [VE10].

The first architecture is “flattened” to provide a likely unachievable theoretical lower bound modeling the best possible implementation that might be attained by the Offset Pipelined placer

and router. Our goal is to apply existing techniques to this flattened architecture and measure the relative algorithm efficiency via the resulting channel widths, a standard approach for router evaluation in the FPGA literature. Channel width measures the amount of inter-domain interconnect resources. By sweeping the channel width, the router can be stressed to determine the minimum channel width required for a given application. We transform the Offset Pipelined placement and routing problem into a more standard pipelined FPGA routing problem that will have similar or relaxed constraints. We remove mode transitions and instead have only a single configuration where every domain has logic resources equal to the Offset Pipelined resources multiplied by the total schedule length. Thus, if in the Offset Pipelined case we have two modes with IIs of 2 and 3, the flattened architecture has 5 times as many logic resources per domain than the Offset Pipelined device. Signals are pipelined so that if the minimum flight time in the schedule is N, the signal must go through exactly N registers in the flattened architecture. In this way, the two architectures have the same scheduling, placement and essentially the same routing constraints, but the additional complexity of mode transitions and issue slot windows have been eliminated. The router for the flattened architecture is a QuickRoute implementation for pipelined routing.

The flattened architecture comparison normalizes channel width to the best result we could expect if signals were evenly distributed among the cycles of the Offset Pipelined version. The channel width attained by a flattened implementation is divided by the total schedule length of the corresponding Offset Pipelined implementation and rounded up to produce this lower bound.

The channel width results in Figure 8.16 show that the Offset Pipelined tool chain achieves mappings with channel widths of approximately a 10% overhead compared to a flattened architecture using QuickRoute, demonstrating that our algorithm is quite close in efficiency to the existing router even though it must deal with a more complex problem. Raw channel widths are presented in Figure 8.17. Note that the overhead corresponds to an average of about 0.6 of a channel across our benchmark suite for the Offset Pipelined device, which is a minor penalty.

In many cases, the channel width overhead using EveryTime routing is a single channel. For a scenario where the EveryTime router uses 5 channels compared to 4 for the flattened implementation the overhead would be 25%. With an average of 10% across the benchmarks, we find that the EveryTime router does not require an unreasonable amount of routing resources to support the Offset Pipelined execution model.

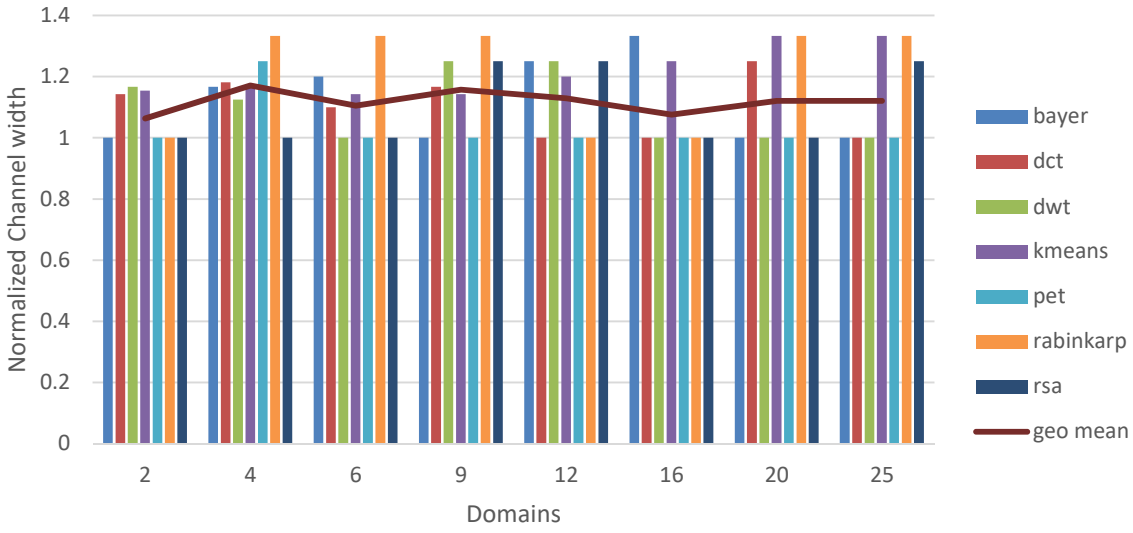

Figure 8.16. Channel widths for EveryTime router normalized to flattened architecture.

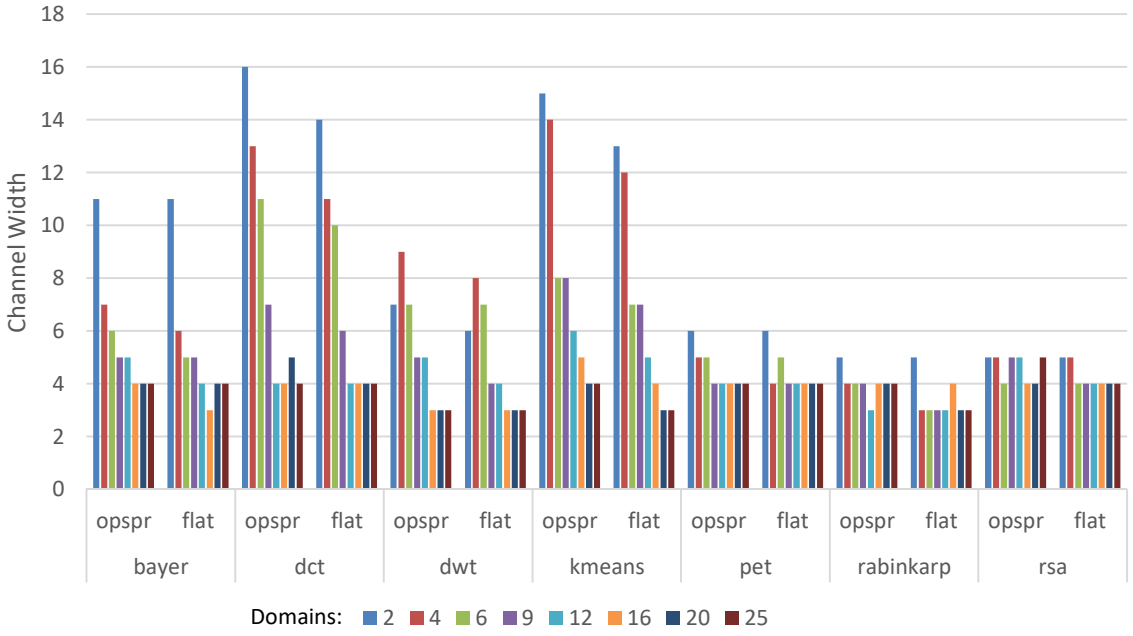

Figure 8.17. Absolute channel widths for EveryTime router compared to flattened baseline.

Our second comparison is to a basline SPR implementation. While the FPGA-like baseline is a useful tool to evaluate the channel width requirements of the EveryTime router, SPR is a more closely related CGRA tool taking advantage of modulo scheduling and time multiplexed resources. Results for this second comparison are shown in Figure 8.18 comparing Offset

Pipelining to SPR. In SPR we are restricted to the single modulo counter based implementation of existing systems, while the EveryTime router makes use the Offset Pipelining execution style.

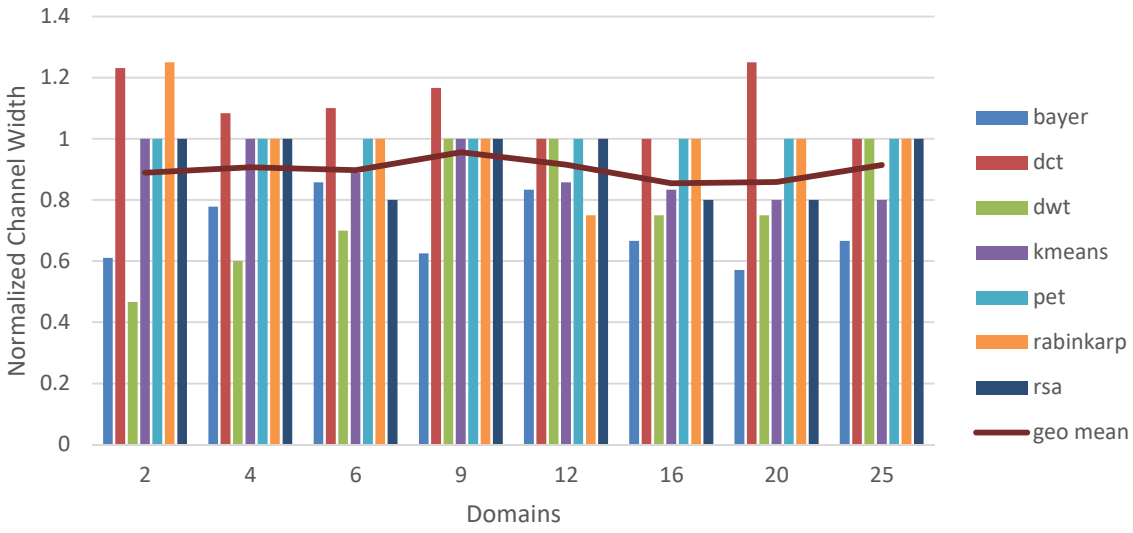

Figure 8.18. Channel width for EveryTime router compared to SPR.

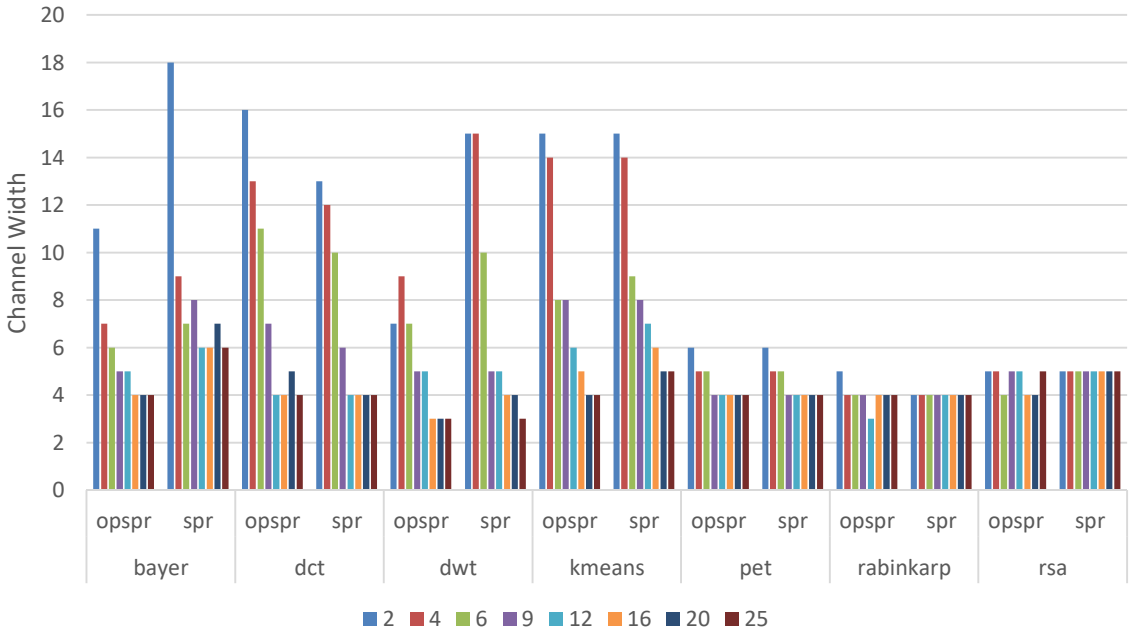

Figure 8.19. Absolute channel widths for EveryTime router compared to baseline SPR.

As seen in the graph, the channel width requirements are heavily influenced by the application. For applications like the discrete wavelet transform and bayer filtering with many modes, the EveryTime router requires fewer channels by allowing resources to be more effectively shared in time. The DCT on the other hand with only two modes uses slightly more channels. Overall, EveryTime routing of the Offset Pipelined implementations uses about 0.9x the channels of SPR. The raw channel width values are presented in Figure 8.19.

The results presented here validate the practicality of EveryTime routing for Offset Pipelined CGRAs. Channel widths remain in a realm comparable to existing techniques while enabling the benefit of the modal execution model.

Chapter 9. SPR AND PREDICATE AWARE SPR

One body of previous work warrants special attention as the progenitor of Offset Pipelining and is introduced in this chapter as background. The broader scope of related work will be covered in Chapter 11. Offset Pipelining is a continuation of work on CGRA compilation techniques initially unified in the Schedule, Place, and Route (SPR) tool. The SPR scheduler is based on IMS, the placer on simulated annealing, and the router on QuickRoute. SPR was further refined [Fri11] to take advantage of predicate-aware mutual exclusion, further improving its performance.

Managing control flow of an application on modulo scheduled CGRA architectures requires a mechanism for converting control dependencies into data dependencies to facilitate software pipelining. As noted in Chapter 2 motivating the development of Offset Pipelining, this requires all control paths simultaneously consuming resources in the mapped application. A second inefficiency is that in low II applications, extra configuration memory in the device is idle. The baseline SPR implementation was enhanced to apply this extra memory to address the resource consumption of more complex control flow in CGRAs.

This chapter begins by providing an overview of predicate aware sharing. This is followed by a discussion of the baseline SPR implementation and its integration into the Offset Pipelining tool flow for evaluation purposes. While an implementation was not available for direct comparison, predicate aware SPR (PA-SPR) is then introduced with attention to how its potential performance will be compared to that of Offset Pipelining in Chapter 10.

9.1 PREDICATE AWARE SHARING

A conventional processor changes the instruction execution sequence in order to handle conditional execution. The example in Figure 9.1 might execute only the sum or difference depending on the value of *diff* for each iteration of the loop by using conditional branch operations. Controlling the program counter in this manner allows a processor to execute only the useful instructions for a given iteration. This is feasible and often preferred on a processor because instructions are small. On the other hand, a single CGRA instruction is very large in comparison making it difficult to change instructions in a data dependent manner at run time.

```
for (i = 0; i < n; i++)
{
   diff = a[i] – b[i];
   if (diff > 0)
      sum = sum + diff;
   else
      sum = sum – diff;
}
```

Figure 9.1. Example to motivate predicate aware sharing.

Control flow changes for a statically scheduled CGRA can instead be represented using predication. The predicated dataflow graph for the Figure 9.1 example is illustrated on the left in Figure 9.2. In this case, both the sum and difference are computed but the comparison operation controls which result is retained by selection via a *phi* node.

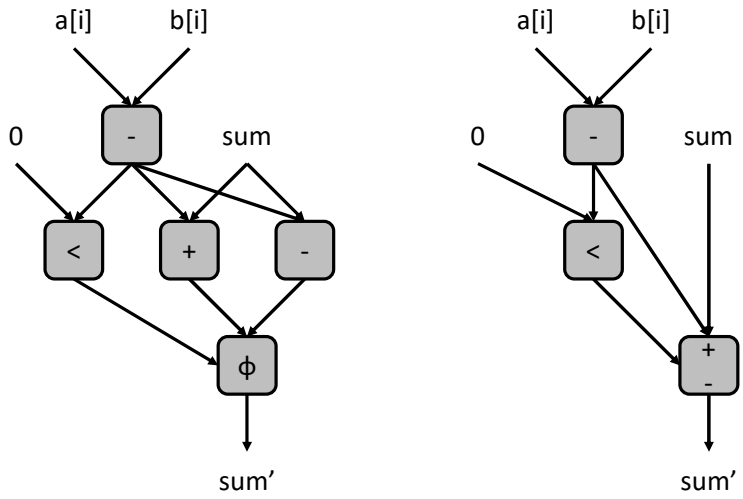

Figure 9.2. Predication (left) and predicate aware sharing (right) of Figure 9.1.

It is clear in the example that only one of the conditional paths will execute on a given iteration. Predicate aware sharing recognizes this property and allows these operations to coexist on the same hardware resource and selects which instruction will execute at run time, illustrated in Figure 9.2 on the right. This is distinct from the conventional processor approach since the schedule issues the same instruction address in either case. The selection of the operation is controlled by the result of the comparison in a data dependent fashion at run time.

A clear benefit is gained by reducing the size of the dataflow graph, evident when comparing the predicated and predicate aware sharing versions in Figure 9.2. Issuing fewer operations reduces pressure in the routing phase. In this case, the recurrence limited II, defined by the recurrence loop from sum' back to sum, is also reduced by eliminating the explicit *phi* node from the predicated approach. Even if the mux operation can be combined with either the add or subtract in the functional unit of the target device, the other operation would need to execute in the prior cycle, requiring an II of 2 for the example.

The benefits of predicate aware sharing come at the cost of additional hardware support. An operation is selected through a combination of the modulo schedule and run time predicate bits. A diagram of modulo counter configuration selection is shown in Figure 9.3. The counter directly controls the address of the configuration retrieved on each cycle of the schedule. There are a variety of ways to insert predicate bits into the configuration process. Figure 9.4 illustrates the approach taken by PA-SPR to augment the modulo counter, effectively changing the address based on predicate bits for configuration lookup.

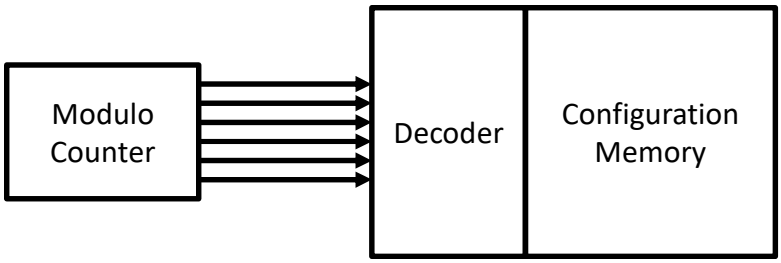

Figure 9.3. Modulo counter controlled configuration memory.

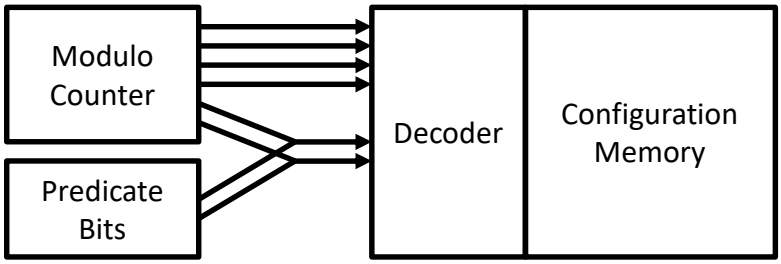

Figure 9.4. Predicate bits combined with modulo counter controlled configuration memory.

A closer examination of Figure 9.4 illustrates some of the tradeoffs with predicate aware sharing in SPR. If the II of the application is large, more bits of the modulo counter are needed

to represent the full schedule. In this case, few or no bits may be available for predicate aware sharing. On the other hand, a short schedule may allow many predicate bits to influence configuration selection. Code with greater control flow complexity and a larger II may limit the effectiveness of predicate aware sharing because the number of predicate bits and therefore the opportunity for shared issue slots is limited.

9.2 BASELINE SPR

The baseline SPR implementation maps dataflow graphs in a modulo scheduled fashion to the target CGRA, as noted previously. This is an important comparison point because the Offset Pipelining infrastructure includes an SPR implementation capable of running the benchmarks presented in this work. While the applications for SPR and Offset Pipelining differ in their organization due to the decomposition into modes for the latter, they were written to be comparable, performing exactly the same computation in both versions. This allows direct and fair comparison between SPR and the Offset Pipelining flow.

9.3 UPPER BOUND FOR PA-SPR PERFORMANCE

The complete predicate-aware SPR system is a complex combination of tight integration with a front-end compiler and augmented scheduling, placement, and routing. Although it has some similarities to Offset Pipelining, the details of the algorithms are different, and the resulting mappings have different tradeoffs. Although it is important to compare Offset Pipelining to PA-SPR, we do not have a complete PA-SPR system available for this testing. Instead, we develop a bound on possible PA-SPR performance that we can use for comparison purposes.

Recall from our previous discussion of scheduling in PA-SPR that the resulting mappings have a single II, which must be large enough to have an issue slot for all instructions in the application. Mutually exclusive operations, those that can never be active simultaneously at run time, can be assigned to the same issue slot. In a multi-mode application, operations from different modes are mutually exclusive, and thus can share issue slots. Therefore the best case for a PA-SPR scheduling is to fit the design into the footprint of the worst-case mode, the mode that requires the largest II when scheduled independently. Ignoring inter-mode constrains, if each mode M independently requires an II of II_M either because of resource constraints or intra-

mode recurrence loops, then the overall application cannot have an II smaller than the largest such mode II. Inter-mode constraints and scheduler inefficiencies may make the system require a larger II, but it cannot be smaller.

For our PA-SPR comparison, we use the largest per-mode II among all of the modes in the application to calculate performance. The limiting II can be computed with our toolchain. We assume that this ideal schedule II survives placement and routing despite these steps generally increasing the II. This provides an optimistic estimate for PA-SPR results.

While we use ideal inter-mode sharing to calculate PA-SPR performance, predicate aware sharing is not limited to these modes. Intra-mode sharing, such as in Figure 9.1, is possible as well. However, across the set of benchmarks used in this work, we find that the worst case modes are recurrence limited such that further sharing will be unable to reduce the II used to calculate PA-SPR performance. We therefore have a reasonable, even optimistic, upper bound for PA-SPR performance used to evaluate Offset Pipelining.

Given this optimistic baseline, where might Offset Pipelining still perform better compared to PA-SPR? Compared to PA-SPR, Offset Pipelining satisfies inter-mode constraints, and will have some increases to the achieved IIs due to the requirements of full placement and routing. However, while Offset Pipelining will pay the per-mode II for each iteration, PA-SPR is forced to pay the worst case II for each iteration. Locked to this II is an inherent inefficiency of the PA-SPR computation model. Thus, for the baseline at least, PA-SPR may appear more favorable when all modes are roughly the same size, or inter-mode constraints significantly impact the overall mapping quality, while OPS will benefit when modes are less balanced, especially when the most frequently executed mode is not the most costly in terms of recurrence or resource requirements. We expect Offset Pipelining to perform better for benchmarks that exhibit multi-mode behavior.

Chapter 10. TOOL CHAIN EVALUATION

This chapter evaluates the complete Offset Pipelining tool chain as compared to SPR. Results presented in Chapter 5 demonstrate the potential benefits of the scheduling approach compared to IMS, the scheduling phase of baseline SPR. Chapter 8 established that the EveryTime router generates solutions within reasonable bounds in terms of available routing resources on a target device. The following sections cover the benchmark applications in more detail and then assess the performance of the complete Offset Pipelining tool chain. We first perform a direct comparison with a full baseline SPR implementation for single mode applications and then move on to multi-mode application mapping compared to SPR and the upper bound on PA-SPR performance described in 9.3.

10.1 BENCHMARKS

The benchmark applications provide a set of signal processing algorithms, typical candidates for CGRA mapping. While the Mosaic project [CFV+07] included the Macah [YCF+08, Ylv10] front end compiler for generating dataflow graphs for the SPR tool chain, there was insufficient support available to develop or modify a comparable infrastructure for Offset Pipelining. Instead, applications are written in C in a style that facilitates conversion to dataflow graphs that can be consumed by the tools. An example of this style is shown in Figure 10.1 making the single assignment nature of the code apparent. The applications were written directly in this style but were compared against reference implementations to ensure correct behavior.

The netlist conversion tool enforces that each variable be written only once and be defined before use. Without a compiler comparable to Macah, this form allows straightforward conversion to a dataflow graph. Each line of code becomes an operation and variables represent the net connections in the resulting netlist. Since a static single assignment (SSA) form is commonly used in compiler intermediate forms, it would be feasible to generate dataflow graphs through a compiler front end such as LLVM [LA17].

A loop variable is represented as a pair of values in the code, one for the current iteration and one for the next. This is illustrated in Figure 10.2 with *s0* and *s0_loop*. The current value for a given iteration is *s0_loop* and is used to calculate the new value of *s0*. At the end of the iteration, *s0_loop* is updated with the new value for the subsequent iteration.

```
B2 = B1 & Const7;
C1 = A_loop >> Const6;
C2 = C1 & Const7;
D1 = A_loop & Const512;
row = D1 != Const512;
sreset = A2 == Const0;
writeval = A2 == Const7;
rowsreset = (sreset && row);
rowwrite = (writeval && row);
z0 = C2 * Const8;
z2 = A2 + z0;
z4 = B2 + z0;
ReadMemory(z9, z2, 0);
```

Figure 10.1. C code example of static single assignment form.

```
z5 = z3 - z4;
z6 = z5 + s0_loop;
s0 = rowsreset ? z5 : z6;
...
s0_loop = s0;
```

Figure 10.2. Representing loop carried nets.

10.1.1 *Dataflow Graph Conversion to XDL*

The dataflow graphs are stored as XDL files using the Torc [SWS+11] infrastructure. Torc provides a comprehensive API for this file format and is used to represent the application through all phases of the mapping. Scheduling adds domain offset and operation time slot information. Placement assigns physical positions to domain offsets and operations. Routing completes the implementation by recording the necessary interconnect configuration in the nets. An example of a portion of an XDL netlist is shown in Figure 10.3 to highlight its structure. The *design* element contains information about the applications including the number of modes, execution counts and the mode transition graph for EveryTime table construction. The *inst* elements represent the dataflow operations corresponding to statements from the original C code. Lastly, *net* elements contain the connections between operations and also hold the extra delay annotations for feedback between stages.

XDL netlists support multiple source and sink terminals (outpins and inpins, respectively) but this is not used for the Offset Pipelining tool chain. Instead, nets are decomposed into two terminal pairs and tagged with indices that informs the tools to treat nets with a common index as a single logical net. During placement, individual sinks must be annotated, but XDL only supports configurations at the net granularity, so a net is needed for each sink. This is merely an artifact of using XDL as the netlist container. The index ensures that logical nets with multiple sources and/or sinks can share routing resources, otherwise PathFinder would interpret a reuse of resources as a conflict that must be resolved. Without the need to annotate an individual sink, we would use a conventional representation as shown in Figure 10.3 where the nets represent the complete net connectivity.

```
design "dct_singlecounter_2mode_pipejump_netlist" mosaic2-1 v1.0, cfg "MODE_COUNT::2
  _ExecutionCount:0:512 _ExecutionCount:1:512 _ModeGraphNode:0:0 _ModeGraphNode:1:1
  _ModeGraphTransition:0:1 _ModeGraphTransition:0:0 _ModeGraphTransition:1:0
  _ModeGraphTransition:1:1";
...
inst "C2_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::BITAND";
inst "D1_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::BITAND";
inst "row_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::NOTEQUAL";
inst "col_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::EQUAL";
...
net "C2",
            outpin "C2_op" O,
            inpin "z0_op" A,
            inpin "x0_op" A,
            cfg "WIDTH::32"
            ;
net "row",
            outpin "row_op" P,
            inpin "rowsreset_lop" B,
            inpin "rowwrite_lop" B,
            inpin "z10_stmread" R,
            cfg "WIDTH::1"
            ;
...
```

Figure 10.3. XDL netlist example prior to scheduling.

The XDL netlist is modified as it passed through each phase of the tool chain. Figure 10.4 shows these changes following scheduling. The *design* element now includes the target device as well as the domain offsets and IIs assigned during scheduling. Operations are assigned time slots and the initial nets are decomposed into two terminal pairs.

```
design "dct_singlecounter_2mode_pipejump_netlist" mosaic2-1 v1.0, cfg "
  DEVICE::m2_dp32c32_2x2 MODE_COUNT::2 OFFSETS::0,2,9,16, _ExecutionCount:0:512
  _ExecutionCount:1:512 _II:0:5 _II:1:7 _ModeGraphNode:0:0 _ModeGraphNode:1:1
  _ModeGraphTransition:0:1 _ModeGraphTransition:0:0 _ModeGraphTransition:1:0
  _ModeGraphTransition:1:1";
...
inst "C2_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::BITAND TIMESLOT::3";
inst "D1_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::BITAND TIMESLOT::4";
inst "row_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::NOTEQUAL TIMESLOT::2";
inst "col_op" "MOSAIC_ALU_SITE", unplaced, cfg "MODE::0 OP::EQUAL TIMESLOT::2";
...
net "C2_0",
            outpin "C2_op" O,
            inpin "z0_op" A,
            cfg "WIDTH::32 INDEX::6"
            ;
net "C2_1",
            output "C2_op" O,
            inpin "x0_op" A,
            cfg "WIDTH::32 INDEX::6"
            ;
...
```

Figure 10.4. XDL netlist example after scheduling.

10.1.2 *Applications*

This section provides details of the various benchmark implementations. A summary of the applications is shown in Table 10.12 with a brief description of each. In order to provide a comparison to existing work, each application is written in two forms. The first is a monolithic version relying on predication for use with SPR to provide baseline performance. The second version takes advantage of the Offset Pipelined execution model to avoid executing unnecessary operations and limits the amount of predication. The table includes the total number of operations in the single and multi-mode versions of the applications, along with the mode count for the multi-mode Offset Pipelined version.

Note that the Bayer, DCT, and DWT applications have deterministic loop bounds throughout and, therefore, do not depend on the actual data sets. The other benchmarks are data dependent. For example, the sample data for K-means converges in three iterations and the PET dataset contains events every 25 samples on average.

Table 10.12. Offset Pipelining Benchmark Applications

Application	Description	Single mode OPs	Multi-mode OPs	Mode Count
Bayer	Bayer filtering, including threshold and black level adjustment	161	139	4
DCT	8x8 discrete cosine transform	96	103	2
DWT	Jpeg2000 discrete wavelet transform	85	102	8
K-means	K-means clustering	121	142	4
PET	Positron emission tomography event detection and normalization	77	78	2
Rabin Karp	String matching	101	115	3
RSA	Encryption and decryption	116	128	4

In order to compare performance among the different implementations, the number of cycles needed to execute test cases is recorded and normalized against the recurrence limited cycle count of an IMS scheduling. This provides insight into the performance of OPS relative to IMS and further allows the applications to be compared to one another.

10.1.2.1 Bayer

This Bayer filter application performs a black level adjustment, edge padding, and the bayer demosaicing, as found in digital camera processing pipelines. The Offset Pipelined version is broken into four modes; the first is the black level adjustment, the second and third handle padding the image, and the fourth performs demosaicing.

10.1.2.2 Discrete Cosine Transform

An 8x8 DCT implementation is logically broken into two modes, one pass over the rows of the image, the other over the columns. This implementation is built using a single counter with bitwise operations used to pick the appropriate values for indexing into coefficient and temporary memory locations. The coefficients are not calculated in the dataflow graph; instead, they are assumed to be pre-calculated and available in a memory block when the application is running. It is a fixed point implementation.

10.1.2.3 Discrete Wavelet Transform

The wavelet transform comes from Jpeg2000 and implements a 9/7 forward transform. Like the DCT, it is a fixed point solution. The algorithm goes through two phases of prediction and update before scaling the result and packing the results for output in a total of eight modes. This sequence of loops makes this implementation a great candidate for OPS since only one loop is

active at a time and both intra and inter-loop iterations can be aggressively pipelined and interleaved.

10.1.2.4 K-means

This application was written to cluster into eight groups with three data channels. By far the most complex application with multiple nested loop bodies for multiple channels, centroid weighting, flags and temporary memory use, it nevertheless has a clear top level decomposition into cluster assignment and centroid update modes.

10.1.2.5 Positron Emission Tomography

The PET application [HDL+09] would be used to detect and assign high resolution time information to scintillator crystal events in a medical scanning system. It is broken into two phases. The threshold phase determines if the sensor has detected the beginning of an event. The normalization phase then computes the detailed arrival time and total energy. This is the smallest application and rapidly hits its recurrence limit.

10.1.2.6 Rabin-Karp

The Rabin-Karp algorithm is used for string matching. It is organized into three modes, one for setup, one for the main processing loop, and a third for a fast inner loop. Including a large conditional block in the main loop, this application is notably different from others such as the DCT which has no conditionals.

10.1.2.7 RSA

The final application is an RSA encryption and decryption block with a 32-bit key. The two modes correspond to the encryption and decryption phases selectable at run time. Like the DCT, this application has two modes that are close to the same size, which lends itself to efficient sharing. However, unlike the DCT, the RSA application has significantly more single bit control logic.

10.2 SINGLE MODE APPLICATIONS IN OPS COMPARED TO SPR

Our first step in evaluating Offset Pipelining is to examine its performance for single mode applications against baseline SPR performance. By taking each application through both the

Offset Pipelining and SPR tool flows, we can determine the degree to which the added scheduling constraints of Offset Pipelining impact performance without the primary advantage of multiple modes.

For a single mode application, only the scheduling phase is unique. The placement and routing portions for single mode applications are indistinguishable from SPR. This is noteworthy for routing since, without the complexity of multiple modes, the EveryTime router is identical to QuickRoute on which it is based.

Figure 10.5 compares Offset Pipelined Scheduling (OPS) and IMS without subsequent placement and routing using a geometric mean of normalized throughout across all seven benchmarks. Performance for each application is normalized to the ideal recurrence limited throughput of the monolithic version. The *OPS Single Mode* data is the OPS algorithm applied to the same monolithic dataflow graph as IMS while the *OPS* data targets the multi-mode versions of the application. We see that OPS applied to a single mode application is less effective compared to IMS. Performance for IMS is 1.85x better than Offset Pipelining and more than 2.2x better on 6 or fewer domains. This deficit decreases to less than 1.2x at 16 domains or more.

Consider the added issue slot window constraint in Offset Pipelining, described in section 5.1. When there are few resources, this becomes a limiting factor since the domain offsets dictate where issue slots will be. In order to meet the needs of the application, Offset Pipelining may need to maintain a higher II to provide sufficient issue slots with only a few domains available to adjust. For applications with a large latency and few resources, the OPS algorithm must rely on the II to meet the issue slot needs of the dataflow graph. When applications support multiple modes, Offset Pipelining can share resources among the modes and is able to overcome the issue slot limitation somewhat in extremely resource poor situations. Figure 10.6 and Figure 10.7 provide the II data for the scheduling results.

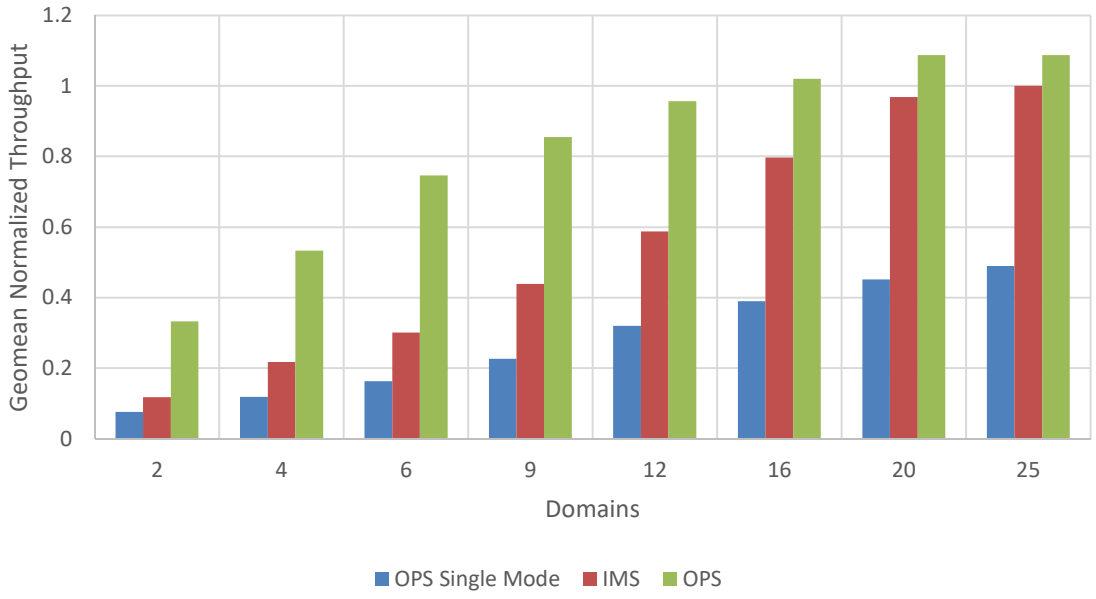

Figure 10.5. OPS to single mode applications.

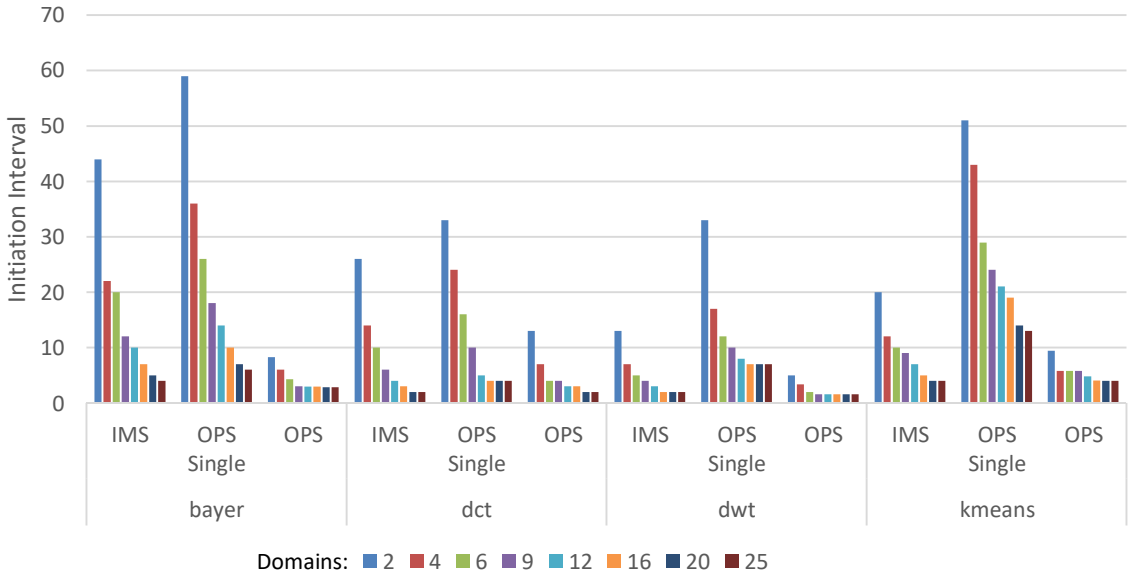

Figure 10.6. Scheduling II data for Bayer, DCT, DWT, and K-means benchmarks.

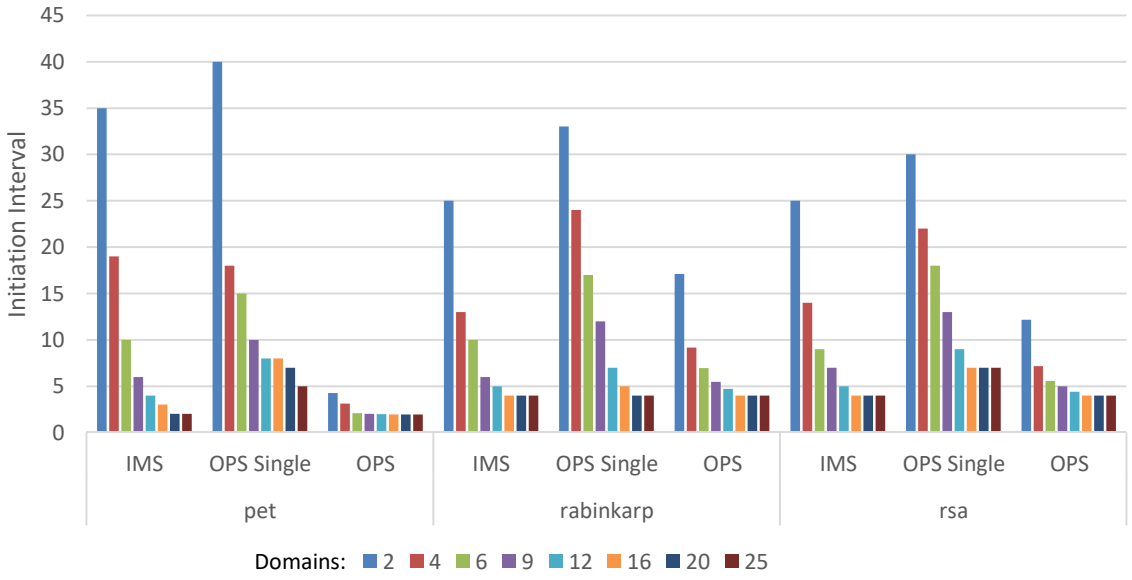

Figure 10.7. Scheduling II data for PET, RabinKarp, and RSA benchmarks.

10.3 General performance vs SPR and PA-SPR

Focusing on the strengths of Offset Pipelining, we consider the multi-mode applications using the Offset Pipelining tool flow including placement and routing, referred to as OPSPR in the figures. Figure 10.8 compares Offset Pipelining to the SPR baseline, which are complete implementation flows through routing, as well to the PA-SPR performance bound discussed in section 9.3 across the benchmark set. While PA-SPR takes advantage of resource sharing similarly to Offset Pipelining, it cannot take advantage of independent IIs. This added flexibility gives Offset Pipelining the edge for more mode oriented applications.

In addition to the independent IIs, the implied phi nodes at mode transitions help eliminate some operations from the dataflow graphs and allow for more compact placement. When these nodes are on recurrence loops, Offset Pipelining may lower the effective II of the application. Multiple modes and implied phi nodes facilitate the improved performance we see over the SPR and PA-SPR solutions.

Comparing Offset Pipelining to the upper bound on PA-SPR, Offset Pipelining provides an average 1.2x better geometric mean performance. The PA-SPR upper bound does not include placement and routing, which may degrade performance. Compared to the baseline SPR

implementation that does include the implementation flow, Offset Pipelining provides 1.7x better performance on the evaluated benchmarks.

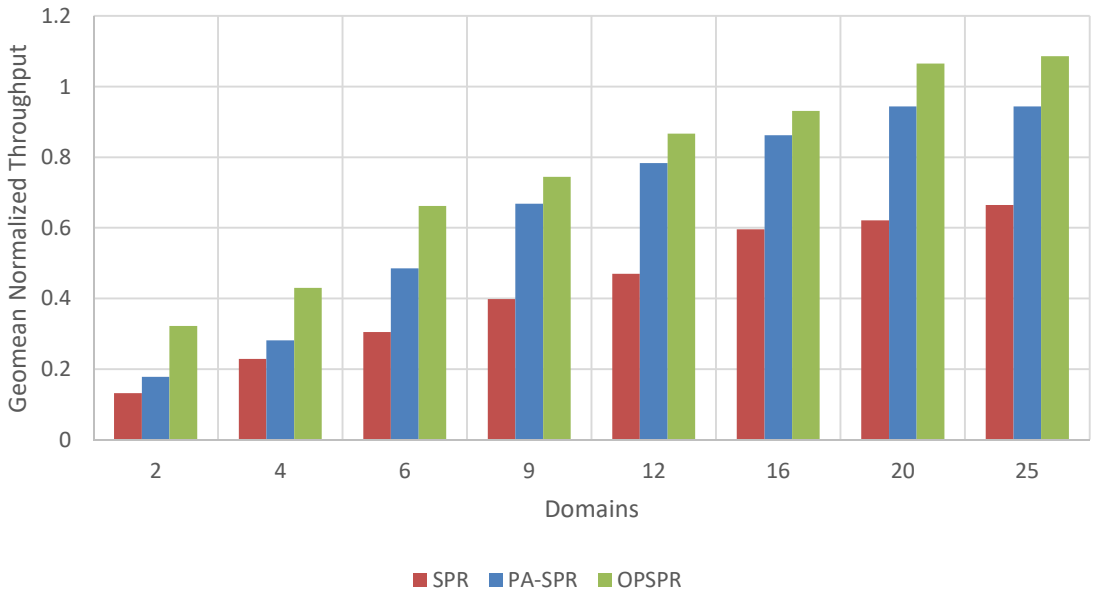

Figure 10.8. Offset Pipelining vs SPR and PA-SPR bound with performance normalized per benchmark to throughput of recurrence limited monolithic version.

10.4 Detailed Results

This section provides per benchmark results of the data summarized in the previous sections. The figures present the number of cycles needed to execute the benchmark under various scenarios described here:

- OPS-single – Offset Pipelined scheduling targeting a monolithic dataflow graph.
- IMS – Iterative modulo scheduling of the monolithic dataflow graph.
- SPR – Full implementation flow of baseline SPR.
- PA-SPR – Bound on predicate aware SPR calculated form worst case mode II
- OPS – Offset Pipelined scheduling for multi-mode dataflow graph.
- OPSPR – Full Offset Pipelining implementation.

There are a few notable features of these results for evaluating Offset Pipelining. Most obviously, Offset Pipelining is inferior to modulo scheduling for single mode applications. On the other hand, highly modal applications such as Bayer and DWT provide significantly better performance than a modulo scheduled approach when resource limited. The PET application also sees a substantial performance advantage with Offset Pipelining on few resources despite having only two modes. The key feature of this application is that a simple mode with a short recurrence II executes significantly more frequently than a much larger mode that executes very infrequently. Such an application would be limited using PA-SPR because it must accommodate the larger mode II.

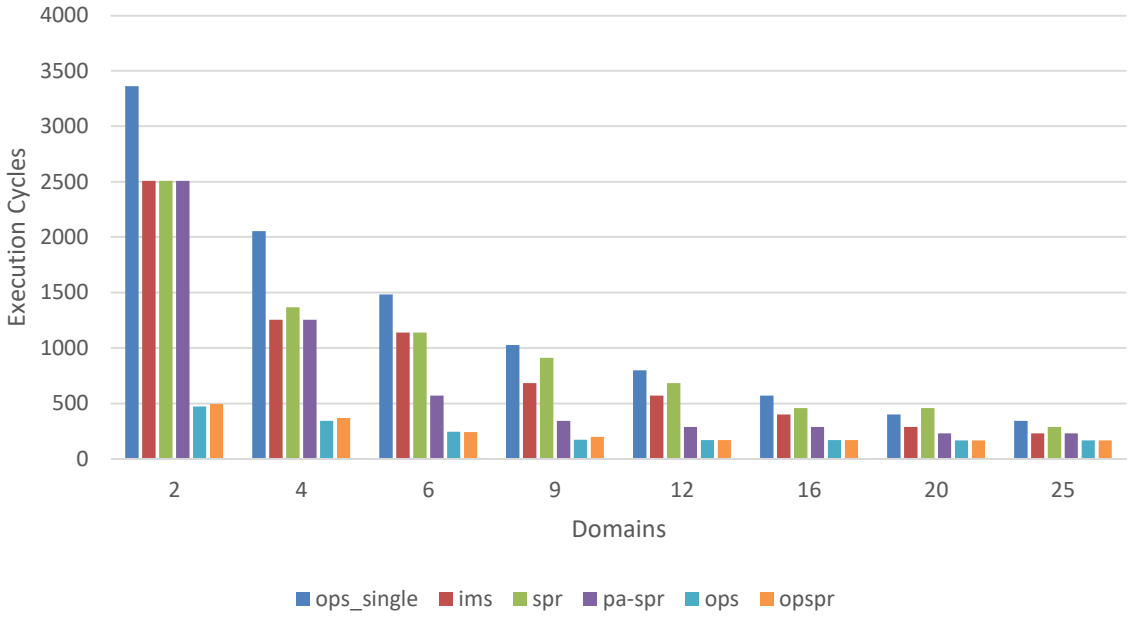

Figure 10.9. Cycles to execute Bayer benchmark.

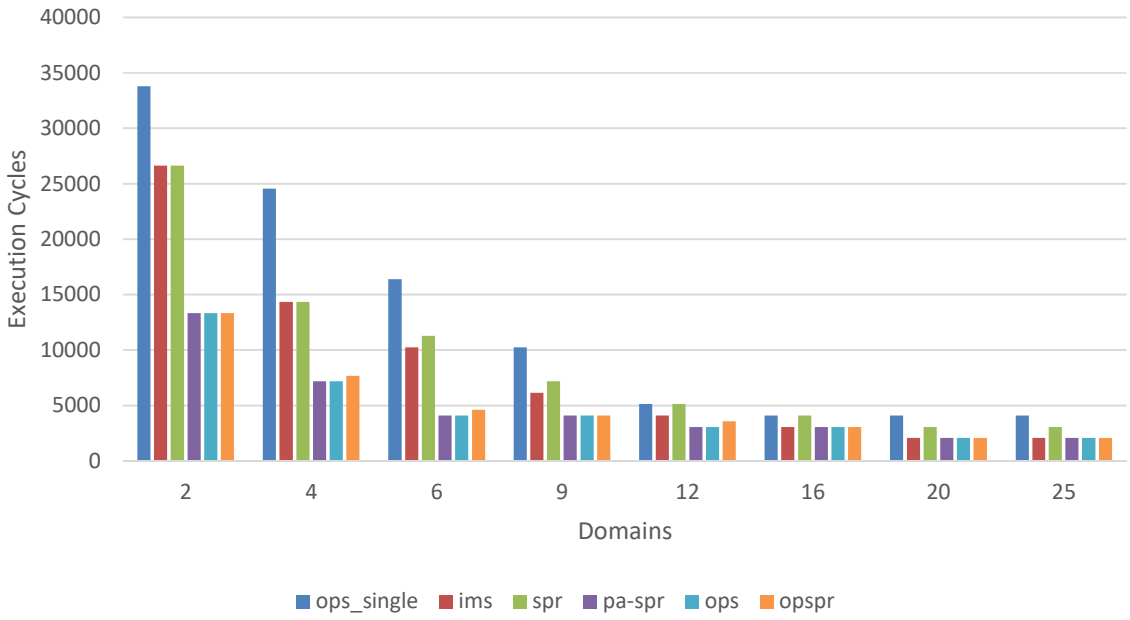

Figure 10.10. Cycles to execute DCT benchmark.

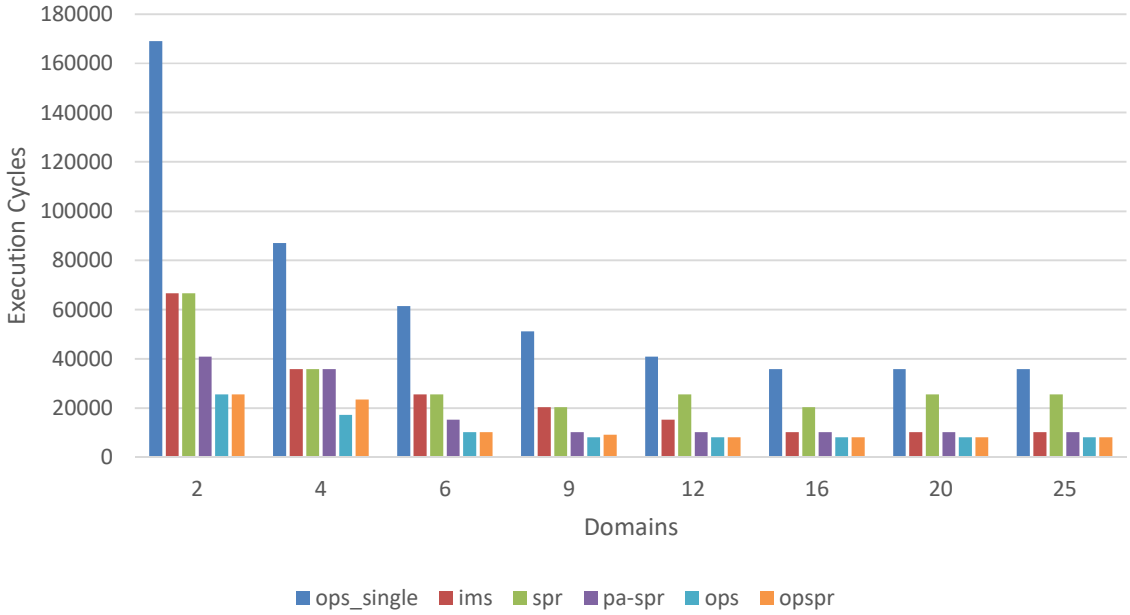

Figure 10.11. Cycles to execute DWT benchmark.

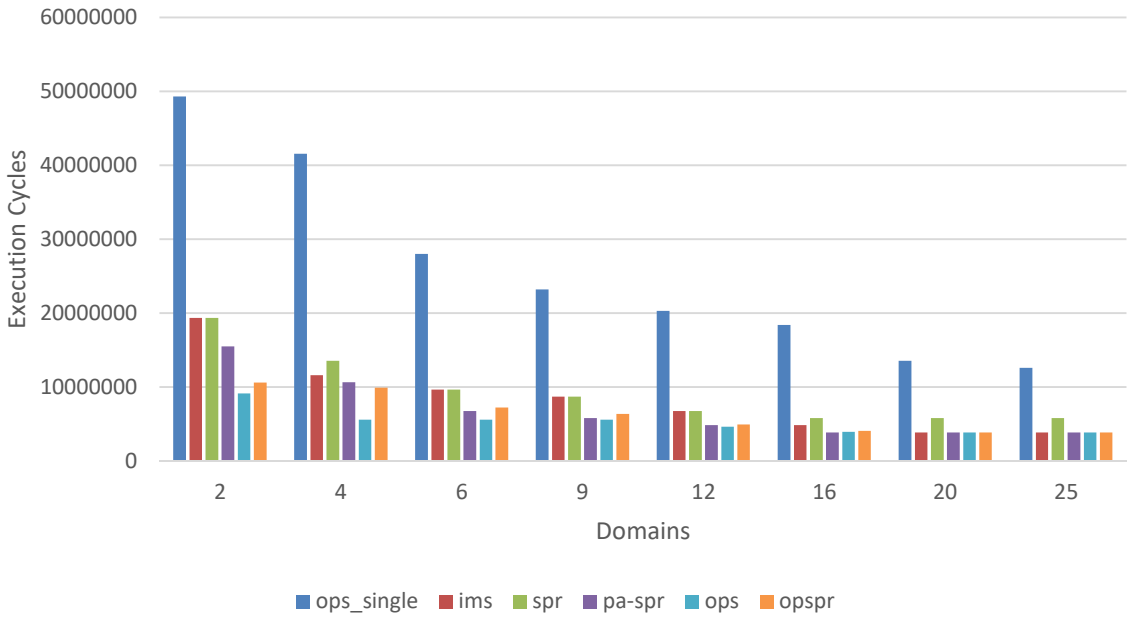

Figure 10.12. Cycles to execute K-means benchmark.

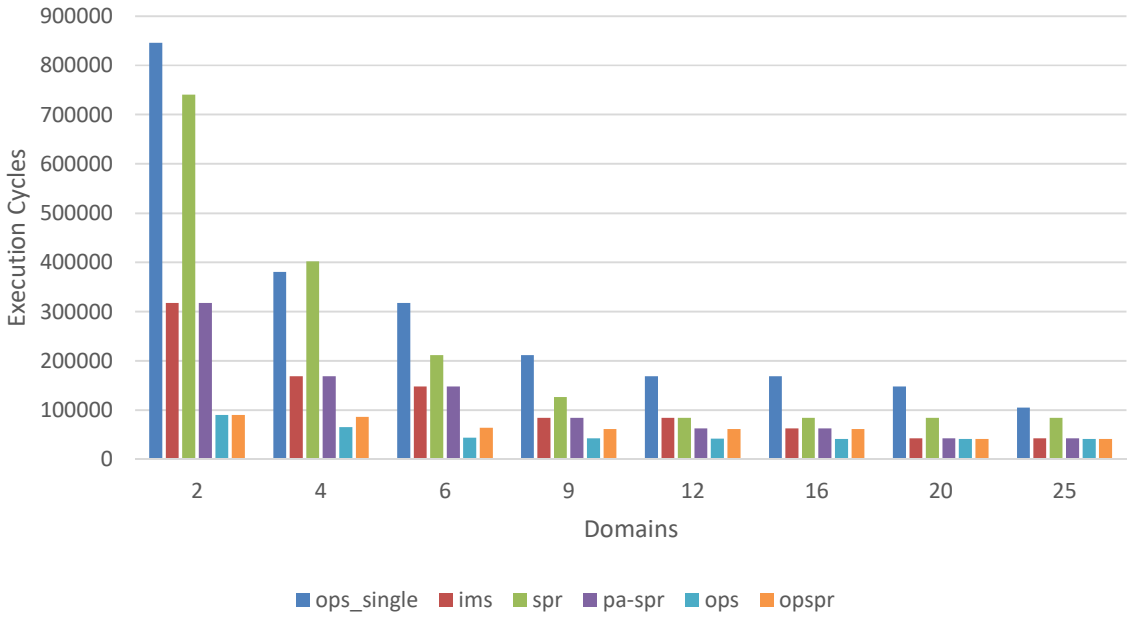

Figure 10.13. Cycles to execute PET benchmark.

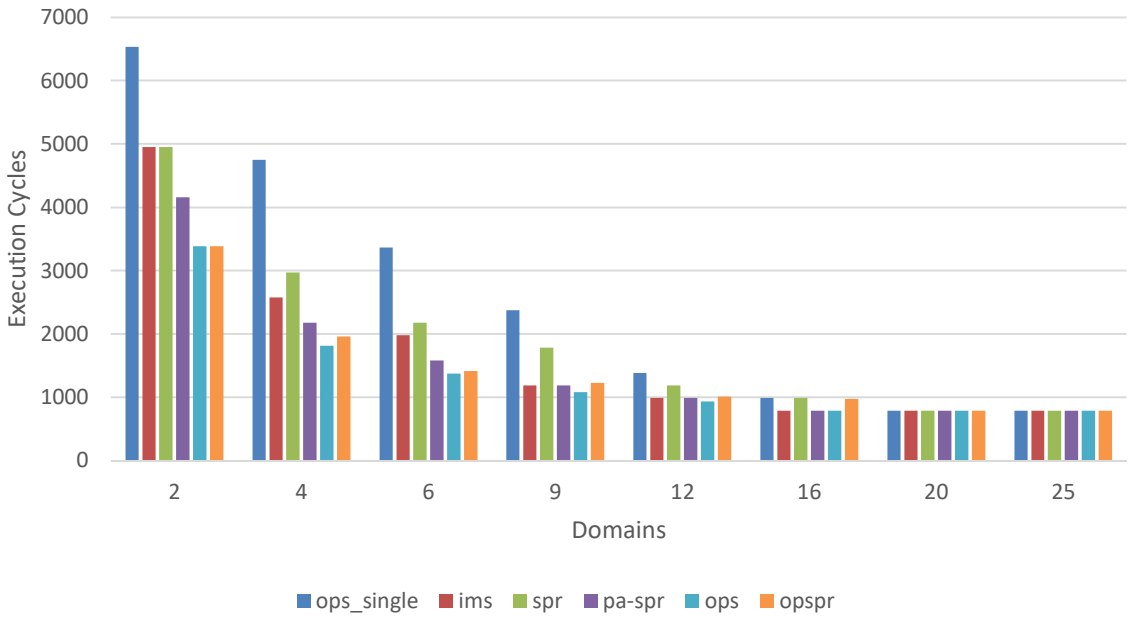

Figure 10.14. Cycles to execute RabinKarp benchmark.

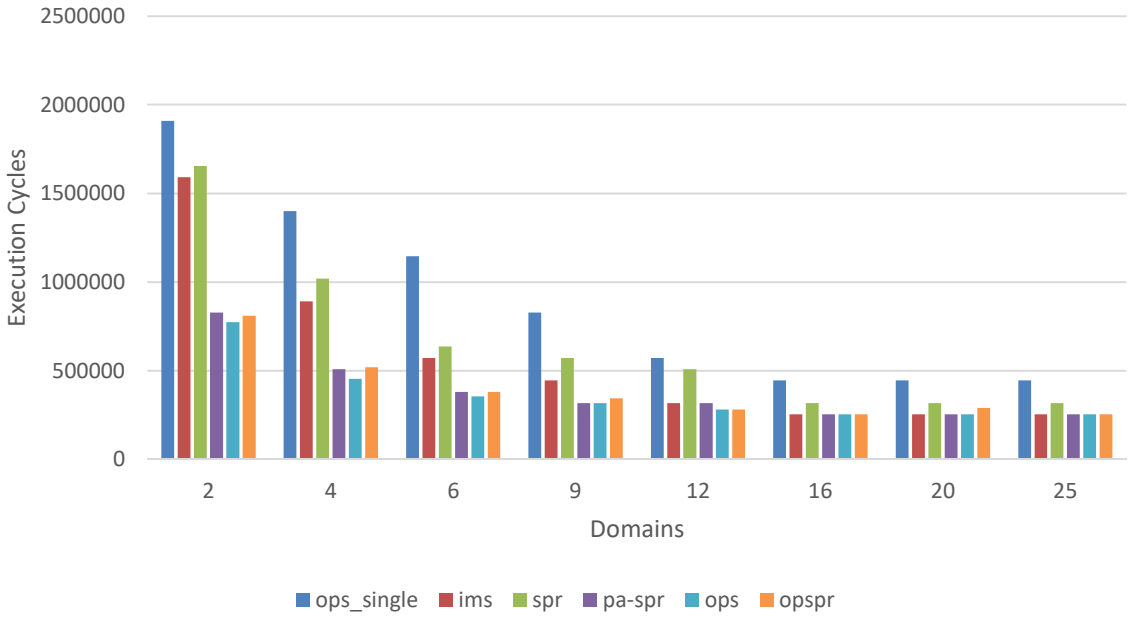

Figure 10.15. Cycles to execute RSA benchmark.

Chapter 11. RELATED WORK

This chapter draws comparisons to other architectures and tools to position Offset Pipelining in the broader context of reconfigurable computing research. We focus on comparing features of Offset Pipelining and pipelined program counter (PPC) architectures to highlight the unique aspects of this work and introduce alternative approaches that have been explored in the literature, each with unique strengths and goals.

The scope of computing hardware and software is enormous even restricted to the realm of reconfigurable computing. This work adds to the list of efforts focused on CGRA architectures and tools. The majority approach innovation from an architecture perspective with many examples such as RaPiD [ECF96], MATRIX [MD96], MorphoSys [SLL+00], and REMARC [MO98]. Such architecture explorations established the performance and power efficiency benefits of CGRAs though tools lagged behind [ECF97] and adopted conventional parallel programming techniques to extract parallelism. However, this approach misses the critical component of development productivity that stunts commercial enthusiasm. The most closely related work is the Mosaic project including SPR, previously discussed in Chapter 9.

The PPC CGRA presented in this work is a direct descendent of the Mosaic CGRA [VE10]. Sharing a similar resource composition and interconnect organization, the fundamental difference lies in how configuration contexts are retrieved for execution. As noted in Chapter 3, we replace modulo counter control with pipelined program counters to facilitate the Offset Pipelining execution model.

11.1 ARCHITECTURE

Research focused on CGRAs has demonstrated the potential performance advantages compared to commodity architectures, for a variety of applications, each with different approaches to device organization and resource composition including Mosaic [VE10], DVLIW [ZFM+05], RaPiD [ECF96], and ADRES [MVV+03]. Roadmaps such as those from UC Berkeley [ABC+06] and DARPA [KBB+08] have highlighted the limits of conventional architectures as fabrication technology continues to advance. A desire for high performance with greater power efficiency, both at large scales for datacenters and small scales for embedded and mobile devices

helps drive interest in and development of CGRAs. This section reviews features of our PPC CGRAs compared to existing architectures.

11.1.1 *Pipelined Program Counters*

There are a variety of control mechanisms explored in CGRA research. RaPiD uses a standalone controller for managing heavily pipelined loop nests, Morphosys relies on a host processor for context selection, and Mosaic context selection is controlled by modulo counters. Modulo counter control has also appeared in Tabula [Teig12] FPGAs. In the majority of cases, device resources can be time multiplexed, in order to take advantage software pipelining and spatial locality in the application mapping.

While a VLIW machine maintains a conventional program counter and the ability to move in and out of modulo scheduled regions of an application, this is too costly for CGRAs. A spatial architecture necessarily has a much larger per cycle configuration size so the effectively unbounded program size of a VLIW processor is not practical. Our proposed pipelined program counters allow a greater variety of run time control than a modulo counter working within the constraints of CGRA architectures.

Even though each domain in a PPC CGRA uses a program counter for sequencing, this functionality should not be confused with the flexibility of a conventional processor. The program must still be short since all configuration memory is considered to be on chip, with the device statically scheduled. This is also distinct from a massively parallel processor array (MPPA) like Ambric [BJW07] in that there is a separate interconnect network for propagating program control information to be discussed in 11.1.4.

11.1.2 *CGRAs as Accelerators*

The majority of CGRA research assumes an accelerator architecture with the device connected to a host processor. The host is responsible for organizing data for consumption by the CGRA and managing the overall computation. Projects like Mosaic, ADRES, DVLIW, and many others fall into this category. Massively Parallel Processor Arrays or MPPAs such as Ambric [BJW07] move away from this notion somewhat, offering a more autonomous solution. This highlights a distinction between CGRAs and MPPAs, where does control of the execution reside? Our

pipelined program counter architecture is statically scheduled just as Mosaic CGRAs and most others are. This limits the program length compared to conventional processors.

Our target architecture is based on the Mosaic CGRA [VE10]. Broadly, the Mosaic device is constructed from a 2-D array of clusters. Each cluster is composed of a number of computational resources connected to a local crossbar and inter-cluster communication occurs over a grid interconnect. While this work replaced modulo counters with pipelined program counters, the architecture is otherwise similar. Experiments with Mosaic explored a variety of topics including front end compilation with Macah [YCF+08, Ylv10], the back end CAD tools [FCV+09], and architecture optimization [VE10]. Note that "control domain" in this work is synonymous with "cluster" in the previous work on SPR [Fri11]. We adopt the term domain to reflect the addition of the program counter that provides these clusters with additional flexibility when scheduling.

11.1.3 *Compute Resources*

Each CGRA architecture has its own mix of available resources. Despite these differences, they all contain arithmetic, logic, and memory resources onto which a computation is mapped. Refinements to the Mosaic CGRA included different types of ALUs to better match the requirements of the targeted applications.

Conventional register files perform poorly in modulo scheduled architectures due to name collisions in the schedule requiring code duplication. Rotating register files alleviate this problem in modulo scheduling [RG81]. On the other hand, our modal execution model does not allow these collisions as a consequence of the explicit issue slots windows. PPC CGRAs replace rotating register files with conventional ones given the difficulty of maintaining register renaming across mode boundaries. As in the Mosaic project, our CGRAs also include FPGA style look-up table (LUT) resources to handle logic that would map inefficiently to word wide ALUs. Stateless computation resources can receive a different configuration for each cycle of the schedule, up to a maximum supported by the device.

Small memory blocks, comparable to memory blocks available in FPGAs, are available throughout the devices. Some CGRAs offer a more heterogeneous organization with memory available in generally larger blocks on an edge of the device rather than distributed throughout [PFK+06].

The Tabula [Teig12] SpaceTime architecture was a commercial product similar to a CGRA using a modulo counter mechanism for time multiplexing. While most CGRAs are dominated by word oriented resources, these devices were LUT based. The tool chain was designed to not expose time multiplexing to the developer instead providing a conventional FPGA tool chain abstraction for the underlying hardware. A purported benefit of this approach was to present memory blocks in the architecture as highly multi-ported with the user clock exposed to the designer a fraction of the context switching clock. This would effectively multiply the number of physical ports by the schedule length and presents a significantly different architectural model to the designer.

11.1.4 *Interconnect*

Scalable tile to tile communication for spatial architectures can be accomplished with an island style interconnect organization. Like FPGA architectures [BR97], each tile is connected to its neighbors in a regular pattern across the device. Mosaic explores static and dynamic interconnect resources to evaluate the tradeoff between mapping quality and cheaper resources [VEWC+09]. Static interconnect has only one configuration for the life of the application, similar to FPGA interconnect. Dynamic resources receive a new configuration each cycle of the schedule. Our architecture uses dynamic interconnect resources exclusively since we are not focusing on optimizing the architecture, a task left for future work. Each domain in our devices contains a crossbar to provide intra-domain connectivity in the same way as Mosaic CGRAs. Inter-domain communication is a limited resource for routing subject to negotiated congestion [ME95] discussed in section 8.9. The interconnect is register rich, allowing the target device to operate at a specified frequency regardless of the application mapping. A fixed frequency, up to 1 GHz on a 65nm process for Mosaic, eliminates the challenge of timing closure found in conventional FPGA design, though it is replaced by the pipelined routing problem.

Pipelined program counter CGRAs add an additional interconnect network to handle program counter information. Each domain, other than the leader, receives its program counter over this network from adjacent domains. The network is configured to ensure that each domain receives the program counter value at the appropriate latency relative to the leader to properly sequence program execution. The replacement of modulo counters with program counters, and the addition of the program counter routing network, are the two features that define the PPC

CGRA in contrast to the Mosaic architectures. Expected overhead for these modifications depends on the granularity of control domains in the target architecture and the resource composition. For Mosaic architectures, related work on dataflow driven execution control [PH12] concludes that this and other modifications to support distributed stalling mechanisms would be a less than 2% area overhead.

11.2 SCHEDULING

Modulo scheduling has a long history in conventional VLIW compiler literature [Rau94, WP95, LGA+96] before being adopted for CGRA mapping in Mosaic [FCV+09], ADRES [MVV+03], and the architecture targeted by edge-centric modulo scheduling [PFK+06]. While SPR adopted iterative modulo scheduling [Rau94], other approaches such as swing modulo scheduling [LGA+96] and edge-centric modulo scheduling [PFM+08] have been explored as well for software pipelining onto CGRA architectures. Swing modulo scheduling works to limit register pressure by minimizing the number simultaneous live values in the resulting schedule. Edge-centric modulo scheduling focuses on routing optimization, ultimately producing a placement as a byproduct. All of these techniques share the limitation of mapping the target code to a single modulo schedule. Enhanced loop flattening [YEH10] and predicate aware sharing [Fri11] work together to avoid this drawback, but as the control complexity of the target application increases, mapping quality suffers.

When targeting multi-mode applications, Offset Pipelining offers a mapping approach to optimize individual software pipelined portions of the application, while retaining the benefits of a short, statically scheduled program. Modulo scheduling with multiple initiation intervals [WP95] provides a similar solution to this problem for conventional VLIW machines. Unfortunately, this is not feasible on CGRA architectures due to longer program length and an inability to distribute the necessary changes in control flow on a cycle by cycle basis. Developed specifically for CGRAs, Offset Pipelining recognizes these issues by staggering issue slot windows through assigning domain offsets. Other CGRAs that share this staggered approach [PFK+06] remain confined to a single II through modulo scheduling.

Alternative scheduling approaches include formulating the problem for optimization with simulated annealing [KGV83, MVV+02] or moving to a different execution model. MPPAs such as Ambric [BJW07] are composed of discrete processors. These architectures are less

tightly integrated than typical CGRAs but can support modulo scheduling across the array. However, they are typically configured by writing individual communicating programs or using traditional parallel programming techniques. While this may be preferred for multi-core architectures, it is difficult to scale applications to architectures like Ambric with hundreds of processors.

11.3 Routing

Beyond the routing background introduced in Chapter 7, most routing for CGRAs is produced as a byproduct of the placement process, usually in conjunction with register assignment. CGRAs that are mapped in this way [HMS+2013, PFK+2006] are more closely related to conventional VLIW machines, lacking a rich programmable interconnect. As a descendent of the Mosaic project, PPC CGRAs rely on pipelined routing [LE04] and negotiated congestion [ME95] to connect the dataflow operations in the application.

11.4 Refining Mapping Techniques

Research on CGRAs has shifted somewhat from the architectural focus of early works toward refining mapping tools. Targeting the CGRA in [PFK+06], EPIMap [HSV12] explores a heuristic approach to dataflow graph transformation to optimize mapping while REGIMap [HSV13] splits the mapping approach into a scheduling and merged placement and routing phase focusing on register assignment. Analytical approaches to graph transformation have also been explore [LYL+13], competitive with REGIMap for mapping quality. Addressing access to memory for CGRA systems looks at the broader infrastructure for practical application of these architectures. The organization of memory resources and managing data movement have been explored [YLS+10] along with transforming the application to avoid conflicts in memory access [YLS+11].

There continues to be innovation in the architecture space as well with an emphasis on mapping efficiency. Modifying the functional units to support dual issue [HSV14] requires significant compiler support for merging these operations. Approaches based on head predication [RSH15] to avoid unnecessary fetch overhead focuses on maximizing energy

efficiency. This refined previous work on a full predication technique that required corresponding mutually exclusive operations on the same functional units [HCL13].

Chapter 12. CONCLUSION

This dissertation presents the Offset Pipelining execution model and an accompanying set of algorithms to map applications to CGRAs supporting that execution model. Offset Pipelined Scheduling enables more efficient mapping of multi-mode signal processing applications, improving the flexibility of CGRA systems. The EveryTime router solves new challenges faced by Offset Pipelining and provides a practical implementation strategy complementing existing tools like PA-SPR that excel for single mode applications. These tools contribute to broadening the utility of CGRA systems. The intention of this work is to contribute to the drive toward accessible hybrid computing solutions to leverage the potential of spatial computing architectures.

12.1 Execution Model

The Offset Pipelining approach software pipelines target code with a granularity finer than that of a monolithic modulo schedule while working within the limits of CGRA architectures. A statically scheduled device supporting more complex branching control flow simplifies hardware design while trying to maintain processor-like flexibility. The independent mode IIs afford the flexibility to optimize individual portions of the target code for improved throughput. The tradeoff of fixed issue slot windows determined by II and offset constrains operation mapping. However, the offset constraints also help ensure that control information is given sufficient time to propagate across the device at run time, a limitation not found in conventional VLIW architectures.

The benchmark applications in this work highlight the benefits of Offset Pipelining in taking advantage of multi-mode execution. For target code comprised of a single loop with little conditional execution, predicate aware modulo scheduling is a superior approach. Very large and complex applications likewise may not be suitable for mapping to a CGRA. Since a pipelined program counter CGRA can easily support modulo scheduling, PPC architectures should be considered a more capable replacement for the conventional modulo scheduled CGRA given the expanded scope of application complexity that can be efficiently mapped.

12.2 SCHEDULING

The Offset Pipelined Scheduling algorithm maps applications to take advantage of the execution model. Adopting an iterative approach, the algorithm explores the scheduling space to fit the application to the available resources. OPS shows improvement over modulo schedules for some selected applications, particularly in resource limited situations. When the application is recurrence limited, the resulting mapping often requires fewer resources for a more compact implementation. This helps maintain a high degree of locality, helpful for subsequent placement and routing. The new scheduling algorithm is further evaluated against PA-SPR which approaches the problem of conventional predicated execution on CGRAs in a different way. While Offset Pipelining has an advantage for highly modal applications with different IIs, PA-SPR can optimize intra-mode mutual exclusion. These two approaches might be combined to offer a more powerful suite of tools for CGRA mapping.

12.3 ROUTING

Offset Pipelining and the associated scheduling algorithm tend to increase routing complexity to accommodate the expanded branching run time behavior. The EveryTime router tracks the set of possible locations and times that resources may be active during routing in order to guarantee signals arrive under any possible run time execution sequence. The routing formulation is compatible with a negotiated congestion framework for resolving global routing contention. The approach is able to handle the Offset Pipelining execution model and route designs with comparable channel widths to prior work. The ability to route designs with little overhead despite the increased complexity demonstrates that the benefits of Offset Pipelining can be realized in practical implementation.

12.4 RETROSPECTIVE

The idea for Offset Pipelining grew out of a desire to continue CGRA development in the vein of the Mosaic project. It was driven by the question of how to get better performance out of CGRAs. This depends a great deal on the application. Many practical scenarios will be limited by factors outside the underlying the computing substrate. Still others may be limited by details of the algorithm solving a particular problem. However, I saw a need to increase the scope of

code that is typically mapped to CGRAs. The computationally intensive inner loops of applications are the primary target of most work in reconfigurable computing. Whether it is an FPGA, CGRA or even a GPU, a host processor usually manages the marshalling of data to an accelerator. While this is a useful model for a wide range of important applications, if the CGRA is more powerful and can efficiently handle modal application behavior, this can avoid expensive off-chip communication. Offset Pipelining offers an execution model designed to support different phases of application execution beyond an inner loop. Supporting per-mode IIs allows optimization of inner loops, but also surrounding code, making a PCC CGRA a more flexible platform for signal processing.

Evaluating Offset Pipelining in the context of the broader array of computing options, we can see the challenge of promoting CGRAs as a competitive platform. Despite slowing single threaded performance increases, multi-core processors continue to improve power efficiency and have an enormous advantage of flexibility and rapid development. GPUs likewise offer impressive performance for certain classes of applications and offer a palatable programming model. Development productivity sinks rapidly for FPGAs while gaining the benefit of highly customized hardware solutions. Unfortunately, CGRAs do not yet make a compelling case compared to established devices. The tools are immature compared to conventional hardware or software flows and there are many subtleties in the various hardware implementations.

12.5 LIMITATIONS

The results presented in this work are promising. However, a more critical perspective is equally important to evaluating the conditions under which Offset Pipelining and the associated mapping algorithms are useful. The benchmarks selected for our evaluation all include significant modal behavior making them good candidates for Offset Pipelining. Applications without this feature would be mapped no better, and usually worse, than with existing techniques.

Many applications can be decomposed into a series of processing steps that form a pipeline of communicating processes. This eliminates the modal behavior desirable for Offset Pipelining. Moving large amounts of data around a device is increasingly a dominant factor in energy consumption [KBB+08], but such a pipeline might be preferable to maximize throughput. Offset Pipelining would likely be less useful in these scenarios, but could still be competitive if the available hardware is highly constrained or if efficiency is preferred over raw throughput.

Unlike the Mosaic project, this work does not include a front end compiler generating dataflow graphs for scheduling. With hand written benchmarks, it was infeasible to customize many versions to target various device sizes. The result that Offset Pipelining excels in a resource limited regime could be bolstered if the applications were automatically tuned to the available resources. This would make it easier to remain resource limited during the mapping to avoid recurrence limited performance.

One of the main goals in exploring the use of spatial architectures like CGRAs is to promote scalable hardware and an accompanying execution model. While control in modulo scheduled CGRAs can be readily distributed among replicated modulo counters, this is not the case for Offset Pipelining. Run time control is ultimately centralized in the offset 0 domain. For applications significantly larger than the benchmarks used here, this centralization may ultimately limit the potential scalability.

A practical architecture might also include computational elements with larger latencies than those explored here. In this work, the majority of operations execute in a single cycle with multiplication being the exception with a two-cycle latency. As latencies increase, the mapped domain offsets likely increase as well which puts more pressure on the EveryTime router as the number of possible execution paths go up with increased offset spacing. To avoid overprovisioning the channel width, striking the correct balance in the device architecture is important.

12.6 Future Work

There are a wide variety of areas to explore related to Offset Pipelining. This initial foray into the execution model and algorithms demonstrates a prototype tool flow but certainly does not thoroughly explore all aspects of the system. This sections points out some areas where further research may be fruitful and imagines Offset Pipelining as a component in a larger CGRA tool suite.

12.6.1 *Front End Compiler*

The algorithms presented in this work form the central pieces of a tool flow for mapping to PPC CGRAs, but notably absent here is a front end compiler for generating dataflow graphs that can be consumed by OPS. Just as the Macah [Ylv10] compiler provided an auto-tuning mechanism

for SPR mapped applications in the Mosaic project, corresponding work for Offset Pipelining could provide a suite of optimizations for PPC CGRAs. This effort might include automatic or programmer specified mode decomposition, transformations such as loop unrolling, and a facility to specify an optimization formula to help tailor the application to the hardware before beginning the back-end mapping flow. The LLVM [LA17] compiler infrastructure, while focused on conventional processor architectures, uses a SSA intermediate representation well suited to providing input to a back end CGRA flow.

12.6.2 *Scheduling Alternatives*

The algorithms themselves could be significantly refined to explore alternative scheduling, placement, and routing strategies as well as automatic application tuning based on feedback from the mapping tools. OPS is inspired by iterative modulo scheduling, but integrating concepts from alternatives like edge-centric modulo scheduling or swing modulo scheduling could improve scheduling. Providing a greater degree of architecture awareness during scheduling would also be useful in improving mapping quality. While the bulk of dataflow graph optimization would likely reside in a front-end compiler, the scheduling process might also integrate some automatic tuning. Lastly, the offset shaping process seems like a ripe target for improvement, perhaps by formulating the problem as an integer linear program as a starting point.

12.6.3 *Device Architecture*

The device architecture itself should also be explored in terms of domain and interconnect composition. A detailed architecture exploration would highlight features that are useful for Offset Pipelining that may be different from conventional CGRAs. While the Mosaic project indicated that scheduled resources are preferable for the bulk of the interconnect, this is not necessarily the case for Offset Pipelining if static routes can be effectively shared at mode transitions. Crossbar depopulation and the tile connectivity pattern should similarly be explored.

A major concession made while developing Offset Pipelining was to remain limited to a homogenous architecture. The original plan to provide different domain types with different resource compositions proved to be problematic for the scheduling algorithm and defining the target architecture required some baseline understanding of how applications are mapped to the

device. With this initial work completed based on the Mosaic cluster composition, it would be an interesting extension to solve the heterogeneous scheduling problem and optimize the architecture around this capability.

12.6.4 *Offset Pipelining as a Component*

While the focus of this work has been on multi-mode application mapping, there is an important broader perspective to consider for CGRA mapping tools. CGRAs supporting Offset Pipelining have two inherent properties that would facilitate use in a larger tool chain infrastructure. The pipelined program counters can be configured to perform as modulo counters for existing tool flows such as PA-SPR that leverage modulo scheduling, so for code consisting of a simple loop, these tools may be preferable to an Offset Pipelined approach. These CGRAs could also be decomposed into independent regions that do not have a fixed scheduling relationship. One region might be modulo scheduled while another Offset Pipelined. They can then communicate through asynchronous channels to build a larger application out of multiple subprogram kernels. Provided the application can be partitioned effectively, such a tool chain could use the best tool for each portion of the application to map to a region of the device. As noted in [PH12], the area overhead for mechanisms to support distributed stalling and program counters would be a less than 2%.

The decision to use modulo scheduling or Offset Pipelining comes down to three major factors:

- How complex is the control flow of the target code?
- How many resources are available for the mapping?
- How sensitive is the application to latency?

It is clear that a single loop matches the modulo scheduling paradigm. Applying Offset Pipelining to a single loop with no branching can be no better than modulo scheduling. It may be worse if the constraint of issue slot windows limits the scheduling.

An application with significant branching can still be modulo scheduled by using predication or predicate aware sharing to provide the necessary control flow. Such an application might still achieve recurrence limited performance if enough resources are available. However, even with abundant resources, the application will eventually become limited by communication due to the physical placement of operations on the device. In the extreme, a

large application with significant branching that has been modulo scheduled will be limited by how quickly control information can be propagated across the device.

On the other hand, Offset Pipelining is designed to support diverse control flow, particularly when an application has distinct phases of execution. It offers better performance by allowing each mode to execute with its own II rather than a single II for the entire application.

Predicate aware sharing introduced in Chapter 9 provides some of the same advantages as Offset Pipelining. Operations can share physical resources, saving issue slots and providing a more compact implementation, which is beneficial for subsequent tool phases. Predicate aware sharing is particularly powerful for managing conditional behavior in inner loops where conventional predication would consume a larger number of resources. On the other hand, the benefit of multiple, independent IIs in Offset Pipelining is suitable for applications that include a sequence of loops, with each implemented as an independent mode. This organization helps optimize each mode in relative isolation to improve overall throughput. A further refinement to the Offset Pipelining tool chain would add support for predicate aware sharing to further broaden the scope of applications that can be efficiently mapped.

A combination of predicate aware sharing with the multi-mode support of Offset Pipelining would provide a versatile infrastructure for CGRA application mapping. While a sequence of loop bodies would be best handled by Offset Pipelining to leverage independent IIs, predicate aware sharing would be applied to conditional execution within the modes to further compact the resulting mapping.

Offset Pipelining offers an alternative execution model for modal applications on CGRAs. While useful for these applications, it does not replace other techniques like PA-SPR. Other research as part of the Mosaic project [Kni10] explored decomposing applications into asynchronous communicating kernels and partitioning the target device to allow different computations to coexist simultaneously. Offset Pipelining becomes a tool for implementation within a region of the device managed by a higher level partitioning tool such as the Mosaic floorplanner [WKY+12]. This opens the door for further optimization tools that refine the implementation strategy to fit larger and more complex applications on CGRA devices.

12.6.5 *Machine Learning Applications*

An excellent example of an application space that might benefit from CGRAs is machine learning. Exploring benchmarks in this space on Offset Pipelined CGRAs would be very interesting, perhaps suggesting new features to add to the architectures or adjustments to the mapping tools to reflect this area of research. Modal application support in Offset Pipelining might be particularly useful for online training and execution of neural networks allowing the device to rapidly switch between these modes. Companies like Google are already exploring custom silicon for these applications in support of the TensorFlow framework [AAB+16]. CGRA architectures could become a central component of integrating machine learning into practical solutions. They can be configured to accelerate the long running, compute intensive dataflow graphs inherent in TensorFlow applications and can additionally support the modest control needs. IBM has taken a different approach to machine learning with spatial computing [MAA+11]. Their neuromorphic architecture aims to mimic biological systems with very low power consumption and asynchronous signaling across the network of neuron cores.

Other companies are exploring machine learning using existing devices rather than custom architectures. Besides a neuromorphic approach, IBM researchers are also integrating FGPAs with POWER servers [Wittig16] demonstrating significant efficiency gains. Microsoft has moved beyond research, using FPGAs initially for accelerating search [PCC+14] and continues to refine the architecture [CCP+16] applying FPGAs to deep learning. The combination of reconfigurability to accommodate new applications and customized datapath precision have proved highly capable while remaining within a reasonable power budget. CGRAs tailored to machine learning would adopt the desired datapath width to move toward an architecture with the benefits of reconfigurability combined with resources optimized for the application domain.

12.7 PARTING THOUGHT

The goal of Offset Pipelining is to explore possible techniques to get the most performance and efficiency out of available silicon resources. By giving designers more flexibility in application mapping, this work increases the scope of CGRA applications, making the CGRA a more compelling option for future research and commercial viability. Offset Pipelining attempts to

strike a balance between practical hardware architectures and automatic high quality mapping. Hopefully this work serves to accelerate future development of CGRA technologies.

BIBLIOGRAPHY

[AAB+16] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

[ABC+06] Asanovic, Krste, et al. The landscape of parallel computing research: A view from berkeley. Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ACD74] Adam, T.L., Chandy, K.M., and Dickson, J.R. A comparison of list schedules for parallel processing systems. Communications of the ACM 17, 12 (December 1974), 685-690.

[BJW07] M. Butts, A. Jones, and P. Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'07), pages 55-64, April 2007.

[BR97] Vaughn Betz and Jonathan Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In International Workshop on Field-Programmable Logic and Applications, 1997.

[CCP+16] Caulfield, Adrian M., Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil et al. "A cloud-scale acceleration architecture." In Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on, pp. 1-13. IEEE, 2016.

[CFV+07] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency. Technical report, Department of Energy NA-22 University Information Technical Interchange Review Meeting, 2007.

[ECF96] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In Reiner W. Hartenstein and Manfred Glesner, editors, International Workshop on Field-Programmable Logic and Applications, pages 126–135. Springer-Verlag, Berlin, 1996.

[ECF97] Ebeling, C., Cronquist, D. C., Franklin, P., Secosky, J., & Berg, S. G. (1997, April). Mapping applications to the RaPiD configurable architecture. In Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on (pp. 106-115). IEEE.

[FCV+09] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, S. Hauck, "SPR: An Architecture-Adaptive CGRA Mapping Tool", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, pp. 191-200, 2009.

[Fri11] Stephen Friedman, Resource Sharing in Modulo-Scheduled Reconfigurable Architectures, Ph.D. Thesis, University of Washington, Dept. of CSE, 2011. http://www.ee.washington.edu/people/faculty/hauck/publications/friedmanPHDthesis.pdf

[HCL13] Han, K., Choi, K., and Lee, J. Compiling control-intensive loops for cgras with state-based full predication. In Proc. DATE (2013), pp. 1579–1582.

[HDL+09] M. Haselman, D. DeWitt, T. K. Lewellen, R. Miyaoka, S. Hauck, "FPGA-Based Front-End Electronics for Positron Emission Tomography", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, pp. 93-102, 2009.

[HSV12] Hamzeh, M., Shrivastava, A. and Vrudhula, S., 2012, June. EPIMap: Using epimorphism to map applications on CGRAs. In Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE (pp. 1280-1287). IEEE.

[HSV13] Hamzeh, M., Shrivastava, A. and Vrudhula, S., 2013, May. REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In Proceedings of the 50th Annual Design Automation Conference (p. 18). ACM.

[HSV14] Hamzeh, Mahdi, Aviral Shrivastava, and Sarma Vrudhula. "Branch-aware loop mapping on CGRAs." In Proceedings of the 51st Annual Design Automation Conference, pp. 1-6. ACM, 2014.

[Huff93] R. A. Huff. Lifetime-sensitive modulo scheduling. In Proc. of the ’93 SIGPLAN Conf. on Programming Language Design and Implementation, pp. 258-267, Albuquerque, NM, June 1993.

[KBB+08] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W. and Hill, K., 2008. Exascale computing study: Technology challenges in achieving exascale systems.

[KGV83] S. Kirkpatrick, Jr. Gelatt, C. D., and M. P. Vecchi. Optimization by Simulated Annealing. Science, 220(4598):671–680, 1983.

[Kni10] Knight, Adam. "Multi-Kernel Macah Support and Applications." Master thesis, University of Washington, 2010.

[LA17] Chris Lattner and Vikram Adve. LLVM. [Online] http://llvm.org, January 2017.

[Lam88] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", In Proceedings of the ACM SIGPLAN 88 Conference on

Programming Language Design and Implementation (PLDI 88), July 1988 pages 318-328. Also published as ACM SIGPLAN Notices 23(7).

[LE04] Song Li and Carl Ebeling. “QuickRoute: A Fast Routing Algorithm for Pipelined Architectures”, IEEE International Conference on Field-Programmable Technology, pp. 73-80, Dec. 2004.

[LGA+96] Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. "Swing module scheduling: a lifetime-sensitive approach." In Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on, pp. 80-86. IEEE, 1996.

[LYL+13] Liu, Dajiang, Shouyi Yin, Leibo Liu, and Shaojun Wei. "Polyhedral model based mapping optimization of loop nests for CGRAs." In Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, pp. 1-8. IEEE, 2013.

[MAA+11] Merolla, Paul, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S. Modha. "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm." In Custom Integrated Circuits Conference (CICC), 2011 IEEE, pp. 1-4. IEEE, 2011.

[MD96] E. Mirsky and A. DeHon, “MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources,” in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1996, pp. 157–166.

[ME95] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In ACM International Symposium on Field-Programmable Gate Arrays, pages 111–117. ACM Press, 1995. Monterey, California, United States.

[MO98] T. Miyamori and K. Olukotun, “REMARC: reconfigurable multimedia array coprocessor,” in FPGA ’98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays. New York, NY, USA: ACM, 1998, p. 261.

[MVV+02] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A Retargetable Compiler for Coarse-Rrained Reconfigurable Architectures. In IEEE International Conference on Field-Programmable Technology, pages 166–173, 2002.

[MVV+03] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In International Workshop on Field-Programmable Logic and Applications. Springer, 2003.

[PCC+14] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi

Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In Proceeding of the 41st annual international symposium on Computer architecuture (ISCA '14). IEEE Press, Piscataway, NJ, USA, 13-24.

[PH12] Robin Panda, Scott Hauck, "Adding Dataflow-Driven Execution Control to a Coarse-Grained Reconfigurable Array", *International Conference on Field Programmable Logic and Applications*, 2012.

[PFK+06] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. Proc. 2006 Intl. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) Oct. 2006, pp. 136-146.

[PFM+08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08). ACM, New York, NY, USA, 166-176.

[Rau94] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In International Symposium on Microarchitecture, pages 63–74, 1994.

[RG81] B.R. Rau and C.D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", In Proceedings of the Fourteenth Annual Workshop on Microprogramming (MICRO-14), December 1981, pages 183-198.

[RSH15] RajendranRadhika, ShriHari, Aviral Shrivastava, and Mahdi Hamzeh. "Path selection based acceleration of conditionals in CGRAs." In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pp. 121-126. EDA Consortium, 2015.

[SEH03] Akshay Sharma, Carl Ebeling, and Scott Hauck. PipeRoute: A Pipelining-Aware Router for FPGAs. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 68–77, New York, NY, USA, 2003. ACM.

[SEH06] A. Sharma, C. Ebeling, and S. Hauck. PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(3):518 – 532, march 2006.

[SLL+00] H. Singh, M.H. Lee, G. Lu, FJ Kurdahi, N. Bagherzadeh, and EM Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. IEEE Transactions on Computers, 49(5):465–481, 2000.

[SWS+11] Steiner, Neil, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. "Torc: towards an open-source tool flow." In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, pp. 41-44. ACM, 2011.

[Teig12] Steve Teig. Going beyond the FPGA with Spacetime. Retrieved November 5, 2016 from http://www.fpl2012.org/keynote4.shtml. FPL 2012.

[VE10] Brian Van Essen. Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays. PhD thesis, University of Washington, 2010.

[VEWC+09] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus Scheduled Interconnect in Coarse-Grained Reconfigurable Arrays. In International Workshop on Field-Programmable Logic and Applications, pages 268–275. IEEE, 2009.

[WBH+92] N.J. Warter, John W. Bockhaus, Grant E. Haab, K. Supraliminal. Enhanced modulo scheduling for loops with conditional branches. In Proc. of the 25th Ann. Intl. Symp. on Microarchitecture, pp. 170-179, Portland, OR, December 1992.

[Wittig16] Ralph Wittig. Power-Efficient Machine Learning on POWER Systems using FPGA Acceleration. Retrieved March 28, 2017 from https://openpowerfoundation.org/presentations/power-efficient-machine-learning-on-power-systems-using-fpga-acceleration/.

[WKY+12] Aaron Wood, Adam Knight, Benjamin Ylvisaker, Scott Hauck, "Multi-Kernel Floorplanning for Enhanced CGRAs", International Conference on Field Programmable Logic and Applications, 2012.

[WP95] Warter-Perez, N.J.; Partamian, N., "Modulo scheduling with multiple initiation intervals," Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on , vol., no., pp.111,118, 29 Nov-1 Dec 1995. Warter-Perez, N.J.; Partamian, N., "Modulo scheduling with multiple initiation intervals," Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on , vol., no., pp.111,118, 29 Nov-1 Dec 1995.

[YCF+08] B. Ylvisaker, A. Carroll, S. Friedman, B. Van Essen, C. Ebeling, D. Grossman, S. Hauck, "Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators", Technical Report, 2008.

[YEH10] B. Ylvisaker, C. Ebeling, S. Hauck, "Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests", Technical Report, 2010.

[Ylv10] Benjamin Ylvisaker. ”C-Level” Programming of Parallel Coprocessor Accelerators. PhD thesis, University of Washington, 2010.

[YLS+10] Kim, Yongjoo, Jongeun Lee, Aviral Shrivastava, Jonghee Yoon, and Yunheung Paek. "Memory-aware application mapping on coarse-grained reconfigurable

arrays." In International conference on High-Performance Embedded Architectures and Compilers, pp. 171-185. Springer Berlin Heidelberg, 2010.

[YLS+11] Kim, Yongjoo, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. "Memory access optimization in compilation for coarse-grained reconfigurable architectures." ACM Transactions on Design Automation of Electronic Systems (TODAES) 16, no. 4 (2011): 42.

VITA

Aaron Wood attended the University of Washington in Seattle for undergraduate studies earning degrees in Electrical Engineering with college honors and Computer Engineering in 2005. He remained at the university in the Electrical Engineering department for graduate school getting a Master degree in 2007. A move to Virginia for his wife's residency program put further studies on hold while working for USC Information Sciences Institute on custom FPGA CAD tools. Following a subsequent move to Oregon, he resumed his graduate work completing a Doctor of Philosophy degree in 2017.