

SPR: An Architecture-Adaptive CGRA Mapping Tool

Stephen Friedman[†]
Benjamin Ylvisaker[†]

Allan Carroll[†]
Carl Ebeling[†]

Brian Van Essen[†]
Scott Hauck[§]

[†] Dept. of Computer Science and Engineering and [§] Dept. of Electrical Engineering
University of Washington, Seattle, WA 98195
{sfriedma, allanca, vanessen, ben8, ebeling}@cs.washington.edu
hauck@ee.washington.edu

ABSTRACT

In this paper we present SPR, a new architecture-adaptive mapping tool for use with Coarse-Grained Reconfigurable Architectures (CGRAs). It combines a VLIW style scheduler and FPGA style placement and pipelined routing algorithms with novel mechanisms for integrating and adapting the algorithms to CGRAs. We introduce a latency padding technique that provides feedback from the placer to the scheduler to meet the constraints of a fixed frequency device with configurable interconnect. Using a new dynamic clustering method during placement, we achieved a 1.3x improvement in throughput of mapped designs. Finally, we introduce an enhancement to the PathFinder algorithm for targeting architectures with a mix of dynamically multiplexed and statically configurable interconnects. The enhanced algorithm is able to successfully share statically configured interconnect in a time-multiplexed way, achieving an average channel width reduction of .5x compared to non-shared static interconnect.

Categories and Subject Descriptors

D.3.4 [Processors]: Retargetable compilers; B.7.2 [Design Aids]: Placement and routing

General Terms

Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Interest in spatial computing is being revitalized because sequential computing performance has been unable to keep pace with increasing transistor density. Spatial computers use a large number of simple parallel processing elements, which operate concurrently, to execute a single application or application kernel. Designers of traditional general purpose processors are attempting to keep pace with Moore's Law by adding multiple processor cores, and are

rapidly moving from multi-core to many-core designs. Processors will contain dozens, if not hundreds of cores in the near future. However, communication, in the form of registers, memories, and wires dominates the area, power, and performance budgets in these new devices. Existing spatial processors, such as FPGAs, are adding more coarse-grained, special purpose units, to minimize these communication and configuration overhead costs. At the collision of the two trends lie Coarse-Grained Reconfigurable Architectures (CGRAs).

CGRAs are spatial processors that consist of word-sized computation and interconnect elements capable of reconfiguration that are scheduled at compile time. Typical compute elements are simple ALUs, multipliers, small CPUs, or even custom logic for FFTs, encryption, or other application-specific operations. CGRA interconnects are register rich and pipelined to increase the possible clock speeds and make time-multiplexing possible. Unlike FPGAs, which are traditionally load-time configurable, CGRAs loop through a small set of configurations, time-multiplexing their resources.

Previously, a number of CGRA architectures have been proposed, including RaPiD [6], ADRES [13], MATRIX [14], Tartan [15], MorphoSys [18], and HSRA [19]. These architectures sampled the possible design space and demonstrated the power, performance, and programmability benefits of using CGRAs.

Each of the previously mentioned CGRA projects required custom mapping tools that supported a limited subset of architectural features. We are developing a new adaptive mapping tool to support a variety of CGRAs. We call this architecture-adaptive mapping tool SPR (Schedule, Place, and Route). SPR's support for features unique to CGRAs makes it a valuable tool for architecture exploration and application mapping across the CGRA devices that have and will be developed. In this paper, we describe techniques in the SPR algorithms that support mapping for statically-scheduled, time-multiplexed CGRAs. For each stage of SPR, we provide an introduction to the base algorithms, and then we describe enhancements for supporting features of CGRAs.

2. RELATED WORK

Despite the large number of CGRAs that have been proposed in the literature, little in the way of flexible tools has been published. Most projects have mapping tools of some form, but they are tied to a specific architecture and/or are only simple assemblers that aid mapping by hand. The most flexible are DRESC [12] and the tool in [9], both of which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'09, February 22–24, 2009, Monterey, California, USA.
Copyright 2009 ACM 978-1-60558-410-2/09/02 ...\$5.00.

only support architectures defined using their limited templates.

Of the existing tools, DRESC is the closest to SPR, as it is also intended as a tool for architecture exploration and application mapping for CGRAs. DRESC exploits loop-level parallelism by pipelining the inner loop of an algorithm. Operators are scheduled in time, placed on a device, and routed simultaneously inside a Simulated Annealing framework. Their results indicate good quality mappings, but the slowdown from using scheduling, placement, and routing jointly within annealing makes it unusable for all but the smallest architectures and algorithms. DRESC only supports fully time-multiplexed resources, not more efficient statically configured resources of architectures like RaPiD, nor does its router handle pipelining in interconnect.

CGRA mapping algorithms draw from previous work on compilers for FPGAs and VLIW processors, because CGRAs share features with both devices. SPR uses Iterative Modulo Scheduling [16] (IMS), Simulated Annealing [8] placement with a cooling schedule inspired by VPR [3], and PathFinder [11] and QuickRoute [10] for pipelined routing.

IMS is a VLIW inspired loop instruction scheduling algorithm. IMS heuristically assigns operations to a schedule specifying a start time for each instruction, taking into account resource constraints and data and control dependencies. SPR uses IMS for initial operation scheduling, and we have extended IMS to support rescheduling with feedback from our placement algorithm, letting us handle the configurable interconnects of CGRAs.

FPGA mapping tools historically use Simulated Annealing for placement and PathFinder for routing. VPR, which has become the de facto standard for FPGA architecture exploration, is similar to SPR in that it seeks to be a flexible and open mapping tool that can provide high quality mappings and support a wide spectrum of architectural features. Unfortunately, it only applies to FPGAs. With the success of VPR, we have chosen to adopt the same base algorithms, though we have extended them to CGRAs by supporting multiplexing and solving the placement and routing issues that arise when using a fixed frequency device.

SPR uses QuickRoute to solve the pipelined routing problem. More recently, QuickRoute was extended to perform timing-driven routing [7] and have reduced memory complexity [4]. SPR does not yet incorporate these extensions, but we hope to support them in the near future.

2.1 Mosaic

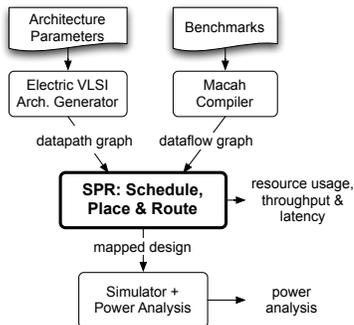


Figure 1: Mosaic project tool chain.

SPR is a component of a larger project called Mosaic [2]. The Mosaic project has started an exploration of architectures and programming tools with the goal of quantifying the architectural trade-offs and necessary innovations in tool support for CGRAs. The project consists of three parts: a new system level language, Macah, an architecture-adaptive back-end mapping tool, SPR, and an architecture generation tool and characterization effort. Figure 1 shows a block diagram of the Mosaic project tools. The final goal is to produce a high-performance, low-power device and a set of compiler tools that will ease the programming burden.

3. MAPPING APPLICATIONS TO CGRAS

As shown by the authors of DRESC [12], mapping applications to CGRAs has similarities to the problems of scheduling computations on VLIW architectures and placing and routing computations on FPGAs. The difficulty comes in making these algorithms work together and adapting them to the particulars of CGRAs. For mapping, we represent the application as a *dataflow graph* (shown in Figure 2) and the architecture into a *datapath graph*. We describe our architecture representation in Section 4.

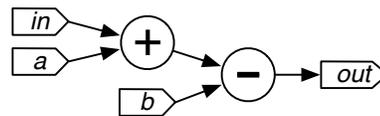


Figure 2: Example of a simple dataflow graph.

In DRESC, the authors chose to implement this mapping process as a monolithic scheduling, placement, and routing algorithm unified within a Simulated Annealing framework. Integrating placement and routing this way was shown to be significantly slower for FPGAs in Independence [17] when compared to the separate stages of VPR [3]. We expect the slowdown to be worse when you include the scheduling for time-multiplexed coarse grain devices, so we avoid the monolithic approach and divide the mapping process into three closely coupled but distinct algorithms:

- Scheduling - ordering operations in time based on data and control dependencies.
- Placing - assigning operations to functional units.
- Routing - mapping data signals between operations using wires and registers.

To illustrate how these algorithms are combined, the main loop of SPR is shown in Figure 3. As you can see, it uses IMS [16], Simulated Annealing [8] placement, and PathFinder [11] with our extension that is described in Section 7.1. We use QuickRoute [10] as the signal level router for PathFinder to produce pipelined routes. This allows flexible use of interconnect registers during routing, rather than being limited like DRESC is to fixed register placement in register files.

The other interesting subroutines are described throughout the rest of this paper. The subroutine `unrollGraph()` of the datapath graph handles translating our architecture description into a graph suitable for placement and routing, and is discussed in Section 5.1. The subroutine `padSchedule()` implements our latency padding technique which communicates the need for extra slack in placement back to the scheduling stage, and is discussed in Section 6.1.

```

SPR(){
  while(iterate){
    minII = iterativeModuloSchedule(minII);
    dataPathGraph.unrollGraph(minII);
    placeSuccess = simAnnealingPlacement();
    if(!placeSuccess){ padSchedule(); }
    else{
      routeSuccess = pathFinderRouting();
      if(!routeSuccess) minII++;
    }
    sprIters++;
    iterate = ( !(placeSuccess && routeSuccess)
               && sprIters < maxIters);
  }
}

```

Figure 3: Main body of SPR

SPR was designed with several assumptions based on the types of programs it will map and the range of current CGRAs. First, we assume we are mapping a kernel consisting of a single loop which will be pipelined by SPR. Currently, a kernel can be described in the Macah language using nested and sequenced loops, and the Macah compiler will turn them into a single loop dataflow graph [5]. Second, we assume we are mapping to a modulo-counter, time-multiplexed architecture. That means the architecture can switch an entire configuration per clock cycle using a modulo-counter. Though some architectures support more complex control, this simple multiplexing is the most frequently implemented approach across a range of architectures. Finally, SPR currently assumes a fixed frequency device where routes cannot be constructed that violate that frequency. In the future we will look at lifting some of these assumptions.

4. ABSTRACT REPRESENTATION

To achieve architecture-adaptability, SPR’s architecture representation remains very abstract. An architecture is represented as a datapath graph, which is defined in Verilog out of a few primitives. Verilog modules prefixed with `primitive_` are used to represent arbitrary functional units. The Verilog is flattened into a directed graph. Primitive Verilog modules form the nodes and wires form the arcs. There are two types of distinguished primitives that receive special treatment in this translation process:

- `primitive_register` - A register to be used by the router to route in time.
- `primitive_tap, primitive_stap` - A configurable connection between two wires (pass gate).

Registers are distinguished so that QuickRoute [10] can use them for pipelined routing. Additionally, we improve QuickRoute’s efficiency by not representing them as nodes, but instead recording them as latency on arcs between nodes.

The connections in the interconnect that are controlled by configuration bits are represented as tap and s-tap devices. A `primitive_tap` device is a dynamic connection, meaning it has an array of bits controlling its configuration allowing time multiplexing. A `primitive_stap` device is a static connection, meaning it has a single bit controlling its configuration for the life of the application. Dynamic connections are more flexible, but have higher area and power requirements for storing and switching the configuration. Static connections can be more area and power efficient, but at the cost of

flexibility. A set of taps whose outputs are connected to the same wire are aggregated into a logical mux by SPR to ensure that two taps are never made to drive the same wire at the same time. With this representation, SPR can support a mix of static and dynamic interconnect, like those found in the RaPiD [6] and MATRIX [14] architectures. Example static and dynamic muxes are shown in Figure 4.

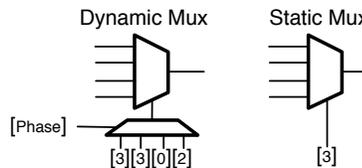


Figure 4: Dynamic and static mux representation.

To support a specific architecture, four things need to be done. First, the architecture must be described in Verilog using primitive nodes as outlined above. Second, a function to estimate the cost of routing from one node to another must be created. This is used for placement cost calculations and the A* search in QuickRoute. Additionally, many architectures support the notion of clusters with cheaper/faster local interconnect and more expensive global interconnect. This is represented in SPR by assigning every node a cluster coordinate, where SPR assumes anything with the same coordinate is in the same cluster. Third, a mapping between datapath graph operation types and primitive functional unit types must be defined. This mapping is a many-to-many relation, for example mapping either an ADD or an OR operation onto an ALU functional unit, or mapping an ADD operation to either an ALU or an ADDER functional unit. Finally, a subroutine must be written for translating the abstract internal configuration into an appropriate form for the architecture, such as a bitstream. The second through fourth items will eventually be handled by SPR plug-ins, but are currently implemented using a simple API to java objects. We believe this minimal amount of work to support a new architecture will allow easy adaptation to a variety of CGRAs. Now that we have described the main loop of SPR and how it represents architectures, we can delve into the details of each stage.

5. SCHEDULING

The scheduling problem is addressed in SPR using the IMS [16] algorithm. The result is a complete schedule that specifies when each operation can execute given the data dependencies and architectural resource constraints. The schedule repeats every II (Initiation Interval) cycles, with a new iteration of the application loop starting each repetition. The II is determined by several things, and the reader is directed to [16] for the details, but one that is important to our discussion is the maximum recurrence loop. When values from the current iteration are needed by future iterations, those values must be computed before the future iteration needs them. This is called a recurrence loop or loop carried dependence, and the largest recurrence loop is a lower bound on the II. This will be important when we are discussing our CGRA specific extensions to the placer.

Using IMS allows us to easily trade off between resource constraints and throughput. To illustrate this, consider the following examples. Table 1 shows a possible schedule for

our example dataflow graph from Figure 2. The target datapath graph in this example contains one ALU, one stream-in device, one stream-out device, and one constant device. In this example we need two ALU operations and two constants per iteration. However, our architecture only has one of each, requiring that the schedule start an iteration every other cycle, with a four cycle latency.

Table 1: Example schedule with II 2 and length 4.

Cycle	alu	str_i	str_o	cnst	
0		in[0]		a[0]	lt0
1	add[0]			b[0]	
2	sub[0]	in[1]		a[1]	lt1
3	add[1]		out[0]	b[1]	
4	sub[1]	in[2]		a[2]	lt2
5	add[2]		out[1]	b[2]	

If we increase the resources available by adding an ALU and another constant, we get the schedule given in Table 2. Our schedule length for a single iteration is still four cycles, but we can initiate a new iteration every cycle, giving us an II of one, thus doubling our throughput.

Table 2: Example schedule with II 1 and length 4.

Cycle	alu0	alu1	str_i	str_o	cnst0	cnst1	
0			in[0]		a[0]		lt0
1	add[0]		in[1]		a[1]	b[0]	lt1
2	add[1]	sub[0]	in[2]		a[2]	b[1]	lt2
3	add[2]	sub[1]	in[3]	out[0]	a[3]	b[2]	lt3

Operators that do not fall on the critical scheduling path may have some schedule slack that allows their start time to change without violating any dependency constraints. In these simple examples, **b** has some slack, and could be scheduled 1 cycle earlier. This schedule slack is preserved and communicated forward to the placer to provide flexibility by allowing moves in time within the slack window. Initially, we produce a tightly packed schedule based on dependency and resource count constraints. In a later stage, we may find that placement or routing is not able to find a solution with this optimistic schedule, and we will iterate to lengthen it. This lengthening will produce more slack and possibly more virtual resources via the unrolling process described in the next section. This increases the chance of a successful place and route, at the cost of latency and/or throughput.

5.1 Modulo Graph Unrolling

Once we have a schedule that meets dependency and resource usage constraints, we need to run placement and routing. However, there is a mismatch between the assumptions for scheduling and the assumptions for standard FPGA place and route algorithms. The scheduler assumes that all resources can be made to do a different operation in each cycle of the II; that is, it has virtualized the resources by a factor of II. Standard FPGA place and route algorithms do not support this type of virtualization. To overcome this difference, SPR unrolls the architecture graph II times, making one copy of the architecture for each cycle of the II. We refer to each cycle within an II as a *phase*. It also re-maps connections with non-zero latencies so that they cross the appropriate number of phases, effectively routing forward in

time. Since we are implementing a modulo schedule, we create a modulo graph by wrapping any connections beyond II phases back around to the beginning. This datapath graph transformation is legal as long as II is less than the depth of the chip’s configuration memory, so we limit the unrolling based on the architecture’s depth. Unrolling the graph turns the CGRA’s time dimension into a third space dimension from the point of view of the placer and router, as shown in Figure 5, allowing standard algorithms to be used. In this figure we see the usual spatial routing on wires and through switch boxes, but registers actually route forward in time to the next cycle, shown with dotted lines.

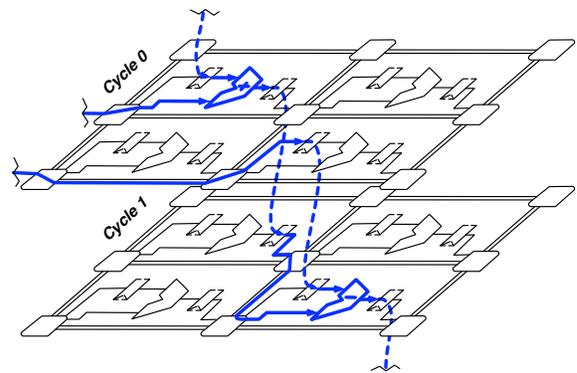


Figure 5: Unrolled datapath graph with mapped dataflow graph.

We maintain information about which unrolled nodes correspond to the same physical device, and the associated phase of the virtual instance. Additionally, all dataflow graph nodes are annotated with their start time and slack from the schedule. This extra information will allow the placement to restrict moves to phases of a device that preserve a legal schedule, but still generate moves in both space and a window of time.

At this point, it is important to note the difference between what we call stateful and stateless devices. For stateless devices, such as an ALU, increasing the II adds another virtual device, because it provides another schedule slot for the physical device. However, this does not work for some devices, such as memories, and we denote those as stateful. An example of this would be a small block RAM, because no matter how much you “unroll” the graph, the same data will be in the same physical memory. With an increase in II, the schedule gets more read and write accesses to the block RAM, but it does not increase the storage capacity of the memory. For these stateful devices, SPR handles the constraints of keeping only one state element in a device, but can virtualize accesses to the device. It groups these accesses so they are mapped to the same physical device.

6. PLACEMENT

Like most FPGA tool flows, SPR’s placer uses Simulated Annealing [8]. When using the Simulated Annealing framework, you must define three key components for your problem: the cooling schedule, move function, and cost function. We chose to adopt the cooling schedule used by VPR [3] because it was shown to work well for FPGAs, and once we have unrolled our architecture, it is very close to a standard FPGA placement problem.

Our move function is more complicated than that for an FPGA. We enforce our scheduling and stateful element constraints through the move function by only generating moves that respect these constraints. We start by choosing a random dataflow node to be moved. We then pick from the set of physical datapath nodes with a compatible type. After that, we choose a random phase from the set of phases in the current schedule slack for this node. As a final check, we ensure that the dataflow node at the destination datapath node is compatible with the phase and type of our current datapath node, and if so we have generated a successful swap move. If not, we keep trying different destination datapath nodes until we find a swap or exhaust all possibilities. In the latter case, we try a different dataflow node and repeat the process. In addition to this simple swap move, we have implemented a more complicated clustering move function, described in Section 6.2. We compare these two methods in the evaluation section.

The last thing that needs to be defined is our cost function. We use a routability driven cost function, with routability estimates defined on a per architecture basis. For the architecture we used in our evaluation, this cost function is the estimated number of muxes used to route a connection with a given amount of clock latency. Once we have a function to estimate the routing cost, the cost of a placement is the sum of the routing cost over all connections in the architecture, plus a penalty for each unroutable connection. These unroutable connections arise because SPR targets fixed frequency devices, and if a route must traverse a large portion of the chip, there will be some forced registering along the way to keep clock frequencies high. If a connection between two operations doesn't have the latency needed to meet the forced register delay constraints, it is marked as "broken." These broken connections incur a penalty cost proportional to the amount of latency that would need to be added to meet the delay constraint.

6.1 Latency Padding

Our initial schedule optimistically assumes only operator computation latency and ignores data movement latency in connections to get the tightest schedule possible. Unfortunately, the placer can't always meet this optimistic schedule because some longer range connections will have forced latency in them due to registering. If there is slack available in the schedule, the placer can shift the slack around so that it is used to handle the forced latency of long wires. However, if there isn't enough slack, it needs to communicate this to the scheduler.

We implemented latency padding to perform this communication. The placer is run to completion, and at the end any connections that are considered unroutable by the architecture specific cost function are candidates for padding. By repeatedly querying the cost function with higher latencies, we can find the minimum amount of extra latency needed before the connection is considered routable. We then add padding that looks like extra delay dependencies to the scheduler, but is treated as slack by the placer. We explored several options for how and where to add padding, which are detailed along with the results in Section 8.4.

The goal of this padding is to directly add slack to problematic areas of the computation. This added padding will affect the scheduling of all down-stream operations and could affect up-stream operations and the II through recurrence

relationships. Thus, we must go all the way back to the scheduling stage and start again. The padding tells the scheduler where to insert more slack, and then the placer uses that extra slack to span the long latency interconnect. The next time around through the placer, the padding may be needed on a different connection due to the random nature of simulated annealing. Fortunately, padding appears as slack to the placer, which can sometimes be moved from the padded connection to where it is needed in the new placement.

6.2 Dynamic Recurrence Clustering

Many architectures group resources into clusters with more flexible and lower latency interconnect. This is found in architectures like HSRA, MorphoSys, and Tartan. As we mentioned before, the largest recurrence loop in an application sets a lower bound on its II. This means that these recurrence loops are effectively our critical path. In order to keep our throughput high, we want to make sure any critical loops in our application take advantage of the faster interconnect offered by clustered nodes in the architecture.

The basic idea behind our clustering is that when attempting to move an operation from one cluster to another, it might need to move nodes from the same recurrence loop to the new cluster as well. This is to avoid higher II's due to inter-cluster communication in the critical loop. For our clustering algorithm, we start out by marking all edges in recurrence loops as clustering edges. Then, when the placer is generating an inter-cluster move, it checks to see which neighboring nodes should be moved to the new cluster as well. If the clustering edges to any nodes in this cluster would become unroutable, we attempt to include those nodes in the move to the destination cluster. Of course, the included neighbor's neighbors may be part of the loop and need to be moved as well, so we repeat this process recursively until we have added no new nodes. We limit our consideration of neighbors to nodes that start in the same cluster, so the biggest move we could generate this way is a full swap of two clusters. The key here is that we only cluster when a connection would otherwise end up being unroutable, which would result in latency padding and an increase in II. This way, we only cluster what is necessary, and we can still spread recurrence loops with enough slack in them across clusters.

7. ROUTING

Routing the signals between the operators in a scheduled and placed dataflow graph requires finding paths containing zero or more registers. To accomplish this, SPR uses QuickRoute [10], a fast, heuristic algorithm that solves the pipelined routing problem.

By using PathFinder [11] with QuickRoute, SPR has a framework that negotiates resources conflicts. As a general conflict solver, PathFinder can be applied to a range of problems that can be framed as a negotiation. Originally, PathFinder was used to optimize global routing by negotiating wire congestion. We show that the same framework, given the correct congestion metric, can be used to negotiate between other conflicts such as those encountered when trying to make use of a statically configured resource in a time multiplexed system. When applied to our unrolled architecture graphs, the original PathFinder will work unmodified for dynamically configurable resources, where the configu-

ration can be changed on every tick of the clock. Many reconfigurable systems, such as RaPiD [6], use more area-efficient and power-efficient interconnect for portions of the system that are set up statically before a computation. We have extended PathFinder to allow sharing of both static and dynamic muxes between signals in different clock cycles. To see what this sharing means, consider the routes shown in Figure 6 across different phases of the same mux.

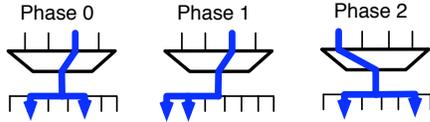


Figure 6: Routes across different phases of a mux.

If this is a dynamic mux, all three signals can share this mux. A static mux would allow sharing of the signals in the first two phases because they share the same input, even though their destinations may be different. The signal attempting to use the mux in Phase 2 would need to be re-routed. Without enhancement, PathFinder is unable to support this type of sharing in mixed static/dynamic architectures.

7.1 Control-based PathFinder

In our enhanced version of PathFinder, static and dynamic resources appear the same to the signal level Quick-Route algorithm. The difference lies in the computation of the congestion costs during the congestion negotiation. The problem with using standard PathFinder is that we have created virtual copies of static muxes that PathFinder sees as completely disjoint routing resources. Even though all virtual copies of a static mux need to have the same input configuration to have a valid mapping, PathFinder will obviously route through different inputs in different phases. On the other hand, dynamic muxes have no constraints on the settings between phases, and so by simply unrolling the graph, the original PathFinder formulation supports dynamic muxes. A straightforward PathFinder extension for supporting static muxes is to simply allow only one signal to ever use a particular static mux, effectively not unrolling it. This can be accomplished by summing the signal counts across the phases. This will only allow 1 signal to ever use a static resource. However, we would instead like to put the static resource into the most useful setting, and share that setting amongst different signals in different phases.

Two observations lead us to our new PathFinder formulation which can time-multiplex signals across static muxes. The first is a simple optimization to PathFinder. We observed that limiting PathFinder to negotiating between signals for the use of a mux output port is equivalent to PathFinder negotiating for the use of all the wire segments and registers driven by that mux. This is because by choosing which signal will occupy the output port of a mux, we have implicitly chosen that the same signal will occupy all wire segments connected to the port, either directly or indirectly through a series of registers. This lets us dramatically reduce the number of resources we have to track PathFinder negotiation information for.

The second observation is that when choosing which signal will occupy the output port of the mux, you are really only choosing which input of the mux should be connected

to the output, i.e. what should the values of the configuration bits for that mux be. This makes the relationship between static and dynamic routing resources more apparent. In the dynamic case, there is a separate configuration available for each phase of the II, so a different input can be chosen by the router in every phase. In the static case, the router can only choose to have one input used for the life of the program. Thus, PathFinder must negotiate for use of the shared configuration bits among unrolled instances of a static mux.

To allow for this new negotiation, we now have two different kinds of congestion. The first is the original PathFinder notion of *signal* congestion: two electrical signals cannot be sent along the mux output wire at the same time. The second is *control* congestion: two signals using a statically configured mux cannot require two different configurations in different phases, but both can use the output wire in different phases.

In the original PathFinder, congestion led to two types of cost: the immediate sharing cost and the history sharing cost. Now that we have two different types of congestion, this leads to four costs to be monitored. We will begin with the immediate sharing costs. The immediate sharing cost for signal congestion remains unchanged from PathFinder, and is the excess number of signals attempting to use a mux in a given phase.

The immediate sharing cost for control congestion is the excess number of configurations used by a mux across all phases. For a dynamic mux, which can have a different control setting in each phase, this will always be zero. However, for a static mux, only one setting is available, so any excess settings needed by signals add to the immediate sharing cost. An example of computing the congestion for the different types of sharing on a 6 input static mux with an II of 4 is depicted in Figure 7.

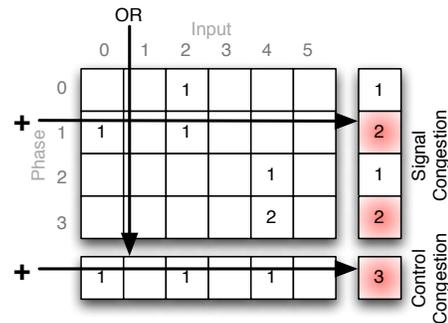


Figure 7: Congestion calculation from signal usage.

The top half of the diagram illustrates the original PathFinder signal congestion costs for each unrolled instance of the mux. By adding up the number of signals using the inputs, we see how many will be on the output. This leads to signal congestion on the phase 1 mux and the phase 3 mux because there are two signals trying to use the mux simultaneously.

The bottom half of our diagram illustrates our new control congestion for a static mux. For each input, we see that input is used in any phase, doing a logical OR across phases. Then we sum across these OR'd values to see how many configurations are used across all phases, obtaining 3

in this example. Because this is a static mux, a maximum uncongested value is 1 input, so the value of 3 here indicates control congestion. Using both types of congestion in PathFinder is straightforward, as PathFinder simply needs to iterate until there is no congestion of either type.

Now let’s look at our new history sharing costs. Again, the history sharing cost for signal congestion from PathFinder is unchanged. The history cost for control congestion is a little more subtle. For a fully dynamic mux, there is no control congestion history to maintain.

For a single static mux, we would like to make the signals using the mux in different phases use the same inputs through PathFinder negotiation. The history cost update must be computed by looking at the mapped signals across all phases of a mux with control congestion. Pseudo code for this update is given in Figure 8. For each signal using one of the unrolled muxes, there are two basic increases to the control cost. The first is an increase to the cost of any input other than the one the signal is currently using, represented by the condition `sig.input!=input`. The second is an increase to the cost of the current input and phase that a signal is using, represented by the condition `sig.input==input && sig.phase==phase`.

```

updateControlHistory() {
  foreach sig of signalsOnMux {
    foreach phase of II{
      foreach input of mux{
        if(sig.input!=input ||
           (sig.input==input &&
            sig.phase==phase))
          historyControlCost[phase][input]++;
      }
    }
  }
}

```

Figure 8: Control congestion history pseudo-code.

The reasons for these two increases will be discussed in the context of an example where two signals, A and B, are using a 6 input static mux with an II of 4. The resulting history cost increases are depicted in Figure 9. Inputs with history cost increases from A are shaded in red(gray) and increases from B are shaded in blue(black). Inputs that are shaded by both will have their cost increased by twice as much as those shaded by a single color.

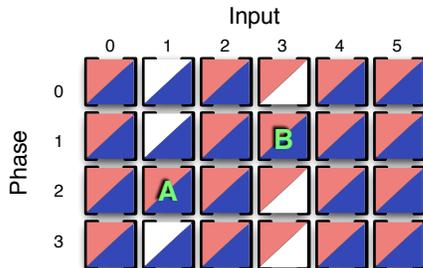


Figure 9: History updates for static control congestion.

For the condition `sig.input!=input`, each signal increases the cost of using a configuration (or input) other than its own. This makes the inputs with signals already using them in a different phase relatively less expensive in future itera-

tions. As the cost increases, either signal A or B could find a cheaper alternate route through a different mux or an input compatible with the other signal, for example A finding an alternate route that uses input 3.

For the condition `sig.input == input && sig.phase == phase`, each of A and B increase the cost of their input in the phase they use it. To see why signals increase the cost of their current input, we will consider what would happen if they didn’t. Suppose the depicted congestion of A and B is the only congestion in the current routing. At this point, the paths A and B are taking are the least cost paths from their sources to their respective sinks. When each signal only increases the cost for inputs it is not using, the cost of the currently used inputs will increase at the same rate. The cost of all others will increase at twice that rate. Since the currently used inputs are penalized by the same amount, neither signal has incentive to take a longer path to use compatible inputs. The only way this congestion will be resolved is if A or B move to a different mux altogether, not through both using the same input.

When A and B *do* increase the cost of their input in the phase they use it, each signal will notice that “the grass is greener” on the input the other signal is using. Whichever signal has a cheaper alternate route to the other input will eventually move first, and the congestion will be resolved, sharing the same input of the static mux in different phases. In this way, we can have several signals sharing our static resources cooperatively.

8. EVALUATION

To evaluate our enhancements, and to provide a tool for the Mosaic architecture exploration [2], we implemented SPR in Java using the schedule, place, and route algorithms described in the previous sections. We used 8 benchmarks that constitute a set of algorithms with loop-level parallelism typically seen in embedded signal processing and scientific computing applications. With only a few benchmarks, we used a two-tailed paired T-test with $p = .1$ for establishing significance in our results. Our architectures are defined as structural Verilog suitable for simulation, generated by the Mosaic Architecture Generator plug-in to the Electric [1] VLSI Design System.

8.1 Architecture

For our experiments, we targeted a 2-D grid CGRA architecture inspired by island-style FPGAs and grid style CGRAs. It is made up of a 2-D array of clusters containing 4 ALUs, 4in/4out stream accessors, 4 structures for holding configured constants, and 2 local block RAMs. Internally, devices in a cluster are connected by a cross-bar. The clusters are connected to each other with a pipelined grid interconnect where we can vary the number of static and dynamic channels. The grid interconnect employs Wilton style switchboxes [20]. Unless otherwise stated, we use an architecture made up of 16 clusters with a 16 track interconnect. In total, our test architecture contains 288 functional units, though we expect the algorithms to scale to thousands of functional units on the basis of their lineage from existing VLIW and FPGA algorithms. SPR will eventually support pipelined functional units which take multiple clock cycles for computation, but for our simple test architecture we assume single cycle operations.

8.2 Benchmarks

The benchmarks were written in Macah with the main loops designated as kernels. These kernels were translated by the Macah compiler into a dataflow graph [5]. The dataflow graph nodes are primitive operations. The nets are either routable connections or dependency constraints, such as sequential memory accesses, that must be respected by the scheduler. A tech mapper translates compiler specific nodes into SPR readable generics and maps from dataflow graph node types to devices in the architecture.

The benchmarks include three simple signal processing kernels: *fir*, *convolution*, and *matrix multiply*, and five more complex kernels from the multimedia and scientific computing space: *K-Means Clustering*, *Smith-Waterman*, *Matched Filter*, *CORDIC* and *Heuristic Block Motion Estimation*.

Table 3: Summary of Benchmarks

Kernel	Nodes	Nets	II	Latency
Motion Estimation	196	387	5	29
Smith-Waterman	214	468	9	58
2-D Convolution	518	1037	5	58
Matched Filter	406	801	4	48
Matrix Multiply	212	434	5	14
CORDIC	157	336	3	33
K-Means Clust.	449	998	7	30
Blocked FIR	255	506	3	46

Relevant statistics for our benchmarks are given in Table 3. The number of nodes and nets quantifies the size of our benchmarks. Most of our benchmarks have adjustable sizes. For example, the number of coefficients used in a FIR can be changed. The minimum II is a lower bound set by both the largest recurrence cycle (RecMinII) and the number of resources needed by the application (ResMinII). Where possible, we scaled our benchmarks up as far as they would go without increasing the ResMinII past the RecMinII. When benchmark scaling increased the RecMinII faster than the ResMinII, we scaled to just past 200 nodes instead. The value for II and latency given in the table are the minimum schedulable values for our 16 cluster architecture. These II’s will also be increased as needed by the placement and routing stages to achieve a successful mapping, and the increase above this baseline is what we measure in evaluating our latency padding and clustering algorithms.

8.3 Graceful Scaling

One problem with using a system like an FPGA for an accelerator is that if your computation doesn’t fit on the particular chip you have, it won’t run without adjustment of the application. Because SPR is designed for use in time multiplexed systems, more virtual hardware resources can be made available at the cost of slower execution. An example is shown in Figure 10. As you can see, as we map the same application to architectures with fewer resources, we still get a valid mapping, but the II’s are increased to make more resources available, with a corresponding decrease in throughput. Similarly, we can put larger applications on the same size architecture at the cost of throughput. In this case, we increase the application size by using more coefficients in the FIR. The 40 coefficient FIR corresponds to the size used in Table 3, so it consists of 255 nodes and 506 nets.

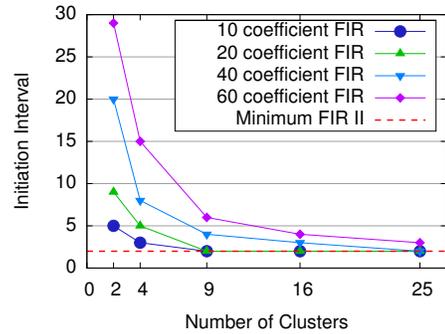


Figure 10: Scaling across architecture sizes.

8.4 Latency Padding Effects

Latency padding is an educated guess as to where more latency in the schedule will aid in placement and routing. It isn’t an exact guess, because adding latency for a given placement will result in a new schedule and a new placement, which may have different latency needs. Additionally, because the placer works in both time and space, the latency may be moved by the placer to make better use of it.

Given a “broken” connection (a single connection that is unroutable due to forced latency constraints), there are several possibilities for adding latency that may fix it on the next scheduling and placement pass. One possibility is to add padding latency only to the connection that is broken, effectively spreading out the operations on both sides of the connection. We will call this *connection padding*.

Another option that works well in the IMS [16] framework is to pad by reserving more time for the source operation. This has the advantage that there will be slack on all outputs for the operation, which means the operation will be able to be moved in time easily by shifting slack from all outputs to all inputs. This gives the placer a little more flexibility in the next pass, but there is potentially more latency added to the system as a whole. We call this *operation padding*.

Once you choose what to pad, you also must choose how much to pad. Even though we know how much latency is needed to make the connection routable, that may not be the best amount to pad by. If there are several broken connections, adding padding to only one may fix all connections if the placer can move the slack to a common ancestor of the sinks. Another possibility is that adding less than the full amount of padding may work because in the next pass, operations that occur at the same time could be completely changed, affecting the placement and the routing.

To get a minimal amount of padding, one could add only 1 cycle of latency at a time and re-run the schedule and placement until it succeeds. These extra iterations cost extra compilation time. To minimize the number of extra iterations, one could pad by the full amount needed to meet the interconnect latency constraints. In our case, we are looking at applications where throughput matters more than the overall latency. Thus, it should be fine to pad by the full amount where the extra latency only affects the overall program latency, not the throughput. However, padding a recurrence cycle could increase the II and decrease our throughput. Therefore, conservative padding could be worth extra compiler run-time for recurrence cycle connections.

We must decide between padding connections or operators, and we must choose either conservative or full padding

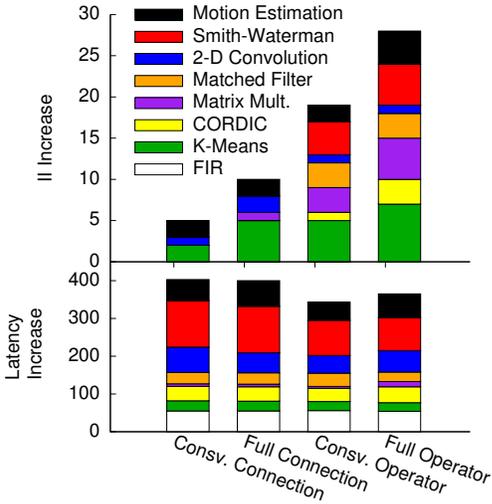


Figure 11: II and Latency effects across different padding settings.

in recurrence cycles. The results of experiments run using these 4 possibilities are shown in Figure 11. Note that some applications don't show up in the bars, and in this case, the II was not increased over the baseline given in Table 3.

Connection based padding is the best option for keeping throughput high, with the conservative padding producing slightly better results, as expected. However, if latency is your primary concern, then conservative operator based padding is the best solution.

8.5 Dynamic Recurrence Clustering

To test the effects of our dynamic recurrence clustering, we ran our tool with it both on and off. The results are shown in Figure 12.

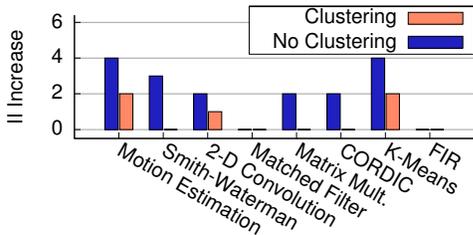


Figure 12: Effects of Clustering on II.

The clustering allowed us to achieve improved IIs, translating into improved throughput, in six of our benchmarks. For the remaining two, the results were the same for both methods. Averaged across all benchmarks, this translates into a significant improvement of 1.3x in throughput. The lack of bars for the Clustering case in the chart indicates that for most of the benchmarks, we were able to achieve the minimum schedulable II when clustering was turned on.

8.6 Static/Dynamic Interconnect

To exercise our new static congestion negotiation algorithm, we mapped our benchmarks on two different architectures, one with a fully dynamic global interconnect, and one with a fully static global interconnect (with dynamic

cluster crossbars). To measure sharing, we count the number of signals mapped to a physical mux in different phases.

The fully dynamic run gives us our baseline for the amount of sharing that would happen in the most flexible system possible, given the actual application we are mapping. With a fully dynamic interconnect, there are no constraints put on signals sharing a mux in different phases. It is *sharing neutral*, and should neither encourage nor discourage sharing. On the other hand, the static interconnect has two forces at work to perturb the amount of sharing. They both originate from the constraint that two signals who share a mux in different phases must use the same input. This can work to lower the amount of sharing in routing rich architectures, because when there is a conflict, it is easier for one signal to use a slightly longer but alternate route through empty muxes than to use the same input as the competing signal. As the routing resources become more scarce, there will be fewer empty muxes to use and some sharing will be required. Once you have negotiated to use the same input for one mux, that means both signals are also using the same upstream mux. In fact, this holds transitively, so if a mux is shared, the same amount of sharing will be forced on all upstream static muxes until a dynamic mux is reached. This will tend to increase sharing.

We found the baseline dynamic sharing to be an average of 1.35 signals/mux, for our benchmarks and a fixed channel width of 16. As a check to ensure this number is reasonable, we calculated expected value of sharing given the II, global interconnect utilization and assuming a uniform distribution, and found it to be 1.21 signals/mux. The measurements should be slightly higher because the actual distribution is not uniform. The fully static run shows that our algorithm successfully allows sharing of statically configured interconnect, demonstrating an impressive 1.32 signals/mux, which is not a significant difference from the dynamic case.

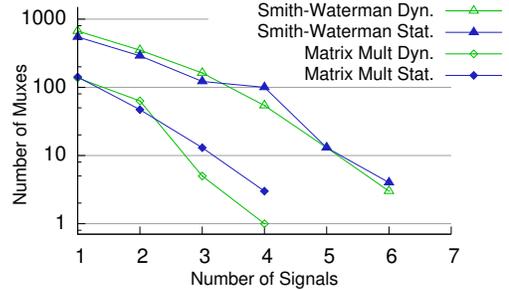


Figure 13: Dynamic and static sharing using the minimum routable static channel width.

If we examine sharing at the stress case of minimum routable channel width, we begin to see more differentiation between static and dynamic resources. In this case, we expect higher utilization to lead to higher amounts of sharing. The tighter resources and higher utilization showed an increase in sharing in the static interconnect to 1.52. As a comparison, we also ran our benchmarks with a dynamic interconnect sized to the minimum static channel width. Here we find the dynamic has significantly lower sharing at 1.47. We believe that in the stress case, the upstream chaining is causing the higher sharing when using static resources. A histogram of the number of signals sharing a mux is plotted for two applications on both static and dynamic interconnect in Figure

13. The two applications shown provide a rough upper and lower bound on the sharing across our applications.

Finally, we found the minimum routable channel width for fully dynamic interconnect, static interconnect using our sharing algorithm, and static interconnect disallowing sharing. The average minimum channel width when using a dynamic interconnect is 7.13 channels. Using a static interconnect with no sharing increased the width all the way to 20.1 channels. By allowing sharing, we reduce this greatly to 10.5 channels on a fully static interconnect. This means we gain all of the power and area savings of a statically configured interconnect while reducing the associated channel width to .5x of what we need without sharing.

9. CONCLUSION

In this paper, we described SPR, an efficient and flexible architecture-adaptive mapping application for exploring the CGRA architecture space. We found our latency padding technique successful in bridging the gap between the VLIW style scheduler and FPGA style placer to meet the constraints of a fixed frequency device with configurable interconnect. After evaluating several methods of padding, we found that conservative padding on a per-unroutable connection basis achieved the best throughput. We show an average improvement of 1.3x in throughput of mapped designs by using a new dynamic clustering method during placement. Finally, we showed that we can effectively share non-multiplexed interconnect in a time-multiplexed system using an enhancement to the PathFinder [11] algorithm. Our results indicate that SPR is capable of mapping a range of applications while supporting unique features of CGRAs.

10. FUTURE WORK

SPR currently makes some limiting assumptions we will lift moving forward. We plan to implement a more timing driven approach to scheduling, placement and routing based off of the work in Armada [7]. We are also planning to incorporate some control schemes that are more advanced than a modulo scheduler, such as the nested-loop controller of RaPiD [6], or the ability to create a program counter out of logic in MATRIX [14]. As the Mosaic project moves forward, SPR's flexibility will be used to assess the architectural trade-offs of dynamic and static interconnects to find the proper mix of resources for target applications.

11. ACKNOWLEDGEMENTS

Department of Energy grant #DE-FG52-06NA27507, and NSF grants #CCF0426147 and #CCF0702621 supported this work. An NSEDG Fellowship supported Allan Carroll.

12. REFERENCES

- [1] Electric VLSI Design System. Sun Microsystems and Static Free Software. <http://www.staticfreesoft.com/>.
- [2] Mosaic Research Group. <http://www.cs.washington.edu/research/lis/mosaic/>.
- [3] 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.
- [4] A. Carroll and C. Ebeling. Reducing the Space Complexity of Pipelined Routing Using Modified Range Encoding. In *International Conference on Field-Programmable Logic and Applications*, 2006.
- [5] A. Carroll, S. Friedman, B. Van Essen, A. Wood, B. Ylvisaker, C. Ebeling, and S. Hauck. Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency. Technical report, Department of Energy NA-22 University Information Technical Interchange Review Meeting, 2007.
- [6] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *International Workshop on Field-Programmable Logic and Applications*, pages 126–135, 1996.
- [7] K. Eguro and S. Hauck. Armada: Timing-driven Pipeline-aware Routing for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 169–178, 2006.
- [8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.
- [9] J.-e. Lee, K. Choi, and N. Dutt. Compilation Approach for Coarse-grained Reconfigurable Architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003.
- [10] S. Li and C. Ebeling. QuickRoute: A Fast Routing Algorithm for Pipelined Architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 73–80, 2004.
- [11] 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.
- [12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A Retargetable Compiler for Coarse-grained Reconfigurable Architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 166–173, 2002.
- [13] 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*, volume 2778, pages 61–70, 2003.
- [14] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 157–166, 1996.
- [15] M. Mishra and S. C. Goldstein. Virtualization on the Tartan Reconfigurable Architecture. In *International Conference on Field-Programmable Logic and Applications*, pages 323–330, 2007.
- [16] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *International Symposium on Microarchitecture*, pages 63–74, 1994.
- [17] A. Sharma, S. Hauck, and C. Ebeling. Architecture-adaptive Routability-driven Placement for FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 427–432, 2005.
- [18] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.
- [19] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzyniek, and A. DeHon. HSRA: High-speed, Hierarchical Synchronous Reconfigurable Array. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 125–134, 1999.
- [20] S. J. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.