Offset Pipelined Scheduling, Placement, and Routing for Branching 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*—Modulo counter based control of coarse-grained reconfigurable arrays (CGRAs) makes them inefficient for applications with multiple execution modes. This work presents an enhanced architecture and accompanying tool chain that add branching control flow to CGRAs. Offset pipelined scheduling leads to a challenging routing problem. The EveryTime router presented here is an architecture adaptive approach that allows paths to consume resources across modes when necessary and share resources when possible to mitigate the added complexity. The Offset Pipelining tool chain provides 2.1 times better average performance on the same device size with approximately the same channel width compared to a modulo scheduling based tool chain and architecture.**

Keywords- CGRA, scheduling, placement, routing

I. INTRODUCTION

Coarse grained reconfigurable arrays (CGRAs) offer a combination of parallelism and improved area and performance overheads compared to commodity architectures. Register rich with word oriented pipelined interconnects, the logic resources are time multiplexed to maximize potential utilization. Many CGRAs are controlled by a global modulo counter, with mappings created via Iterative Modulo Scheduling (IMS). IMS is efficient for inner loop pipelining of the critical kernel of an application.

Mapping more complex control flow to CGRAs generally incurs a penalty in terms of utilization and the ability to automatically map applications. Arbitrary control flow can be modulo scheduled onto a device using predication, but this approach suffers from idle issue slots. As the control flow becomes more complex, the modulo schedule must support all possible execution paths through the dataflow graph, and instructions on inactive paths still consume issue slots. To avoid the problem of a large predicated dataflow graph, other CAD flows and architectures decompose the device into more independent regions, requiring a mapping process that manages these communicating regions.

A variety of CGRA architectures have been proposed and evaluated. These have demonstrated the re-configurability, power, and performance advantages of this type of architecture for a range of compute intensive application domains. However, they remain limited to the aforementioned modulo counter controlled execution, or are processor arrays which are much less tightly coupled, and thus less efficient. This work introduces a complete back end tool chain for scheduling, placement and routing to map dataflow graphs to a pipelined program counter CGRA architecture that can overcome these issues.

Offset Pipelined Scheduling (OPS) [1] offers a statically scheduled solution for applications with more complex control flow. The approach mitigates the problem of wasted issue slots in multi-mode computations, allowing comparable performance with fewer resources, or better performance when resource constrained compared to modulo scheduling. Along with improved utilization, Offset Pipelining helps limit unnecessary data movement. By changing to another execution mode, intermediate data can remain in situ. Lastly, Offset Pipelining may improve recurrence limited applications by eliminating explicit phi nodes in the dataflow graph in favor of implicit phi nodes at mode transitions.

Modal behavior is a common feature of signal processing algorithms that can benefit from Offset Pipelining. For example, a positron emission tomography (PET) event detector has two clear execution modes. It spends the majority of its time searching for scintillator crystal events, when one is found, a complex localization algorithm runs to process the pulse. Such an application can benefit from OPS style execution because it avoids wasting issue slots on a large, infrequently run block of code.

In this paper we demonstrate the power of OPS systems by presenting a novel, complete toolchain for Branching CGRAs. This includes a previously presented scheduling algorithm, a simulated annealing placer, and a new EveryTime router.

Our new router is based on the QuickRoute [2] pipelined router and handles the added complexity of routing through the possible mode and time combinations that are a byproduct of the scheduling and placement.

After we present the algorithms, we then show the power of OPS for a set of signal processing benchmarks. Note that OPS and IMS represent complementary techniques – an OPS-enabled architecture can support strictly modulo-counter based computations in situations where the application does not exhibit modal operation.

II. RELATED WORK

Related architectures primarily fall into two categories: massively parallel processor arrays (MPPAs) and CGRAs. MPPAs such as Ambric [3] and Tilera [4] are composed of discrete processors. They are less tightly integrated than the proposed CGRA and are programmed using traditional parallel programming techniques.

CGRAs such as Mosaic [5] and ADRES [6] are designed for modulo scheduled execution. These architectures are tightly integrated and offer tool support [7] to leverage the array but are limited to modulo counters for control.

The Tabula [8] SpaceTime architecture was 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. Modulo scheduling, and IMS [9] in particular, inspire the software pipelined schedule and iterative nature of OPS. Modulo scheduling with multiple initiation intervals [10] explores more flexible execution similar to OPS. However, this work targets a traditional VLIW machine and assumes a single program counter, very different from the goal of OPS to help automate mapping to the staggered control domains on a pipelined program counter CGRA.

Another important distinction between architectures and tools in this space is how control of the CGRA is managed at run time. Many approaches provide control externally by configuring the CGRA for a particular task and then reconfiguring it during execution of a given application. Offset Pipelining is a static mapping process for a multi-mode application requiring no reconfiguration at run time. This limits the amount of non-application data that needs to be transferred to the device just to control execution.

III. Pipelined Program Counter CGRAs

The following subsections describe the branching CGRA architecture targeted by the Offset Pipelining tool chain.

A. *Pipelined Program Counter Architecture*

Unlike a conventional modulo counter controlled CGRA, a pipelined program counter architecture adds a local program counter to each resource cluster (domain) for issuing instructions. However, the instruction pointer is received from a neighboring domain in the array. A designated lead domain contains the logic to make program counter adjustments, thus managing the execution sequence of the application on the device. All other domain program counters receive their instruction pointers from adjacent domains, essentially a “follow the leader” style of execution. In this way, the program counters issue instructions staggered from one another, but follow a common execution path managed by the lead domain.

B. *Offsets*

In order to accommodate the time needed to deliver the instruction pointer, as well as provide issue slots at the necessary times for a given application, each domain is assigned an offset that fixes its execution relative to the lead domain, which is assigned an offset of 0.

In order to receive the delayed program counter, each non-leader domain offset must be greater than at least one of its neighbors, which represents a constraint on the legal arrangement of resources. The offset is a configured property of the domain shared by all modes and is a critical component of the Offset Pipelining execution model.

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

C. *Architecture Parameters*

The domain composition is based on the cluster developed in [11]. For this work each domain is composed of the following logic resources:

- 2x 32-bit ALUs
- 2x 4-input LUTs
- 1x 4 KB memory
- 1x 8 entry register file
- 1x program counter
- 1x input stream port
- 1x output stream port

The ALUs support a variety of arithmetic, logic and comparison operations. Operations are single cycle, except multiplication which has a two cycle latency. Retiming chains (units that can be configured to add anywhere between 0 and 3 cycles of delay) are available at each domain input to provide additional routing flexibility.

In standard modulo-scheduled architectures, such as that of SPR [7], rotating register files have been shown to be more appropriate than standard register files because of the limits of these systems. However, in an OPS-based system, standard register files can be used efficiently, since different modes can support different access patterns into the register file.

Interconnect resources are registered at multiplexor outputs and divided into two categories. The first is a 32-bit word oriented resource with a valid bit which connects to ALU inputs and outputs, memory, register file and stream ports. A second single bit interconnect is available for LUTs which also connects to memory write, program counter control signals and ALU control and flag bits.

Inter-domain interconnect is track-oriented where each track comprises a 32-bit and 1-bit signal in each direction. A crossbar for each interconnect width connects domain resources to inter-domain switch boxes. While values can be routed within a domain in a single cycle, inter-domain communication takes a minimum of three cycles in order to exit the logic block, transit the interconnect to an adjacent domain and enter the destination logic block.

The architecture supports an 8-bit program counter, with dedicated resources to transfer this to neighboring domains. With an 8 bit program counter, each domain supports at most 256 unique configurations or “instructions”.

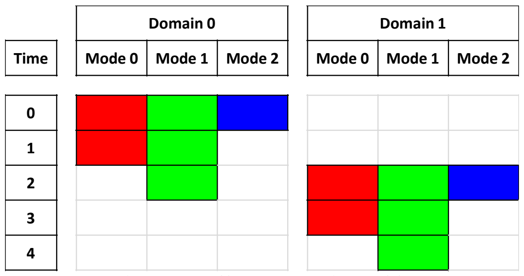


Figure 1. Offset Reservation Table

The program counter manages per cycle configurations for the domain it controls. While this is a feature added to every domain compared to a conventional CGRA, the expected overhead is minor, particularly considering the added flexibility. Work on enhancing CGRA architectures for dataflow driven execution [12] demonstrates that these costs are modest.

D. *Execution Model*

By assigning a fixed offset to each domain, an arbitrary execution sequence can be managed by the lead domain while allowing different iterations of different modes to execute simultaneously on the device. Each domain ultimately follows the execution sequence of the leader, delayed by *offset* cycles.

The cascading control of a pipelined program counter CGRA mitigates the cost of prologue and epilogue specific code by effectively merging it with the steady state loop behavior. Each domain executes *initiation interval* (II) operations for the current mode, and then either repeats the mode or branches to another. A given loop iteration of the application is spread across the device with operations assigned to domains with different offsets.

Special prologue and epilogue code found in modulo scheduled systems is unnecessary for Offset Pipelining because each iteration has been broken down to execute across the offset domains. The equivalent of prologue code on a pipelined program counter CGRA is the lead domain beginning an iteration of a mode while the other domains in the system continue to execute the iteration that the lead domain started *offset* cycles previously. Epilogue code is this tail of execution that continues even after the lead domain has moved on to a different iteration of a possibly different mode.

IV. Dataflow Graphs

Applications are described in a static single assignment form written in C. This representation can be readily compared to a reference implementation for verifying functionality. It is converted into a dataflow graph represented as an XDL netlist leveraging Torc [13] APIs. As part of the conversion, mode execution frequency and a mode transition graph are included for later use by the various phases of mapping. Execution frequencies are used to prioritize mode II adjustments in scheduling and the mode transition graph defines the possible sequences of mode execution used for EveryTime routing.

```
initializeIIs()
do {
   initializeOffsets()
   resetLooseSchedulingCache()
   do {
      buildORT()
      if ASAPschedule(tight)
         return SUCCESS
      offsetsUpdated = offsetAdjustment()
   } while (offsetsUpdated)
   iisUpdated = incrementIIs()
} while (iisUpdated)
```

Figure 2. Top level OPS algorithm

V. Scheduling

The scheduling portion of the tool chain is an iterative variant of list scheduling that adds the concept of domain offsets and the associated issue slot windows to the process. Like iterative modulo scheduling, when the scheduling routine fails to find a legal schedule, an adjustment is made. However, instead of merely increasing the II of the system, OPS adjusts either the domain offsets or the mode IIs in order to allow a subsequent scheduling attempt to make progress towards a legal scheduling.

A. *Offset Reservation Table*

The offset reservation table (ORT) is analogous to the modulo reservation table (MRT) used in iterative modulo scheduling (IMS) [9]. 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 is shown in Fig. 1 for three modes with IIs 2, 3, and 1 from left to right. Note that the relative positions of time slots in the ORT correspond to the domain offsets of 0 and 2 for the respective domains, defining the available time slots for the enclosed logic units.

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 on IMS, sharing its node prioritization scheme, list scheduling algorithm and notion of adjusting the scheduling window on failure. A pseudo code representation of OPS is shown in Fig. 2.

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 in an earlier cycle. We perform loose scheduling to estimate the desired number of issue slots at each time, before adjusting domain offsets to try to meet these demands. Within a given set of mode IIs, the algorithm only increases domain offsets.

C. *Operation Prioritization*

OPS uses a height based priority scheme very similar to IMS [9]. 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 in that 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.

D. *Offset Adjustment*

The main driver of progress for OPS is adjusting offsets to provide a better fit for the application. The modification starts with a loop that performs loose scheduling 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 this “offset shaping”, these 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 with a tight scheduling attempt.

The shaping process moves offsets to better match the available issue slots to satisfy a given application, based on the loose scheduling results. Since the scheduler uses an as-soon-as-possible scheduling and the offsets start at zero, shaping basically involves increasing offsets to meet the schedule’s demands. Specifically, it makes use of two observations: (1) if the first issue slot of a domain is always unused, that domain’s offset can be increased; (2) if there is a scheduled operation later than any available issue slot, increase a domain’s offset to reach this operation. When no deterministic progress can be made, a search of possible offset increments is performed to find a legal scheduling.

E. *Incrementing IIs*

The initial IIs for OPS are calculated in a manner similar to IMS [9] and begin 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. This heuristic incrementally provides a minimal impact on overall application performance.

When the offset adjustment procedure is unsuccessful, the algorithm increases an II to add flexibility to the scheduling. While IMS only has one II to increment, OPS must choose which mode II will be degraded. The 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 to minimize the impact on overall application performance.

VI. Placement

A scheduled netlist includes a set of domain offsets and assigned time slots for each operation in the dataflow graph. The scheduler guarantees that the domain offsets will provide sufficient issue slots for the scheduled operations. The simulated annealing placer must assign offsets to the physical domains and also assign operations to issue slots in the domains. An initial placement assigns domains and operations randomly while respecting the scheduled time slots. The move function swaps pairs of operations at the same time slot relative to the start of an iteration, or swaps the entire contents of two domains. Both of these move types preserve the schedule constraints while exploring the space of possible placements for the scheduled netlist. Selection of the move type is relative to the number of movable objects of each type, either operation or domain.

The cost function reflects the minimum possible latency between source and sink placement, which indicates whether or not the net can be routed. A successful placement reaches a cost of 0 indicating that each source and sink pair in the design can be routed to achieve the required latency, leaving the issue of congestion to the router.

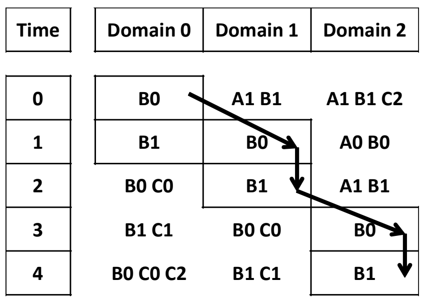
Following the VPR [14] cooling schedule, if the placer cannot find a legal placement, nets that have non-zero cost are annotated and the netlist is passed back to the scheduler. The scheduler produces a new solution with added latency for the failing nets to provide more flexibility in placement.

VII. Routing

The routing problem for an Offset Pipelined CGRA is unique to the proposed execution style. This section will introduce the problem, provide an overview of the EveryTime router approach, and discuss an enhancement for routing nets with a decoupled source and sink relationship.

A. *Dealing with Iteration Space*

Routing on a modulo scheduled architecture requires tracking use of physical resources for each time slot in the schedule, effectively unrolling the architecture graph in time to represent the available resources. Adding the dimension of independent modes means that physical resources must be tracked by mode as well as time within that mode. This is sufficient for scheduling and placement and the data structures used for those phases of the tool chain reflect this organization. However, the router must further understand how to traverse the possible mode transitions and track the utilization of a resource in multiple modes and times simultaneously.



Time	Domain 0	Domain 1	Domain 2
0	B0	A1 B1	A1 B1 C2
1	B1	B0	A0 B0
2	B0 C0	B1	A1 B1
3	B1 C1	B0 C0	B0
4	B0 C0 C2	B1 C1	B1

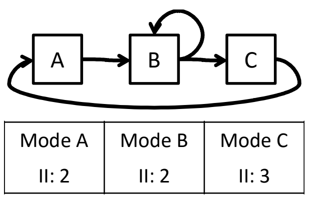
Figure 3. EveryTime route.

For modulo scheduling, the next cycle is always known, namely it is "(the current cycle plus one) modulo II". In the Offset Pipelined system, moving forward or backward in time relative to a known point can lead to multiple possible modes and times in different iterations. This is determined by the assigned domain offsets and how the physical resources are traversed. In the most basic case, within a domain, moving forward in time one cycle has two possibilities: either the next cycle is still within the II cycles of the current iteration, or the next cycle is in a new iteration. The new iteration may be in the same mode, or one or more other modes, depending on the mode transition graph.

Moving between domains adds further complexity since the destination domain usually has a different offset. There are three cases depending on the particular mode, time, and difference in offset between the two domains. The destination may be in an earlier iteration, the same iteration or a later iteration. To illustrate this concept, EveryTime sequence data is shown in Fig. 3 along with a simplified representation of a route with the mode transition graph shown in Fig. 4. The net in question has a source in domain 0 in cycle 0 of mode B (denoted B0). The sink is in in domain 2 in mode/time B1 relative to that domain with a flight time for the net of 4 cycles. The intervening domain 1 has an offset one greater than the source domain and two less than the sink. All issue slots for the iteration containing the source and sink are boxed in Fig. 3 to help visualize the offset relationship. The highlighted path traverses only resources in the iteration of the source and sink. However, if the offset of domain 1 were increased, the EveryTime sequence for the domain would be shifted later by one cycle and the original path would then visit resources outside the current iteration.

The other issue slots are populated using the mode transition graph to determine the possible modes and times that may be active relative to the source and sink of the net. For example, on domain 0, at time 2, we have passed the end of the iteration of B, so a new iteration will begin. The mode transition graph tells us this could be either another iteration of B, or an iteration of C, so we have B0 or C0 in this slot. By the time cycle 4 is reached, one of two different iterations of mode C may be active. Not shown in these examples is tracking of the specific iteration to which a resource belongs. This is important for iteration delayed nets to ensure the signal arrives at the correct iteration.

The set of all modes and times that could be active for a resource while routing is the foundation of the EveryTime concept. Further from an anchor mode and time, a larger set of active modes and times develops. The router tries to avoid this situation, but handles it when necessary.



Mode A	Mode B	Mode C
II: 2	II: 2	II: 3

Figure 4. Example mode transition graph.

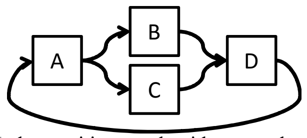


Figure 5. Mode transition graph with two paths from A to D.

Applying standard CGRA routers to the multi-mode routing problem is difficult. To illustrate this, consider a net from mode A to mode D with a mode transition graph as shown in Fig. 5. The problem is that two paths, one traversing mode B, the other mode C, must emerge at the same point to enter mode D and reach the sink. It is difficult to formulate this type of routing in a way that allows PathFinder to resolve contention between what is essentially the same net. The EveryTime approach solves this problem by consuming all the times that may be active along a given path.

B. *EveryTime Routing*

The EveryTime router uses the QuickRoute [2] algorithm at its core for pipelined routing. The router knows the mode and time of the source and sink, which provides fixed anchor points for the routing. If the source and sink belong to the same iteration, or there is only one sequence of modes connecting source and sink, the net is considered "time locked", that is, the exact flight time of the signal is known. In this case, as the router explores a resource in the architecture, we know how far we are from the source and from the sink, in both time and space. This allows calculation of the set of active modes and times from the mode transition graph, as illustrated in Fig. 3. Note that since signals can traverse domains with very different offsets, this calculation must consider iterations from before the source or after the sink.

From a cost perspective, a path must pay for the use of all the mode/times that are active along the path. For example, if we try to use a resource where six different mode/times are possible, the cost of this resource is the sum of the costs of each of the six mode/time possibilities. This will encourage routes to use paths that are less uncertain, but allow paths to use whatever resources necessary to achieve the required routing.

C. *Resolving Congestion*

PathFinder [15] provides the mechanism to resolve congestion. A given net consumes whatever mode/times tuples are part of the active set for each node in the path.

PathFinder operates only on the mode/time information to address present and history sharing costs.

While conventional routing algorithms support nets with a single source but multiple sinks, Offset Pipelined netlists also involve nets with multiple sources in certain situations. For example, a loop index variable in an application can be initialized in one mode and updated in another, and therefore has two possible sources. This is reasonable since the dynamic execution pattern will determine which source actually generated a given signal at run time.

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

D. *Unlocked Nets*

So far we have considered "time locked" nets, where the mode execution sequence between the source and sink is uniquely known by traversal of the mode transition graph. "Time unlocked" nets have multiple mode execution sequences between source and sink. For example, considering the mode transition graph in Fig. 4, a signal with a source in A and a sink in C is a "time unlocked" net, since there can be one or more iterations of B between A and C. In this case, the router faces two challenges: (1) we must find a way to retain the value of the signal during an arbitrary number of iterations of mode B, perhaps via a feedback path; (2) the set of possible mode/times on the path (as shown in Fig. 3) becomes very complicated, since there are many possible execution paths.

We solve this problem with a series of steps. To handle the feedback path, we require each unlocked net to enter a register file before reaching the destination. All routing from the source to the register file is source-relative, in that we compute the set of possible executing modes and times relative to the source mode. All routing from the register file to the sink is sink-relative, where we compute the set of possible executing modes and times relative to the sink mode. In this way, we convert the time unlocked route into two time locked signals stitched together via a register file. Note that this register file is dynamically determined via PipeRoute's [16] phased search concept.

Due to multiple possible run time execution sequences, the time required to send the signal from source to sink is not known. However, we use the mode transition graph to find the minimum delay required. Meeting this minimum delay ensures the communication will complete no matter which mode sequence executes at run time.

Routing out of the register file is simple, since a register can be read multiple times. However, to determine when the register should actually be written, we use the 1-bit valid signal associated with each 32-bit routing track to control the register file write enable. Negotiations for register assignment is handled via EveryTime and PathFinder, which imposes a cost on all times that a given value is live.

Finally, we consider what resources the EveryTime router must use when routing a time unlocked signal. For example, consider a signal from mode A to B in Fig. 5 (although this is a time locked signal, it will help illustrate the issues faced with time unlocked signals as well). However, since we are routing from A to an immediately following iteration of B, we would not need to complete this route if mode C were initiated. To handle such cases, we consult the mode transition graph to prune mode/times along the path that would imply an additional intervening initiation of the source or sink mode.

TABLE I. APPLICATIONS

Application	Description
Bayer	Bayer filtering, including 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

VIII. EVALUATION

The Offset Pipelining tool chain is evaluated in comparison to an SPR implementation that uses iterative modulo scheduling as the scheduling foundation. The benchmarks are introduced and sample results presented to provide insight into Offset Pipelined behavior.

A. *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. The Offset Pipelining tool chain is compared to SPR [7] for several reasons. The pipelined program counter CGRA used in this work is modeled after the architectures SPR targets; both tool chains consume the same type of control dataflow graph input; the two approaches share a certain amount of common lineage such as a scheduler based on list scheduling, a simulated annealing placement phase, and a PathFinder and QuickRoute based routing scheme.

The applications are mapped using each tool chain to a given size architecture. We then compare the number of cycles required to execute on a representative dataset.

Unfortunately, our SPR implementation does not support the more advanced mutual exclusion sharing available in later development of the tool [17]. This "predicate aware" version of SPR (PA-SPR) applies predicates to the instruction fetch to ensure exclusive use of issue slots. In

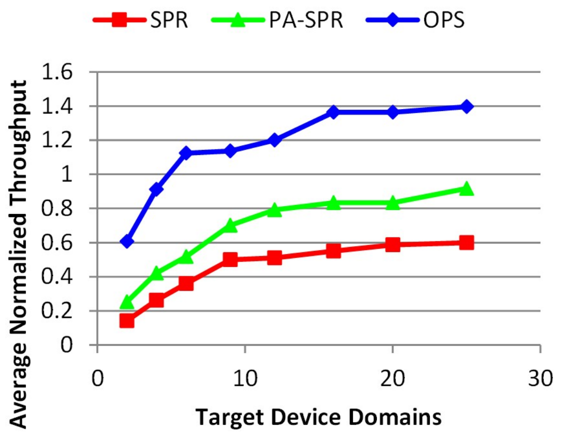


Figure 6. Performance relative to SPR baseline.

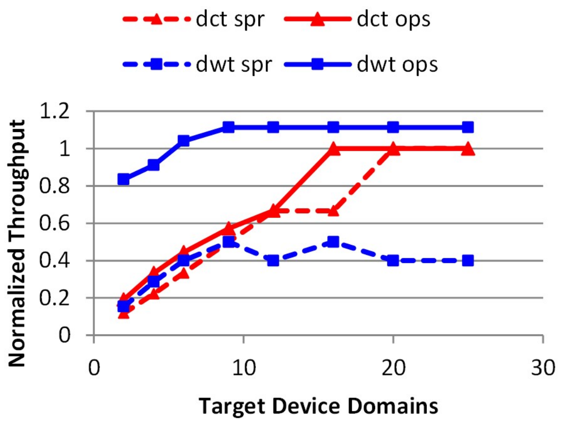


Figure 7. DCT and DWT performance normalized to recurrence limit.

order to simulate this capability, we schedule the multi-mode version of each application using OPS for the available resources and then calculate the theoretical performance of the SPR implementation by using the largest mode II, since the single II constraint still applies. This estimate is optimistic since it does not require SPR to complete placement and routing and ignores the required phi nodes, but stands in as a upper bound of what mutual exclusion SPR might achieve.

B. *Effects of Placement and Routing*

While a previous introduction to Offset Pipelined Scheduling shows a solid advantage over Iterative Modulo Scheduling due to a reduction in wasted issue slots, the practical implementation including placement and routing must deal with constraints that SPR does not. The main difficulty in taking the applications through placement and routing is ensuring that the placed netlist will be routable, with a secondary concern being the number of routing tracks necessary to realize the complete implementation.

While generally straightforward in its implementation, the placer plays an important role in preparation for routing. Only a placement cost of zero is allowed to proceed to routing since the cost function reflects the routability of all nets in the target application. While it does not address congestion, the placer guarantees that each individual net has a possible route.

For routing, a binary search over available tracks determines the minimum number needed to route the design. In a real device with a specific number of resources, the router would provide feedback to the placer and scheduler when it is unable to route the design. Given the generally modest track counts for the tests performed in this work, it seems reasonable to assume a specific CGRA implementation would have sufficient resources to realize a wide variety of useful applications.

C. *Results*

A comparison of Offset Pipelining to SPR and to an upper bound on what predicate-aware SPR might achieve is shown in Fig. 6. Performance for each application is normalized to the recurrence limited performance of the modulo scheduled dataflow graph. Offset Pipelining achieves approximately 2.1 times the performance of the SPR baseline on average and 1.4 times the performance of PA-SPR. Through a combination of reduced effective II and a more compact implementation using multiple modes, a significant portion of the potential advantage is realized through placement and routing.

In general, we see Offset Pipelined implementations achieving better performance with the same number of resources as SPR. Alternatively, a smaller device using Offset Pipelining can achieve the same performance as the conventional CGRA mapped with SPR.

The average number of tracks to route the designs among all device sizes and applications is 4.8. The maximum is 20 and the minimum is 2. Track counts tend to be larger for smaller devices. Routing must deal with more signals in a tighter physical area as well as higher IIs when heavily resource constrained.

Looking at the DCT and DWT applications in particular provides good insight into the effectiveness and limits of OPS. Fig. 7 shows normalized performance results for these two applications implemented with SPR and OPS. The DCT is broken into two modes and performance is slightly improved over the SPR baseline. Looking at the DWT, which has eight modes, there is a substantial performance improvement. This is because OPS can eliminate predication and minimize inter-mode routing conflicts, which can provide a significant advantage.

This is not to say that the modulo scheduled approach is inferior; in fact it may still be preferred for applications with limited control. SPR does not have to deal with the added complexity of EveryTime routing and has fewer constraints on issue slot assignment, but as the modal behavior of the target application increases, the OPS style execution outstrips a modulo scheduled solution.

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.

IX. Future Work

Many avenues are available for further study. An architecture exploration would help identify optimization strategies for pipelined program counter CGRAs. Occasional routing failures can be attributed to poor placement with high congestion. A more sophisticated cost function might be able to address this issue and improve router run time. While stream ports are independent logic resources, a future implementation might merge them into memory blocks that can be configured as a queue.

The most important component needed to make a more viable tool chain would be a dedicated compiler front end perhaps based on LLVM. Netlist generation would be straightforward since the existing applications are converted from single assignment form commonly used for optimization in conventional compilers.

X. Conclusion

When considering word-oriented FPGAs and FPGA-like systems, architectures have split into MPPA and CGRA style devices. CGRAs provide excellent parallelism, and have automatic mapping tools that spread a computation across a large fabric. The restriction of CGRAs 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 toward 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 back end tool chain for these devices. The tool automatically schedules, places and routes the application to achieve a high-performance and dense implementation.

Acknowledgment

NSF grant #1116248 supported this work.

References

[1] Aaron Wood, Scott Hauck, "Offset Pipelined Scheduling: Conditional Branching for CGRAs", IEEE Symposium on Field-Programmable Custom Computing Machines, 2015.

[2] Song Li; Ebeling, C., "QuickRoute: a fast routing algorithm for pipelined architectures," in Field-Programmable Technology, 2004. Proceedings, pp.73-80, Dec. 2004.

[3] M. Butts, A. Jones, and P. Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'08), pages 55-64, April 2008.

[4] Tilera. http://www.tilera.com/.

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

[6] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In International Conference on Field-Programmable Logic and Applications, pages 61–70, 2003.

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

[8] Tabula. http://www.tabula.com/

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

[10] Warter-Perez, N.J.; Partamian, N., "Modulo scheduling with multiple initiation intervals," Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on, pp. 111-118, Dec 1995.

[11] Brian Van Essen, Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays, Ph.D. Thesis, University of Washington, Dept. of CSE, 2010. http://www.ee.washington.edu/people/faculty/hauck/publications/dissertation-vanessen.pdf.

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

[13] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French, "Torc: Towards an Open-Source Tool Flow," ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 41–44, 2011.

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

[15] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-based Performance-driven Router for FPGAs. In ACM International Symposium on Field-Programmable Gate Arrays, pages 111–117, 1995.

[16] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, No. 3, pp. 518-532, March 2006.

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