Offset Pipelined Scheduling: Conditional Branching for CGRAs

Aaron Wood
Department of Electrical Engineering
University of Washington
Seattle, WA USA
arw82@uw.edu

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA USA
hauck@uw.edu

Abstract—Coarse Grained Reconfigurable Arrays (CGRAs) offer improved energy efficiency and performance over conventional architectures. However, the modulo counter based control of these devices makes them inefficient for applications with multiple execution modes. This work presents a new type of architecture that adds support for branching control flow to CGRAs. The pipelined program counter CGRA framework blends the high parallelism of traditional CGRAs with the flexibility of commodity processors. Offset Pipelined Scheduling (OPS) is the basis of an enhanced CGRA tool chain targeting these devices. OPS is shown to provide an average 1.94x speed up for benchmarks that are resource limited when modulo scheduled.

Keywords- CGRA, scheduling, software pipelining

I. INTRODUCTION

Coarse grained reconfigurable arrays (CGRAs) offer improved energy efficiency and performance compared to commodity architectures. When well utilized, they provide high peak performance through massive parallelism. They embrace features of standard FPGAs while moving away from bit oriented resources towards larger logic blocks and correspondingly wider interconnect. Such architectures are register rich using pipelined interconnect and time multiplexed resources to maximize potential utilization. CGRAs are capable of greater parallelism than commodity processors and have lower area and performance overheads compared to FPGAs.

The time multiplexed control of a CGRA is managed by a global modulo counter. Each counter value selects a configuration of the logic and routing resources on the device to execute in a cyclic sequence. A pseudo code example and a corresponding schedule table are shown in Fig. 1. For the purposes of this discussion, the operations simply depend on the previously numbered operations within the same basic block. A modulo schedule is well matched to a single loop body.

Managing more complex control flow becomes inefficient. An application consisting of a sequence of loop bodies will have wasted issue slots because a given iteration will be executing only one of the loop bodies (Fig. 2). In this example, not even half the issue slots are doing useful work on any given iteration. A better execution style for this example would allow each mode of the application to execute as though in isolation.

The ability to transition from one mode of execution to another suggests a branching capability comparable to that of a conventional processor. However, filling and draining the execution pipeline can be costly for applications with high latency or relatively few iterations between mode transitions. Moreover, the ability to branch brings the additional difficulty of broadcasting the change in control flow across the device.

In this paper we introduce Offset Pipelined Scheduling (OPS), which provides a solution to these problems. By staggering the execution of resources across the device it provides the time needed to send changes in control flow. In OPS, a single control unit acts as a leader, performing operations from a given mode as well as computing loop or branch conditions to determine the next mode to initiate. Other units act as followers of this leader, receiving the same program counter value from the leader, only delayed by an offset amount. In this way, the multiple resources in the system 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 units execute their own portions of each of these modes, in the same order, delayed by a fixed offset from their neighboring

for {
  op1
  op2
  op3
  op4
}

Figure 1. Simple modulo schedule

<table>
<thead>
<tr>
<th></th>
<th>Mode 0</th>
<th>Mode 1</th>
<th>Mode 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU0</td>
<td>ALU0</td>
<td>ALU0</td>
<td>ALU0</td>
</tr>
<tr>
<td>ALU1</td>
<td>ALU1</td>
<td>ALU1</td>
<td>ALU1</td>
</tr>
<tr>
<td>op1</td>
<td>op5</td>
<td>op1</td>
<td>op5</td>
</tr>
<tr>
<td>op2</td>
<td>op6</td>
<td>op2</td>
<td>op6</td>
</tr>
<tr>
<td>op3</td>
<td>op7</td>
<td>op3</td>
<td>op7</td>
</tr>
<tr>
<td>op4</td>
<td>op8</td>
<td>op4</td>
<td>op8</td>
</tr>
<tr>
<td>op5</td>
<td>op9</td>
<td>op9</td>
<td>op9</td>
</tr>
</tbody>
</table>

Figure 2. Modulo scheduled sequenced loop bodies
units. In this way, the fill and drain periods of subsequent modes are completely overlapped. The example from Fig. 2 is shown with OPS style execution in Fig. 3. Transitions between modes begin at cycles 4, 5, and 11, with ALU1 trailing by two cycles. Each mode effectively operates with its own initiation interval (II), eliminating most wasted issue slots. The necessary hardware support provides pipelined program counters to manage the execution of the offset control domains.

This paper describes the hardware architecture and scheduling algorithm for mapping multi-mode applications. Then, a discussion of the OPS algorithm is followed by a brief description of benchmark creation and the philosophy of effectively decomposing applications into modes. Finally, the evaluation demonstrates the capabilities of OPS using several signal processing benchmarks.

II. EXECUTION ON BRANCHING CGRAS

The following subsections detail features of offset pipelined schedules and the target enhanced CGRA architectures. The Fig. 3 example is used to discuss modes, offsets and the relationship to pipelined program counter architectures.

A. Expanding the Scope of CGRA Code

Software pipelining by modulo scheduling was originally envisioned for VLIW machines where functional units must be statically scheduled to provide maximum utilization. Modulo scheduling for CGRAs is a logical extension with modulo counter control and a focus on inner loop parallelism. In this paper we broaden the scope of application code mapped to CGRAs, including allowing multiple loops to coexist on the same device.

B. Modes

A mode is a partition of the target code that will be run exclusively until a transition to another mode. It is a subset of the code mapped to the CGRA. Consider two successive loop bodies which can be logically divided into two separate modes and executed in isolation. Such a distinction matters little in the context of a conventional processor since branching to move between blocks of code 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. Particular care must be taken to manage filling and draining the pipeline when transitioning between modes. In a conventional CGRA, control logic must execute by predating operations with side effects across the entire target code body.

K-means clustering provides an example of modal behavior in an application. The algorithm logically breaks into two modes; assigning observations to clusters and updating cluster centroids. A conventional CGRA must execute both modes on every iteration or face reconfiguring the device at each transition with either option generating a sizeable overhead.

C. Pipelined Program Counter Architecture

The key feature of a pipelined program counter architecture is the ability for a processing domain to receive its instruction pointer from a neighboring domain in the array. Each program counter controls a small domain of resources, including ALUs, LUTs, memories and interconnect. An application mapped by OPS contains a tree arrangement of domains each receiving an instruction pointer from its parent. A domain can be thought of as a small VLIW machine controlled by a single program counter managing specific resources. OPS forges the domains into a unit to efficiently execute the target code.

While OPS relies on the unique feature of pipelined program counters, this architecture still supports modulo scheduled applications by configuring each domain to execute as a modulo counter. A pipelined program counter CGRA is generally compatible with prior work using modulo schedules.

D. Offsets

The root of the tree of domains is the leader and issues all control flow decisions. Control information then cascades through the domains. The lead domain is defined to have an offset of 0. Other domains have offsets measured relative to the lead. In Fig. 3, ALU0 is the sole resource of the lead domain, while the domain of ALU1 has an offset of two. The offset is simply 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 is beyond the scope of this paper. 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 must be the same across all modes.

An application may call for multiple domains at offset 0. In this case, control logic must be duplicated for each domain that will have offset 0 in order to keep these domains synchronized appropriately. This issue will be addressed in subsequent work on placement and routing.
CGRA mitigates the cost of prologue and epilogue specific code by effectively merging it with the steady state loop behavior. This effect can be seen in Fig 3. Each domain executes II operations for the current mode and then either repeats the mode or branches to another. Cycles 7 and 8 represent the steady state for one mode. Cycles 5 and 6 contain the prologue while cycles 11 and 12 the epilogue; in each case, the domains continue to merely execute II operations from their current mode. The overall program length is the sum of the IIs of all modes. In contrast, modulo scheduled CGRAs would require significant extra control logic to represent this prologue and epilogue behavior.

III. OFFSET PIPELINED SCHEDULING

The introduction to the OPS algorithm begins with a brief discussion of the main data structure. A top level outline of the process to explain the overall flow follows. The subsequent sections fill in details of the implementation.

A. Offset Reservation Table

The offset reservation table (ORT) is analogous to the modulo reservation table (MRT) used in iterative modulo scheduling (IMS) [1]. It is constructed using the collection of domain offsets, per mode II information and the composition of resources in each control domain. Each logic element in the device provides II issue slots starting at the domain offset to which it belongs. There are separate issue slots for each mode. An example ORT in Fig. 4 corresponds to the previously discussed execution trace in Fig. 3. Note that the relative positions of operations in the ORT correspond to the positions shown in the execution trace.

Traditional modulo scheduling fits all operations into II cycles by placing operations into time slots modulo the maximum schedule length of II cycles. In OPS we instead set domain offsets late enough to provide issue slots at the necessary times for the operations in the dataflow graph. OPS is best suited to an architecture with many relatively small control domains instead of a few large ones as it relies on this flexibility to map the target application.

B. Top Level Process

OPS scheduling consists of an inner loop that schedules operations and adjusts domain offsets to attempt to map the application and an outer loop that increases individual mode IIs if the inner loop cannot find a legal schedule. The scheduling itself is based largely on IMS. A pseudo code representation of OPS is shown in Fig. 5.

The core function is an as-soon-as-possible scheduling pass that attempts to schedule the target netlist into the current IIs and offsets. 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. 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. We perform loose scheduling at times because a subsequent increase in a domain offset may move unused issue slots later in order to handle the loosely scheduled operations. Within a given set of mode IIs, the algorithm only increases domain offsets, never decreases them.

The inner loop body attempts a tight scheduling (line 7 Fig. 5). When successful, this terminates the algorithm with a completed schedule. Failure leads to offset adjustment. The concept of offset adjustment is to pull offsets later to cover operations that must eventually have a legal issue slot but do not with the current offset assignment. If offset adjustment can make no further headway, the inner loop terminates and IIs are incremented.

C. Operation Prioritization

OPS uses a height based priority scheme very similar to IMS [1]. 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; the II of the mode in question is used to calculate the distance. Loop carried nets that transit mode boundaries calculate distance using the mode II of the source operation. The delay calculation is likewise used throughout OPS to legally schedule operations within and between modes.

D. Offset Adjustment

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

Figure 5. Top level OPS algorithm

```java
1 initializeIIs()
2 do {
3   initializeOffsets()
4   resetLooseSchedulingCache()
5   do {
6     buildORT()
7     if ASAPschedule(loose)
8       return SUCCESS
9     offsetsChanged = true
10    } while (offsetsChanged)
11    return offsetsChanged
```

Figure 6. Offset adjustment
The main driver of progress for OPS is adjusting offsets to provide a better fit for the application (Fig. 6). The modification starts with a loop that performs loose scheduling (line 3) and then adjusts offsets until they can no longer be deterministically pushed to cover later slots based on the needs of the target netlist. If offsets were changed during the shaping, the function returns true and the new offsets are tried in a tight scheduling. When the offsets cannot be updated using the shaping function, a heuristic approach is employed to try a set of offset candidates and select the most promising one to continue attempting a tight scheduling.

The shapeOffsets function (line 4 Fig. 6) is broken into two phases, front end shaping and back end shaping. Front end shaping takes the loose scheduling result and attempts to increase the offsets, in a way that is guaranteed to be conservative. Offsets will never increase too much, but after front end shaping, some offsets may still be too small. Initially operations have a time slot, but no assigned domain. The algorithm searches for the earliest unassigned operation in any mode, and sets the offset of a free domain to the schedule position of that operation. We then fill that domain's remaining issue slots across all modes with as many operations as possible, at the time slot determined in the loose schedule. After this, if any free operations and domains remain, the iteration continues with the new earliest unassigned operation. The example in Fig. 7 is a front end shaping for a single mode and increments two offsets.

Back end shaping also performs offset increases, but focuses on the operations scheduled latest rather than the earliest. All domains are marked as unadjusted, with their current offset settings from previous adjustment steps, and all operations are marked as unassigned. We iteratively take the domain with the largest offset and, if necessary, move its offset late enough so that it has an issue slot at the scheduled time of the latest unassigned operation. Then, for each mode M, we assign the I_{M} latest unassigned operations to this domain. This algorithm iterates until we run out of domains or unassigned operations. The example in Fig. 8 moves an offset to 2 in order to cover the latest operation. This move also covers the second operation at the same time slot and will force the uncovered operation to be scheduled later on a subsequent tight scheduling attempt.

Note that back end shaping is quite conservative. For example, consider a single-mode algorithm with an II of 2 and with 2 unassigned operations that both are scheduled for time t, and all other operations have an earlier time slot. Our algorithm will assign a domain to offset t-I, which provides an issue slot at times t-I and t. Both of the two unassigned operations are assigned to this single domain, even though neither is ready to issue at time t-I. Recall that this is as soon as possible scheduling. We make this choice because the optimal answer is unclear. If the domain is given an offset of t, it would force one operation to be moved later, or if we have two domains with offset of t-I, each operation will have a valid issue slot. We choose to take the conservative approach and allow subsequent exploration steps to resolve the ambiguity.

Front end and back end shaping are provably conservative. When this deterministic shaping fails to adjust the offsets, an offset exploration routine takes over. The possible candidates are generated by incrementing each offset; duplicates are ignored. Each candidate offset configuration is scheduled in loose mode and the numbers of operations without an issue slot (dangling operations) are tallied. The offset candidate set with the lowest number of dangling operations is selected. In the event of a tie, the heuristic employed is to select the candidate that incremented the highest offset.

The offset adjustment routine fails and returns false if no operation is scheduled at time zero or if no offset candidate can be generated. The former case indicates that the existing offset constraints are forcing operations to issue later due to loop carried dependencies. In this case, it will not be possible to find a legal schedule at the current II and offset assignments. The latter case indicates a situation where the offsets have been stretched to the maximum extent possible such that the latest issue slot is equal to the largest sum of operation latencies among the modes.

Offsets are initialized to zero at the beginning of each iteration of the II increment loop. The initial offsets could instead be set to reflect placement and routing limitations of the target device.

### E. Incrementing IIs

The initial IIs for OPS are calculated in a manner similar to IMS and 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 the 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. All modes that have an operation involved in a positive cycle become candidates for II increment. The selection is driven by a priority scheme based on the frequency of execution of each mode. The first lowest priority mode involved in a positive cycle has its II incremented. This heuristic incrementally provides a minimal impact on overall application performance while working towards a set of minimum IIs.

When the offset adjustment procedure is unsuccessful, the algorithm increases IIs to add flexibility to the scheduling. While IMS only has one II to increment, OPS must choose which mode II will be degraded. The same priority information used for initial II calculation is combined with an overhead value. The overhead is the product of mode priority and the ratio of current mode II to the minimum mode II calculated at initialization.

The algorithm prefers to increment the lowest overhead mode since this will minimize the impact on overall application performance. This preference is capped at 2x the overhead of any other mode, heuristically selected to avoid skewing the IIs too far.

IV. TARGET CODE

This section provides an overview of how benchmarks are prepared for consumption by OPS and how to make good mode decomposition decisions to achieve the best performance from OPS.

A. Application Preparation

The benchmark code is written in C and then converted to a dataflow graph stored in a physical netlist container from the Torc [2] project. Since we do not currently have a front end compiler, the code is written in a static single assignment form by hand. While writing benchmarks in this way is labor intensive, it provides greater control over the resulting netlist and can be readily automated since a single assignment form is commonly used in compilers as an intermediate representation for optimization. The C implementation results are easily compared to reference code to ensure correctness.

Each legal operation has a one to one mapping with ALU, LUT, memory or stream resource available in the test architecture. Memory and stream operations are represented as function calls that provide the appropriate behavior when executing the C program. Phi nodes in the dataflow graph are represented by C ternary operators. 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 can also be driven from different modes, and thus have more than one source operation.

Modes are annotated using C labels. All assignment statements which become schedulable operations must appear after a label. Operations are assigned to the last label parsed. Goto statements are used to provide the necessary loop control. Conditional goto statements become branch operations for the lead program counter and represent transitions to other modes.

B. Desireable Application Features

There are many ways an application might be decomposed into modes. Modes should ideally be as independent of one another as possible. In order to break an application into modes, the control logic to manage the transitions between modes must be added to convert a baseline single mode implementation to multiple modes. Modes are beneficial when they reduce the amount of 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 control logic to orchestrate the execution.

Another issue is increasing II due to adding conditional branch operations necessary to make mode transitions. For loops with constant bounds, this can be avoided by pre-calculating the branch in previous loop iterations to avoid lengthening recurrence loops. This overhead 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 be able to 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 still consumes issue slots potentially increasing the II. SPR supports mutual exclusion [3] 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 that may be unnecessarily costly. OPS will schedule the loop bodies onto the same resources with the recognition that their execution is mutually exclusive, but allows separate IIs and consumes instruction memory for exactly the number of operations needed.

V. EVALUATION

Offset Pipelined Scheduling is evaluated in comparison to iterative modulo scheduling. The first section describes the target architecture. The benchmarks are introduced and sample results are presented to provide insight into OPS behavior. Summarized results close the discussion.

A. Architecture

The target architecture consists of control domains with two 32-bit ALUs, two 4-input LUTs, a memory block, and a stream port. Each domain also contains a program counter unit for managing the flow of execution. The selected CGRA configuration is based on optimized architectures developed in [4]. This paper focuses strictly on scheduling; placement and routing will be considered in future papers.

While the target architecture is designed for use with OPS, it is also used for the modulo scheduling baseline. In this case, the device has the same resources but is assumed to be configured to mimic a simple modulo counter.
TABLE I. APPLICATIONS

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayer</td>
<td>Bayer filtering, includes threshold and black level adjustment</td>
</tr>
<tr>
<td>DCT</td>
<td>8x8 discrete cosine transform</td>
</tr>
<tr>
<td>DWT</td>
<td>Jpeg2000 discrete wavelet transform</td>
</tr>
<tr>
<td>K-means</td>
<td>K-means clustering with three channels and eight clusters</td>
</tr>
<tr>
<td>PET</td>
<td>Positron emission tomography event detection and normalization</td>
</tr>
</tbody>
</table>

B. Benchmarks

The applications used for evaluation are listed in Table 1 with a brief description. They represent a cross section of signal processing algorithms typical for CGRAs. In order to compare performance between the OPS and IMS implementations, the cycles are normalized to the recurrence limited cycles 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. Fig. 9 shows results for the five benchmarks at four different resource settings.

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 when enough resources are allocated. 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, lowering the effective recurrence limit of this implementation. With few resources, all applications show significant improvement over IMS: in a resource constrained regime, IMS must issue unused operations from inactive modes making it much less efficient when targeting more than a single inner-most loop body.

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 every 25 samples on average.

1) Bayer

This application performs a black level adjustment, edge padding and the bayer demosaicing as found in digital camera processing pipelines. The OPS version is broken into three modes, the first is the black level adjustment, the second handles padding the image and the third performs the demosaicing. While the OPS performance is initially good compared to IMS, it suffers due to an extra cycle in the recurrence loop from the control logic for mode transitions when the single mode version can reach the recurrence II.

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. It is also unrolled eight times to provide more computation relative to the control logic. 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.

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 then packing the results for output. 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.

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.

5) Positron Emission Tomography

The PET application [5] 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 the recurrence limit.

C. Scheduling Behavior

To help demonstrate how the OPS algorithm proceeds, we will focus on the DCT algorithm run on 9 domains. Fig. 10 shows the offset assignment of the domains for each tight scheduling pass during the execution. There are two II increments that occur before the 4th and 8th scheduling passes indicated by the reset of all offsets to zero. The final offset
progression finishes with a successful scheduling. From a netlist perspective, the inner loop of the DCT pass was unrolled to provide more computational work relative to the control overhead. The final DCT offset spacing is relatively uniform, with the stride of the mode IIIs providing a long latency for each iteration while allowing adjacent iterations to coexist in time on the CGRA.

Next we consider the 8-mode DWT. Fig. 11 shows the individual mode IIIs for 1 to 10 domains. The sum of the IIIs 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 schedule for the IMS implementation. While the OPS implementation has a larger overall program size, it outperforms the IMS implementation due to reduced execution overhead.

D. Results

Fig. 12 summarizes the results reporting the ratio of IMS to OPS normalized cycles to provide a speed-up metric. The geometric mean is also included, aggregating across all applications. As we provide more resources, 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 Fig. 13 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.

When provided with enough resources, the applications eventually reach 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.

As shown above, 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 (the intrinsic recurrence loops). Since this IMS solution would be significantly larger
than the corresponding OPS solution, IMS will likely have greater problems when placement and routing are considered. Those steps are beyond the scope of this paper and will be addressed in future work.

It is also important to note that the hardware needed to support IMS and OPS schedules are relatively similar. In fact, an OPS device can be converted to run full-fledged IMS schedules simply by forcing all domains to perform a single loop. Thus, a single design suite could use OPS for resource-constrained modal computations, and use IMS for single-mode designs, or designs without resource constraints.

OPS runtime did not exceed approximately 10 seconds for any of the schedules generated in this work running on modest seven year old commodity hardware.

VI. CONCLUSION

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 also demonstrated a complete scheduling algorithm for these devices. The tool automatically schedules issue slots, determines individual domain offsets, and sets mode IIIs to achieve a high-performance and dense implementation. In future work we will extend these efforts to include placement and routing tools for this new style of coarse-grained computational resource.

VII. RELATED WORK

Related architectures primarily fall into two categories; massively parallel processor arrays (MPPAs) and CGRAs. MPPAs such as Ambric [6] and Tilera [7] are composed of discrete processors. They are less tightly integrated than the proposed CGRA and must be programmed using traditional parallel programming techniques.

CGRAs such as Mosaic [8] and ADRES [9] are designed for modulo scheduled execution. These architectures are tightly integrated and offer tool support [10] to leverage the array but are limited to modulo counters for control.

The Tabula [11] 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 [1] in particular inspire the software pipelined schedule and iterative nature of OPS. Modulo scheduling with multiple initiation intervals [12] 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.

ACKNOWLEDGMENT

NSF grant #1116248 supported this work. Special thanks to Suzanne Wood for development of the K-means benchmark.

REFERENCES