Offset Pipelined Scheduling for Coarse Grain Reconfigurable Architectures

AARON WOOD, University of Washington
SCOTT HAUCK, University of Washington

Coarse Grain Reconfigurable Arrays (CGRAs) offer improved energy efficiency and performance over conventional architectures. However, the limitations of modulo counter oriented execution on these devices restricts their broader application. This paper introduces the Offset Pipelining execution model and an associated scheduling algorithm to address these limits. The proposed approach broadens the scope of applications that can be efficiently mapped to CGRA architectures while mitigating the challenges of scalability and application development. A pipelined program counter CGRA framework blends the high parallelism of traditional CGRAs with the flexibility of commodity processors. Applications scheduled to take advantage of Offset Pipelining provide an average 1.94X speed up compared to a modulo scheduled implementation for resource limited scenarios. Resource utilization is 0.56X that of modulo scheduling to achieve performance parity. Offset Pipelining offers improved performance and flexibility for CGRAs.

• Computer systems organization → Other architectures → Reconfigurable computing • Computer systems organization → Other architectures → Data flow architectures • Hardware → Integrated Circuits → Reconfigurable logic and FPGAs • Hardware → Electronic design automation → High-level and register-transfer level synthesis → Operations scheduling

Additional Key Words and Phrases: CGRAs, Scheduling

1. INTRODUCTION

Spatial architectures, such as FPGAs and CGRAs, gain much of their computational advantage by spreading a repetitive computation across a large array of functional units. Offering significantly more resources, these devices support much greater parallelism than found in standard multicore processors. With computation and interconnect resources explicitly assigned during compilation via placement and routing techniques, these devices can use cheaper interconnect resources, which can support scaling up to millions of individual processing elements. In contrast, a more flexible interconnect that supports completely arbitrary communication demands would severely limit the scaling of these devices.

However, the parallelism of FPGAs comes at a cost when compared to processors. Due to the static, fixed allocation of functional units, all possible computational paths must be implemented in the system at the same time. If the computation has phases or special-case computations for infrequent events, resources must be allocated for all possible execution paths simultaneously. Therefore, FPGAs tend to be used primarily for regular, repetitive computations, limiting their broader applicability. There have been approaches that time-multiplex FPGAs [DeHon 1994] [Trimberger 1997], and their CGRA relatives [Ebeling 1996] [Mei 2003] [Carroll 2007], which means that a given resource is used to support different operations over time. However, since the assignment to the timeslot on a given resource is generally fixed, the problem remains that during each trip through the time-multiplexed schedule, operations from every possible execution path must be issued. Traversals through multiple potential execution paths can therefore become prohibitively expensive.

Processors generally do not suffer from this problem, primarily because of their branching capabilities. Microprocessor instructions for all possible execution paths only consume instruction memory space, an increasingly inexpensive resource. For the expensive resources, which include the issue slots for instructions, generally only those operations necessary for the current execution sequence are actually issued. A processor can therefore efficiently support a very complex algorithm with many alternative execution paths, yet only consume resources based on the current needs of the algorithm.

As a result of these differences, microprocessors are able to support almost any type of computation, while FPGAs and CGRAs are generally relegated to more repetitive applications. In this work, we propose an alternative which couples the massive parallelism and scalability of spatial architectures with a processor's ability to efficiently support complex control flow via branching. Thus, we seek to significantly increase the range of applications that can take advantage of CGRAs.

In this paper we first discuss modulo-scheduled architectures and modulo scheduling, the basis for most time-multiplexed FPGA and CGRA architectures. This sets the stage to introduce Offset Pipelining, our new computation model. We then present Offset Pipeline Scheduling (OPS), an iterative modulo scheduling (IMS) inspired algorithm that can automatically map a design leveraging the execution model. Finally, we compare OPS and IMS applications, demonstrating the potential of this new approach.

Note that mapping to a time-multiplexed FPGA or CGRA requires more than a scheduling algorithm; scheduled applications must also be placed and routed onto the device. In our companion paper [Wood-Routing] we present placement and routing algorithms for these systems, which provide a complete tool chain leveraging Offset Pipelining.

2. ASSUMPTIONS

A few assumptions are offered in this paper to simplify the discussion. Although a CGRA may contain many types of computation resources, including ALUs, LUTs and memories, we will initially consider a device composed exclusively of ALUs. All operations are further assumed to execute in one cycle. Neither of these assumptions is required, but merely simplify examples to help clarify the concepts introduced.

3. TIME-MULTIPLEXED EXECUTION

In a standard spatial architecture such as an FPGA, a device with N functional units can support a computation of at most N operations since operations are statically assigned to individual functional units, and each functional unit has instruction memory for exactly one operation. However, numerous researchers [DeHon 1994] [Ebeling 1996] [Trimberger 1997] have investigated time-multiplexing, where functional units are provided larger instruction memories and the device cycles through a set of configurations. In these systems, all functional units simultaneously execute the local instruction stored at location 0, then, in the next cycle, execute instruction 1 and so on. Thus, in order to map a computation of N operations onto a time-multiplexed device with F functional units, each unit will have a program length of at least $\lceil N/F \rceil$ with each unit executing instructions 0, 1, … $\lceil N/F \rceil$-1 and then restarting at 0. Since the allocation of operations to functional units and time slots is static, the computation is predictable and can be supported by a scalable, efficient interconnect network. The control of this system is also simple; the "program counter" for such a device is a modulo counter, which increments in a cyclic fashion through the program. The program length is referred to as the *Initiation Interval* or *II* for the application.

A simple way to map an application onto a modulo-counter based architecture is to handle all of the operations for a given iteration of the application within one II before moving on to the next iteration of the algorithm. Thus, for the code in Figure 1, we might produce the program shown in Figure 2 for a device with two ALUs.

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

Figure 1. Example of easily pipelined code.

Cycle	ALU0	ALU1
0	read a	read b
1	+	
2	>>	
3	write c	

Figure 2. Naïve schedule for the program in Figure 1.

Unfortunately, this schedule makes poor use of the available resources and cannot take advantage of 3 or more ALUs because of the data dependencies between different operations. An improved mapping overlaps the operations of successive iterations of the loop as shown in the execution trace in Figure 3.

Cycle	ALU0	ALU1	ALU2
0	read a	read b	
1	+		
2	**read a**	**read b**	>>
3	**+**		write c
4	read a	read b	**>>**
5	+		**write c**

Figure 3. Execution trace of improved schedule.

Here we start the execution of iteration 0 in cycle 0 and iteration 1 in cycle 2, even though iteration 0 has not yet finished. This overlapping of successive iterations is legal because we make sure not to reverse the order of instructions that have data or control dependencies. Dependencies occur within a given iteration and between iterations. For example, the addition operation in the example must follow the stream reads of its operands within the iteration. An instance of an inter-iteration constraint is to require the write operation of iteration I to precede the write operation of iteration I+1 to prevent data corruption.

Cycle	ALU0	ALU1	ALU2
0	read a	read b	>>
1	+		write c

Figure 4. Improved schedule.

To achieve the execution trace of Figure 3, an II of 2 is needed, corresponding to the spacing of iteration starts. This program is shown in Figure 4. In this case, the operations on ALU2 belong to the previous iteration relative to ALU0 and ALU1. Scheduling that interleaves iterations in this way is called modulo scheduling.

4. MODULO SCHEDULING

The basis of many modulo schedulers is the Iterative Modulo Scheduling algorithm (IMS) [Rau 1994]. The goal is to extract the most parallelism from the loop to keep the available ALU resources busy. The first step is to find a lower bound on the possible II. Each instruction requires an available issue slot, so for F functional units and N operations, the II must be at least ⌈N/F⌉. This is known as the resource constrained II, or ResII. We must also consider inter-iteration dependencies. Imagine there were 4 reads from the same input stream in a loop; we clearly need an II of at least 4, since only one read can happen each clock cycle. This limit is the recurrence constrained II, or RecII, and is found by looking for cycles in the dependence graph. The IMS algorithm begins with an II set to the maximum of the initial lower bounds of ResII and RecII, though it may be increased when needed during scheduling.

The primary data structure for scheduling is the modulo reservation table (MRT). For the previous example, this can be visualized as Figure 4. 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 arrangement of the operations at the given II. The loop selects operations from a priority queue, prioritized by their height in the dataflow graph. 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 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 satisfies all data dependency and resource constraints, creating an execution pattern that can overlap successive iterations of the target loop body.

Modulo scheduling provides a significant benefit by maximizing resource utilization of the hardware. However, code is rarely as simple as the example above. The next section covers a technique to manage more complex control flow in the target code.

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

Figure 5. Code example requiring predication.

4.1 Conditional Code with Predication

While an inner loop without complex control flow is preferable for modulo scheduling, most applications include conditional code. If the earlier example is modified as shown in Figure 5, the modulo schedule must now accommodate the conditional statements.

In order to fit all of the target code into the schedule, both execution paths must be allocated issue slots since either is possible. At run time, both results for c are generated and the *greater than* comparison produces a predicate, which is used to control a phi node that determines the value to store. This technique is called predication and eliminates the conditional statements. The predicated version of Figure 5 is shown in Figure 6. The predicate value p controls the ternary operator producing c3.

```
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);
}
```

Cycle	ALU0	ALU1	ALU2
0	read a	read b	p?c1:c2
1	a > b	a + b	write c3
2	c0 >> 1	a - b	

Figure 6. Predicated version of Figure 5 (left) and resulting modulo schedule (right).

While predication works well in situations when the amount of code being predicated is small, large blocks of predicated operations incur significant overhead costs. Two such scenarios are shown in Figure 7, exhibiting the limits of predication.

```
kmeans {
   while (not converged) {
      foreach item
            match to best class
      foreach class
            compute best centroid
}
```

```
PET {
   while (1) {
      get incoming data.
      compute statistics
      if (event detected)
         complex computation
}
```

Figure 7. Algorithm sketches for K-means and PET event detection.

- **Multiple Phases**: Many algorithms go through multiple phases; when one phase is active, the rest are inactive. For example, a K-means clustering first assigns values to a cluster and then updates the cluster centers, while a discrete wavelet transform or a 2D DCT is often defined by multiple passes across the data. In each of these cases, predication requires that issue slots be reserved for all phases simultaneously, even though only one phase is actually active. Note that for some cases, very deep pipelining may address this, but not if there are cyclic dependencies between phases such as K-Means Clustering.
- **Exception Code**: Some algorithms have a simple common case and a more complex operation for unusual circumstances. For example, a PET event detector normally does simple data movements, but in a small, unpredictable fraction of circumstances, it performs much more complex processing [Haselman 2008].

In each of these scenarios, IMS with predication wastes a large number of issue slots on instructions that are not needed every iteration. This either significantly boosts the II, slowing the computation, or significantly increases the number of resources needed. A microprocessor-based system, which can use branching and other complex control flow, avoids much of this waste by fetching only those operations that are actually needed at that time. In the rest of this paper, we will present an alternative execution model that retains most of the benefits of modulo counter architectures, but avoids the issue slot waste of modulo scheduling for complex applications. We also present a scheduling algorithm that exploits the opportunities of this new execution model.

5. OFFSET PIPELINING EXECUTION MODEL

Our new execution model, called Offset Pipelining, considers a computation divided into subsets of the target code called modes and then organizes the operations of the modes to efficiently run on a CGRA architecture. A mode is composed of one or more basic blocks. An algorithm implementation becomes the execution of a series of modes over time while also interleaving different mode iterations in a manner similar to modulo scheduling.

The three colored basic blocks in Figure 8 correspond to modes for this example. For IMS, the inner loop would need to be flattened (Figure 9). The if statement conditions would then be converted to predicates to provide the necessary control.

Although this example is quite small, it illustrates the mutually exclusive execution of the modes. An example modulo schedule that shares the same mode transition structure is shown in Figure 10 on the left. The schedule includes all of the operations for the complete application.

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

Figure 8. Code example motivating multi-mode application support.

```
mode0 = 1; mode1 = 0; mode2 = 0;
while (1) {
  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 9. Flattened code in preparation for modulo scheduling.

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 IMS the instructions for an iteration in row 1 may be intended to execute at time 1, 1+II, 1+2*II, etc.

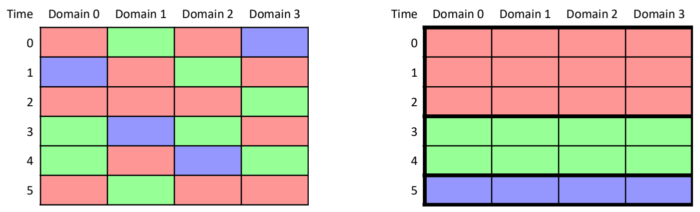

Figure 10. Example modulo schedule after flattening and IMS scheduling (left) and modulo schedule organized by mode (right).

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 in Figure 10. 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. Thus, for the example in Figure 8 the data read is 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 11. This skewing allows the system to support modes with deep sequences of instructions. 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 12), inspiring the name of the execution style.

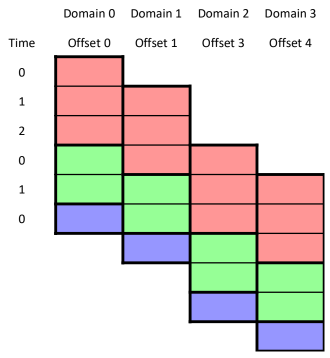

Figure 11. Staggering resource start times.

From an execution perspective, the lead ALU 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 13 shows an example execution trace. Instead of an II of 6 for the modulo schedule in Figure 10, the Offset Pipelined approach has an effective II less than 3, depending on the relative frequency of mode initiations.

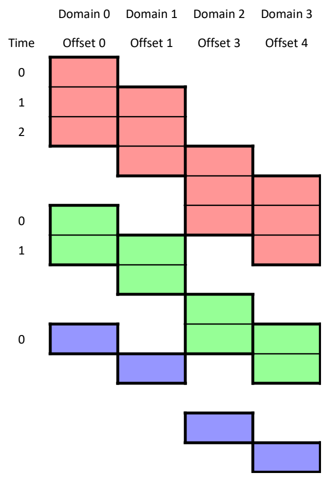

Figure 12. Independent mode schedules with shared staggering between resources.

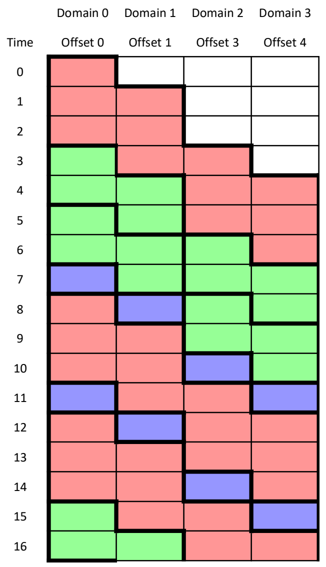

Figure 13. Offset Pipelining execution trace of Figure 12 schedule.

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.

A mapping is composed of a lead domain, which determines the series of modes to execute, while the rest of the domains are follower domains. 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 for 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 + II_M - 1 on that iteration. Each follower domain I, with offset O_I, will spend cycles T+O_I … T + O_I + II_M - 1 on that iteration, as shown in Figure 13. 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.

5.1 Phi Node Elimination

Offset Pipelining provides a further benefit for optimizing application execution. Consider the for loop in Figure 14 (left). For a standard modulo counter based architecture, the loop execution would be predicated as shown (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 (Figure 15 top).

In contrast, Offset Pipelining can eliminate phi nodes with the invocation of modes implicitly performing the phi node functions. For example, the OPS version of the same example is shown in Figure 15 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 14. Phi node example for inner loop predication (left) and predicated version (right).

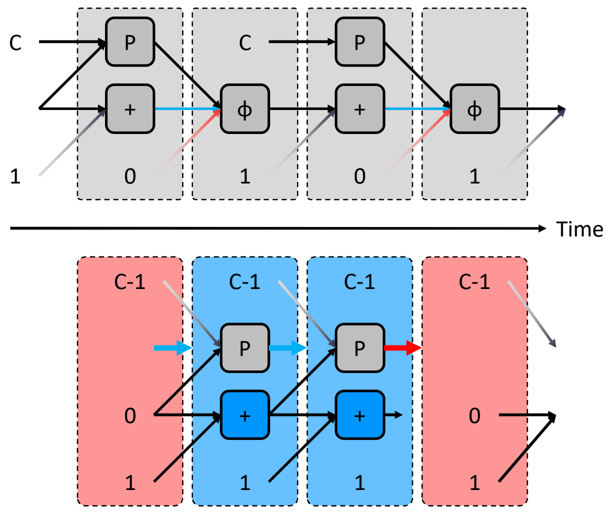

Figure 15. Phi nodes in modulo schedule (top) eliminated with Offset Pipelining (bottom).

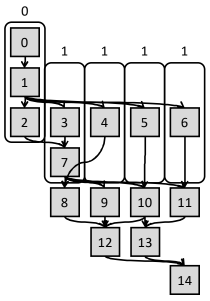

Figure 16. Initial scheduling.

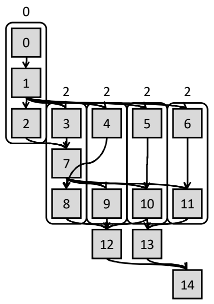

Figure 17. Front end shaping.

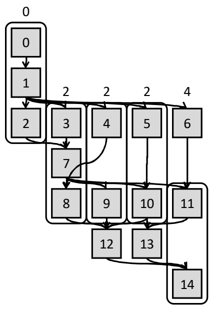

Figure 18. Back end shaping.

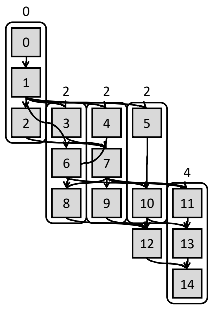

Figure 19. Rescheduling.

6. SCHEDULING EXAMPLE

The next sections present a novel algorithm for a complete scheduling for Offset Pipelining designs and an experimental validation of the approach. To help understand the algorithm, consider manually scheduling a single mode application (Figure 16). The grey boxes are operations with dataflow constraints depicted by arrows. The application has a minII of 3 on a 5-ALU architecture. ALUs are shown as rectangles to indicate their issue slots, with the offset listed above. The initial offset assignment is shown in Figure 16, and is set to architecture-defined minimums.

We employ as-soon-as-possible (ASAP) scheduling, 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 16, consider the earliest instructions in the application. Given 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 the earliest slots, that domain can be shifted later without degrading the schedule quality. This is a process we call front-end shaping. Figure 17 shows the result of front-end shaping, with the offsets now set to 0, 2, 2, 2, 2.

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. 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 18. This step is called back-end shaping. After front end and back end shaping, the application is rescheduled in Figure 19 which moves operation 6 a cycle later.

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 19. In such a case, we do offset exploration, which involves adding 1 to each domain 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 20. The 0, 2, 2, 3, 4 case provides a complete schedule for the application. In the sections that follow, we take this intuitive example of scheduling and transform it into a complete scheduling algorithm for applying Offset Pipelining.

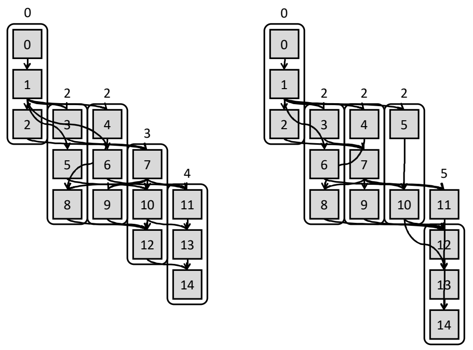

Figure 20. Offset Exploration with 0, 2, 2, 3, 4 and 0, 2, 2, 2, 5.

7. OFFSET PIPELINED SCHEDULING ALGORITHM

In the sections that follow, we present the details of the new Offset Pipelined Scheduling algorithm. The algorithm consists primarily of an outer loop that sets per mode IIs and an inner loop that uses ASAP 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.

7.1 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 [Rau 1994] 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. Issue slot times are measured relative to the first issue slot in the lead domain with offset 0. There are separate issue slots offered by a resource for each mode.

An example ORT is shown in Figure 21. 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. 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.

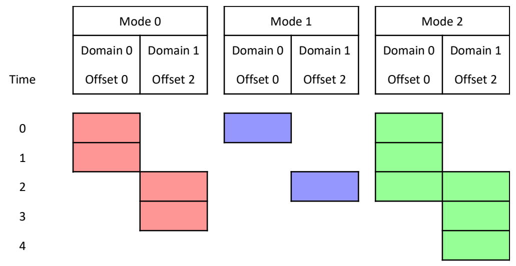

Figure 21. Example Offset Reservation Table.

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.

7.2 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 22. 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 implementation.

```
initializeIIs()
do {
   initializeOffsets()
   resetLooseSchedulingCache()
   do {
      buildORT()
      if ASAPschedule(tight)
         return SUCCESS
      offsetsUpdated = offsetAdjustment()
   } while (offsetsUpdated)
   iisUpdated = incrementIIs()
} while (iisUpdated)
```
Figure 22. Top level OPS algorithm.

7.2.1 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 23 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 its predecessors.

This function can 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 at times because a subsequent increase in a domain offset may move unused issue slots later in order to handle the loosely scheduled operations. Loose scheduling primarily provides feedback during the offset adjustment phase.

The inner loop body attempts a tight scheduling (Figure 22 line 7). When successful, this terminates the algorithm with a completed schedule. Failure leads to offset adjustment.

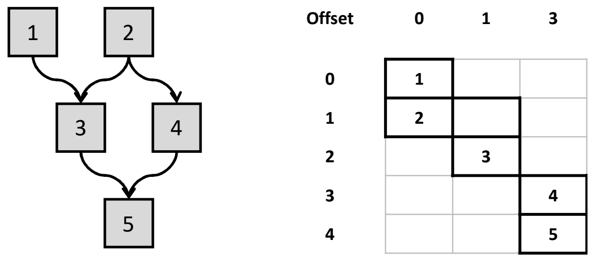

Figure 23. As-soon-as-possible operation scheduling.

7.2.1.1 Operation Prioritization and Delay Calculation

Operation ordering for OPS is accomplished with a height based priority scheme similar to IMS [Rau 1994]. The difference involves the multi-mode nature of the target netlists. For loop carried nets returning to the same mode, the calculation is the same as that of IMS; the II of the mode in question is multiplied by the iteration distance (Equation 1). For loop carried nets that transit mode boundaries, the distance is calculated using the sum of mode IIs of the intervening iterations between source and sink, including the source mode, but excluding the sink mode. This accommodates the minimum distance between mode iteration initiations of the source and sink operations.

$$\texttt{Delay = SourceOpDelay - (II * IterationDistance)} \tag{1}$$

$$\texttt{Delay = SourceOpDelay - SumShortestPathIIs} \tag{2}$$

The delay calculation is likewise used throughout OPS to legally schedule operations within and between modes. The delay between two operations, as measured between their inputs, is a function of source operation parameters and parameters of the net connecting the two operations. This delay is described by Equation 2 with parameters detailed in Table 1 for nets that remain within one iteration or connect to the subsequent iteration.

Table 1. 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 given operation, the calculated delay is added to the time slot of each operation driving that operation. The maximum delay value among all inputs to the new operation determines the earliest time the operation may be scheduled in order to ensure all inputs are available.

```
 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 24. Offset adjustment pseudo code

7.2.2 Offset Adjustment

With a scheduling mechanism in hand, the next issue to address is setting the domain offsets. The concept of offset adjustment is to judiciously increase offsets to provide issue slots for operations without legal issue slots in the current offset assignment. The offset adjustment process is done in two phases, illustrated in Figure 24. 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.

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.

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.

7.2.2.1 Front end shaping

The shapeOffsets function (Figure 24 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 issue 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, the 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 17.

7.2.2.2 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 is shown in Figure 18: 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 later. Operation 11 and those preceding can be addressed by the other domains without further offset adjustment.

7.2.2.3 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 adjustment to the offsets, an offset exploration routine takes over. The possible candidates are generated by incrementing each offset individually, ignoring duplicate offset configurations (e.g. if there are 3 domains with offset 6, we 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 heuristically select the candidate that incremented the lowest offset.

7.2.3 II Adjustment

OPS resorts to II adjustment if 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. 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.

7.2.3.1 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. These annotations might also be set manually by the developer. A higher priority corresponds to a more frequently executed mode. It is desirable to minimize the II 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 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.

7.2.3.2 II Initialization

The initial IIs for OPS are calculated in a manner similar to IMS and the computation begins 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 the 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.

7.2.4 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.

7.3 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. 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.

7.4 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. 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 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. In the event of a tie, the offset value that occurs least frequently in the collection of offset assignments is selected. The example in Figure 25 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.

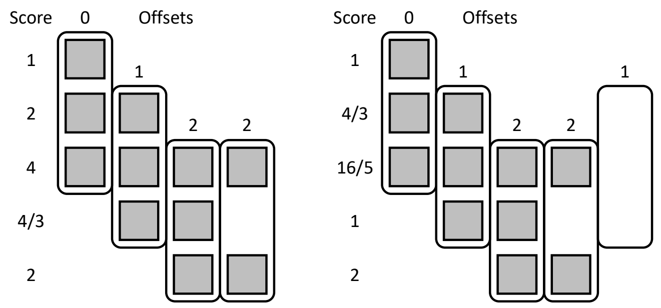

Figure 25. Spare domain offset assignment.

7.5 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.

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 26 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. The second case on the right can be resolved by offset adjustment to provide a domain that covers both operations in the schedule.

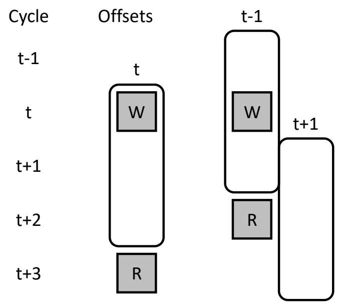

Figure 26. Memory operation scheduling cases.

Scheduling stream operations uses the same checks for cases where a stream is both written to and read from within the target application. 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.

8. 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 [Van Essen 2010]. A summary of resources available in each domain for this work is shown in Table 2. Resources within a domain are connected to a crossbar for communication, while between domains signals traverse an island style registered interconnect structure.

Table 2. Domain resource composition for OPS vs IMS evaluation.

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

8.1 Benchmarks

The benchmark applications used in this evaluation are summarized in Table 3. 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 27 shows results for the five benchmarks when scheduled onto four different device sizes.

Table 3. 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

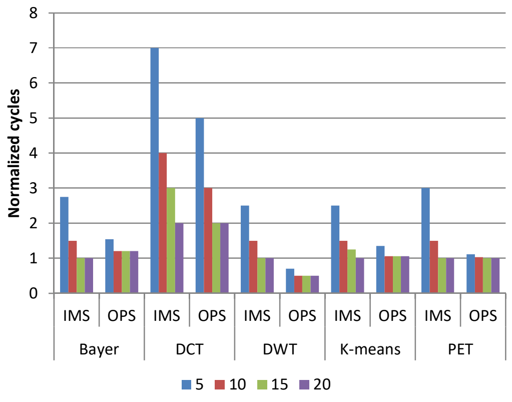

Figure 27. OPS vs IMS when provided with four different device sizes.

The IMS implementation reaches the recurrence limit for all applications except the DCT, which does not reach the recurrence limit until 34 domains are allocated. The OPS Bayer implementation suffers some overhead associated with mode transitions, so cannot match IMS on larger arrays. On the other hand, the DWT implementation outperforms 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, all applications show significant improvement over IMS due to issuing unused operations from inactive modes on every iteration, making targeting more than a single loop body much less efficient. 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.

8.2 Scheduling Behavior

To help demonstrate how the OPS algorithm executes, consider an 8-mode DWT algorithm. Figure 28 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 reasonable given the target architecture supports 64 instructions per domain.

OPS runtime is fast, not exceeding approximately 10 seconds for any of the schedules generated in this evaluation. This is negligible compared to the runtimes of the placement and routing portions of an FPGA or CGRA the tool chain.

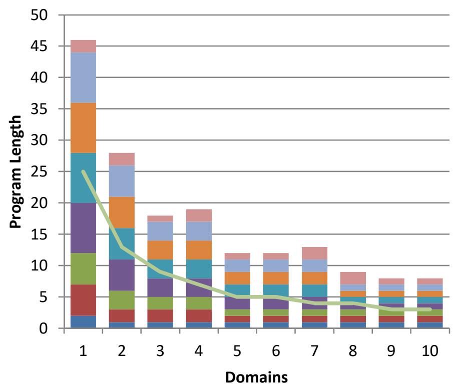

Figure 28. DWT mode II progression. Stacked bars are OPS mode IIs. The line is IMS II.

8.3 Results

Figure 29 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 detailed view of the DWT and PET results in Figure 30 shows the twofold advantage of OPS. The same performance can be achieved with fewer resources or better performance can be achieved with the same number of resources when operating in a resource limited area of the curve.

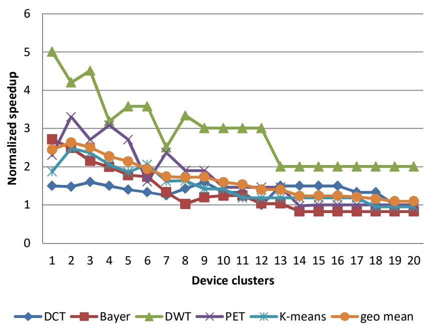

Figure 29. OPS vs IMS performance summary.

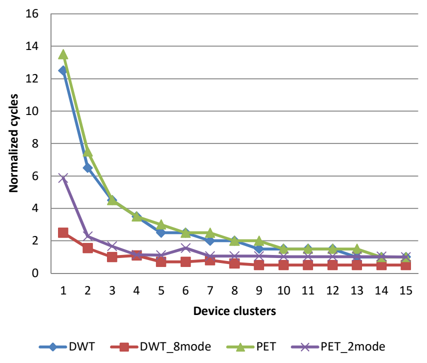

Figure 30. Detailed results for DWT and PET.

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.

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 considered.

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.

9. RELATED WORK

The related work presented here draws comparisons to other architectures and tools to position Offset Pipelining and the associated scheduling algorithm in the broader context of reconfigurable computing research.

Related architectures primarily fall into two categories; massively parallel processor arrays (MPPAs) and CGRAs. MPPAs such as Ambric [Butts 2008] and Tilera [Taylor 2002] are composed of discrete processors. They are less tightly integrated than the proposed approach and must be programmed using traditional parallel programming techniques. CGRAs such as Mosaic [Carroll 2007] and ADRES [Mei 2003] are designed for modulo scheduled execution. These architectures are tightly integrated and most offer tool support [Friedman 2009][Mei 2002] to leverage the array, but are limited to modulo counters for control.

The Tabula [Teig 2012] SpaceTime architecture is a commercial product similar to a CGRA using a modulo counter mechanism for time multiplexing. However, these devices are fine grained and provide a conventional FPGA tool chain abstraction for the underlying hardware.

Algorithms for mapping applications to CGRAs have been based on compiler tools for VLIW and FPGA architectures. Scheduling specifically draws from modulo scheduling work treating the device as a large VLIW style processor. Modulo scheduling and IMS [Rau 1994] in particular inspire the software pipelined schedule and iterative nature of OPS. Modulo scheduling with multiple initiation intervals [Warter-Perez 1995] explores more flexible execution similar to OPS. This earlier work targets a more traditional VLIW machine with a single program counter, very different from the goal of OPS to help automate mapping to multiple control domains on an enhanced CGRA.

Some FPGA architecture support partial reconfiguration, allowing their configurations to change at runtime. While this can be used to support the type of modal application proposed here, it is significantly different due to the relative amount of time it takes to reconfigure a portion of the FPGA and the configuration data that must be sent to the device each time partial reconfiguration occurs.

10. CONCLUSIONS

When considering word-oriented FPGAs and FPGA-like systems, architectures have split into MPPA and CGRA style devices. CGRAs provide a huge amount of parallelism, and have automatic mapping tools that can spread a computation across a large fabric, but their restriction to modulo-counter style control significantly limits their ability to support applications with more complex control flow. MPPAs provide an array of full-fledged processors, with a great deal of fine-grained parallelism, but they are much less tightly coupled than CGRAs and generally must be programmed via explicitly parallel programming techniques.

In this paper we have taken a step towards merging these two styles of devices. By providing a new program counter model, that keeps communication local yet can support more complex looping styles, we can support a much richer set of applications. Essentially, this integration of the conditional branch and complex control flow operations significantly increases the computational density, and range of target applications, supportable by these systems.

We have demonstrated a complete scheduling algorithm for these devices. The tool automatically schedules issue slots, determines individual domain offsets, and sets mode IIs to achieve a high-performance and dense implementation. Compared to IMS, OPS solutions provide 1.94X better throughput or use 0.56X area over the benchmark suite.

REFERENCES

Michael Butts, Anthony Mark Jones, and Paul Wasson. 2007. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines.

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

Andre DeHon. 1994. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In Duncan A. Buell and Kenneth L. Pocek, editors, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. 31-39.

Carl Ebeling, Darren C. Cronquist, and Paul Franklin. 1996. RaPiD – Reconfigurable Pipelined Datapath. In International Workshop on Field Programmable Logic and Applications, Springer Berlin Heidelberg, 126-135.

Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: An Architecture-Adaptive CGRA Mapping Tool. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* ACM, 191-200.

Michael Haselman, Robert Miyaoka, Thomas K. Lewellen, Scott Hauck, Wendy McDougald, and Don DeWitt. 2009. FPGA-Based Front-End Electronics for Positron Emission Tomography. In ACM/SIGDA Symposium on Field-Programmable Gate Arrays, ACM, 93-102.

Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In International Workshop on Field-Programmable Logic and Applications. Springer, 61-70.

Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2002. DRESC: A Retargetable Compiler for Coarse-grained Reconfigurable Architectures. In IEEE International Conference on Field-Programmable Technology, IEEE, 166–173.

B. Ramakrishna Rau. 1994. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In Conference on Computing Frontiers, ACM, 63-74.

Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen Matt Frank, Saman Amarasinghe and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In IEEE Micro, IEEE, 25-35.

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

Steve Trimberger, Khue Duong, and Bob Conn. 1997. Architecture Issues and Solutions for a High-Capacity FPGA. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM, 3-9.

Brian Van Essen. 2010. Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays. Ph.D. Thesis, University of Washington, Dept. of CSE.

Nancy J. Warter-Perez, Noubar Partamian. 1995. Modulo Scheduling with Multiple Initiation Intervals. In Proceedings of the 28th Annual International Symposium on Microarchitecture, IEEE, 111-118.

Michael J. Wirthlin, Brad L. Hutchings. 1996. Sequencing Run-time Reconfigured Hardware with Software. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 122-128.

Aaron Wood and Scott Hauck. EveryTime Routing for Offset Pipelined Coarse Grain Reconfigurable Architectures. Submitted to ACM Transactions on Reconfigurable Technology and Systems.