Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency[*]

Allan Carroll,[†] Stephen Friedman,[†] Brian Van Essen,[†] Aaron Wood,[‡] Benjamin Ylvisaker,[†] Carl Ebeling,[†] and Scott Hauck[‡]

[†]Department of Computer Science and Engineering
University of Washington

[‡]Department of Electrical Engineering
University of Washington

Abstract

Coarse-grained reconfigurable architectures (CGRAs) have the potential to offer performance approaching an ASIC with the flexibility, within an application domain, similar to a digital signal processor. In the past, coarse-grained reconfigurable architectures have been encumbered by challenging programming models that are either too far removed from the hardware to offer reasonable performance or bury the programmer in the minutiae of hardware specification. Additionally, the ratio of performance to power hasn't been compelling enough to overcome the hurdles of the programming model to drive adoption.

The goal of our research is to improve the power efficiency of a CGRA at an architectural level, with respect to a traditional island-style FPGA. Additionally, we are continuing previous research into a unified mapping tool that simplifies the scheduling, placement, and routing of an application onto a CGRA.

1. Introduction

Interest in spatial computing is being revitalized by the limitations of sequential processors to deliver performance improvements that are commensurate with the increase in silicon density provided by Moore's Law. Spatial computing is a technique that commonly uses a large number of simple parallel processing elements, which operate concurrently, to execute a single application or application kernel. Examples of spatial computing systems are dedicated hardware, commonly in the form of application specific integrated circuits (ASICs), or in recent years, configurable hardware such as field programmable gate arrays (FPGAs).

Spatial computing, as seen in both ASICs and FPGAs, offers great power and performance advantages over sequential processors, *e.g.* general purpose processors and digital signal processors (DSPs), for certain classes of applications [1] such as digital signal processing (DSP) and scientific computing. ASICs typically provide the best performance and power characteristics, at the expense of a very costly and long development cycle. FPGAs provide the low cost of an off the shelf component yet they have, relatively speaking, poor area, power and performance characteristics due to the overhead of providing a flexible computing substrate.

These inherent limitations of both ASICs and FPGAs drive the development of a new class of spatial computing, the coarse-grained reconfigurable architecture (CGRA). The goal of a CGRA is to have the power and performance advantages of an ASIC as well as the cost and flexibility of an FPGA. To achieve these goals, our CGRA is designed for datapath computation, rather than general purpose computation. We are targeting the application domain encompassing DSP and scientific computing. By narrowing the scope of computation that is supported we can reduce the overhead associated with providing flexibility in a computing substrate.

1.1. Challenges of configurable computing

Configurable computing is a subset of spatial computing that provides computation and communication resources that can be configured, or programmed, at load time or run time. FPGAs are a common example of a load time configurable computing system, while CGRAs such as RaPiD [2] and PipeRench [3] are examples of run time configurable computing systems.

In mainstream FGPAs, shown in Figure 1, the computing substrate is composed of a sea of 4-input look-up tables (4-LUTs) that communicate via an island-style

[*]This material is based upon work supported by the Department of Energy under Grant No. DE-FG52-06NA27507.

[†]A. Carroll, S. Friedman, B. Van Essen, B. Ylvisaker, and C. Ebeling are with the Department of Computer Science & Engineering, University of Washington, Seattle, WA (e-mail: {allanca, sfriedma, vanessen, ben8, ebeling}@cs.washington.edu).

[‡]A. Wood and S. Hauck are with the Department of Electrical Engineering, University of Washington, Seattle, WA (e-mail: {arw82, hauck}@ee.washington.edu).

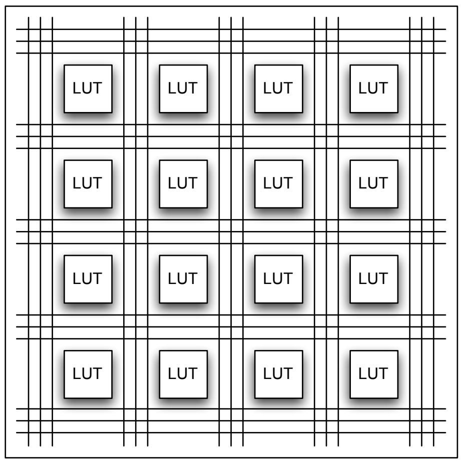


Figure 1. FPGA Block Diagram

interconnect topology. The functionality of an FPGA is specified by a configuration file, which is loaded at power-up, and controls the operation of the system on a per LUT and per routing channel basis. FPGAs traditionally have focused on supporting general purpose bitwise computation instead of word-wide datapath computation. Recent developments in FPGA architectures have added monolithic, word-wide multiplier and block memory structures to the otherwise bit-oriented fabric. Despite these improvements Kuon and Rose [4] determined that an application targeted to an FPGA typically suffers a 21-40x overhead in area, a 3-4x delay in critical path, and a $\sim$12x reduction in power efficiency versus an ASIC.

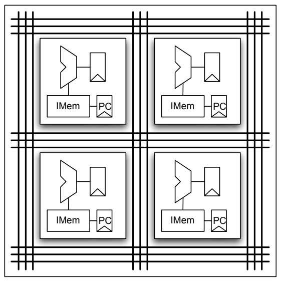
The goal of our CGRA is to reduce these gaps, on average, by an order of magnitude. We will achieve this goal by designing an architecture that is focused on datapath computations for DSP and scientific computing. This means that the bulk of the computation and communication resources will be scaled for multi-bit data words. Examples of potential coarse-grained architectures are shown in Figure 2(a) and 2(b). This transition from bitwise to word-wide structures should significantly reduce the overhead of the configuration logic. One of the challenges in this effort is that only including a word-wide datapath is insufficient for many real applications, and so a bitwise control network must be judiciously overlaid on the datapath to provide a good balance of control versus overhead.

Another challenge common to spatial computers is the management of computation and communication resources, and how they are exposed to the programmer. In the simplest form, a programmer may know that an architecture contains *P* processing elements and *C* communication channels and is responsible for ensuring that an application fits within the available resources. Another alternative, exemplified in general purpose processors, is to completely abstract away the quantity of these resources and through time-multiplexing and virtualization to provide the illusion of "virtually unbounded" resources. The goal for our CGRA is to strike a middle ground, where a programmer doesn't have to know the exact number of any given resource and that between the tool chain and the architecture there are several features that make it possible to efficiently map a "larger" application onto a "smaller", fixed sized, spatial computer.

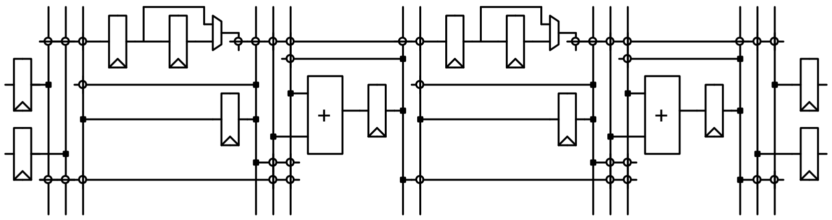
For our spatial computers, CGRAs, the runtime configurable (reconfigurable) nature of the architecture is a method for time-multiplexing some of the resources within the computing substrate, which enables a degree of resource abstraction for the programmer. However, time-multiplexing leads to the very challenging problem of programming an architecture in both time and space.

To address the challenges of programming a CGRA in both time and space, we have developed a unified schedule, place, and route tool called SPR. Using iterative feedback for integrated scheduling and placement, SPR can automatically time-multiplex an application, represented as a dataflow graph, and place it on the spatial computing substrate, thus simultaneously mapping in both space and time. In broad strokes, SPR creates a schedule for a dataflow graph that represents both an architecture's resource constraints and an application's true data dependencies. To start the process, SPR identifies the initial schedule, which is based only on true data dependencies. SPR then attempts to map that schedule onto an architecture, represented as a datapath graph, and, as it encounters resource constraints (aka structural hazards), it will extend the schedule and iterate. Once a legal schedule and placement are found the dataflow graph is routed onto the datapath graph, and as structural hazards are identified the system reschedules and replaces as necessary.

SPR automatically and iteratively searches for legal mappings of an application to an architecture. This decreases the effort the programmer must make to get a good application mapping. It also enables the performance of the application to degrade gracefully when SPR must increase the application runtime in order to use fewer device resources. History has shown that the difficulties encountered by an application's programmer directly influence, and impede, the adoption and effectiveness of an architecture. Therefore the development of this tool is paramount to the usability and success of a CGRA.



(a) Island style coarse-grained spatial computer



(b) Pipelined coarse-grained reconfigurable array

Figure 2. CGRA Block Diagrams

2. Exploring Coarse-grained Architectures

To create a CGRA that is focused on datapath computations for a particular application domain is a balancing act akin to designing an ASIC and a FPGA simultaneously. Narrowing the application domain significantly makes the design of the CGRA very much like that of a programmable ASIC. Widening the application domain requires a more flexible datapath that requires more configurable overhead and has less overall efficiency compared to an FPGA. Therefore, our research focuses on identifying key architectural features that substantially influence power, programmability, and performance. Having isolated these design features and characterized the nature of an application domain, we can construct an architecture that is efficient for any application in the domain, not just one specific applications or kernel.

To understand the trade-offs between architectural choices we need a mechanism to efficiently explore the CGRA design space. Our tool chain that will provide this flexibility is illustrated in Figure 3. The key components of the system are the architecture generator, the system power analyzer, the unified mapper (which performs scheduling, placement, and routing to map an application onto an architecture), and finally a logic synthesis tool that creates dataflow graphs from applications. Details of each of these tools is provided in Sections 3, 4, 5, and 6 respectively.

3. Generating Coarse-grained Architectures

Unlike the design of a general purpose processor, DSP, or FPGA, the specification and design of a CGRA is heavily influenced by the target application domain. For example, by its very nature, the data plane of a CGRA is designed to operate on multi-bit operands. As indicated previously, for efficiency, a separate single-bit control plane is used to avoid wasting word-wide computation resources for computing single bit control signals. This partitioning in functionality is one of several resource balancing challenges that is affected by the choice of application domain. Other challenges in resource balancing include choosing the right mix of operators, the right mix of computation and storage elements, and the right mix of heterogeneous and homogeneous processing elements.

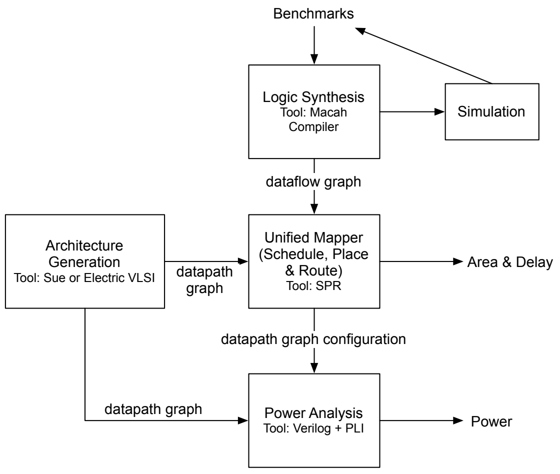


Figure 3. CGRA Architecture Tool Chain

The goal of our architecture generation and exploration is to identify and isolate the impact different architectural features have, in terms of power efficiency and performance, for a range of applications. Using the results from this study and sample applications from our application domain of choice, we can construct an efficient coarse-grained architecture.

3.1. CGRA Power advantages

Studies of the power consumption within an FPGA, as shown by Poon et al. [5], identify that 50% to 60% of the energy of an FPGA is dissipated in the routing fabric, 20% to 40% in the logic blocks and 5% to 40% in the clock network. Using these number as a guideline, we see that the interconnect offers the greatest chance for energy optimization, followed by the logic blocks. One key aspect in the transition from fine-grained to coarse-grained configurable architectures is the overhead associated with configuration control logic. Unfortunately, the energy consumed by the configuration logic in a standard FPGA was not reported in [5], making direct comparison impossible. Therefore, improvements in the CGRA will be measured indirectly at the system level.

By abstracting the effects of a given silicon manufacturing technology, custom layout, or even the specifics of implementing a power efficient arithmetic unit we can focus on architectural features that provide a significant power advantage. The primary source of architectural power efficiency is the transition from a bitwise to word-wide datapath, which can be further refined into categories for interconnect, arithmetic and logic, and configuration overhead.

Some options for improving the efficiency of the interconnect within the data plane are:

1. Reducing the number of pair-wise routing options, *i.e.* connectivity

2. Reducing interconnect switching complexity by bundling wires into buses

3. Reducing the average interconnect length by inserting pipelining registers

4. Using a more power efficient numerical representation, such as signed magnitude or Gray codes, for communication as shown in [6].

Some options for improving the efficiency of the arithmetic and logic resources within the data plane are:

1. Using dedicated, atomic, resources for computation

2. Pipelining arithmetic units

3. Reduced resources for intra-word communication in arithmetic units, *e.g.* carry-chains or conditional flags.

Reduction in the overhead of the configuration logic for the data plane can be achieved by:

1. Configuring buses rather than individual wires

2. Sharing configuration bits across larger compute elements

3. An overall reduction in total number of possible configurations and thus a reduction in the configuration state size

Making an architecture more coarse-grained means that computation is done via dedicated adders, multipliers, etc. rather than constructing such units from more primitive functional units such as a 4-LUT. While a dedicated arithmetic unit is guaranteed to be more efficient than one composed of 4-LUTs, finding a balance of dedicated resources that match the flexibility of the 4-LUTs is difficult, and the overhead of multiple dedicated resources could easily outweigh the individual advantages of each resource.

One common use for CGRAs is to accelerate the computationally intensive kernel(s) of a larger application. These kernels are typically inner loops of algorithms or a set of nested loops. Frequently, it is possible to pipeline these kernels, thus exploiting the application's existing parallelism and increasing its performance. If the application domain of a CGRA is rich with pipelinable kernels then it is advantageous to have a general bias towards a specific flow of computation and data in the datapath and to include dedicated resources for pipelining in the interconnect.

The control plane of a CGRA plays a similar role to the general purpose spatial computing fabric of an FPGA. It is composed of bitwise logic and communication resources, is flexible and highly connected. Given these requirements, it is likely that the control plane's architecture will be similar to a standard FPGA's architecture. From a power efficiency standpoint, the control plane will perform similarly, and thus provide no advantage over a standard FPGA. However, the control plane will be only one portion of a CGRA making its contribution to the energy overhead smaller.

3.2. Evaluation and Analysis of CGRAs

The quality of a CGRA can be rated along several metrics, such as routability, bandwidth, latency, and Joules per datum. For example, our first study focuses on power trade-offs for different interconnect architecture topologies and control mechanisms. Different schemes for interconnect control range from configured, to statically scheduled, to dynamically scheduled. Configured interconnect is determined at CGRA load time, and should provide the lowest static and dynamic power consumption. Statically scheduled interconnect requires a "program" that sequences the configuration of the interconnect throughout the course of an applications execution. Statically scheduled interconnect will probably consume more power per interconnect channel, but will probably require less channel capacity than a strictly configured interconnect. Dynamically sched-

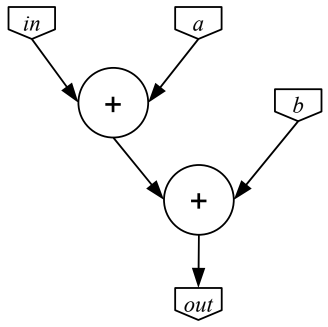


Figure 4. Example of a simple dataflow graph.

uled interconnect probably offers the greatest utilization for a given channel capacity, but will probably have the highest power consumption per channel. Given these dimensions, the optimal interconnect for a given application domain will probably be some mix of configured, statically scheduled and dynamic interconnect. The point of our study is to evaluate this space and produce a figure of merit, in terms of energy consumed per application, for these trade-offs.

3.3. Building CGRAs

One standard method of specifying an architecture is using a hardware description language such as Verilog or VHDL. However, we have found that one invaluable debugging technique for CGRAs is to visualize how an application is mapped to a given architecture, represented by a datapath graph, in both space and time. Therefore we have put considerable effort into making visualizable datapath graphs. To facilitate visualization we use a schematic editor, such as SUE or Electric VLSI, and Verilog to create both logical and geometric information for each CGRA. To illustrate the benefit of visualization, Figure 5 shows an example of how a small dataflow graph, Figure 4, was mapped to the pipelined CGRA example from Figure 2(b) over the course of three cycles. The solid thin black lines represent the actual datapath of the pipelined CGRA from Figure 2(b). The dashed red lines show how data is communicated, through registers, from cycle to cycle. The thick blue lines show the routing resources used by the application.

4. Power Modeling

Traditionally, power modeling of a digital design in CMOS technology focused only on the dynamic power contribution of the system, as the static power contribution was minimal. However, advances in modern VLSI CMOS processes have dramatically increased the leakage power of transistors. This shifted the balance of power consumption to an almost even distribution between static and dynamic sources. The shifting balance between static and dynamic sources creates a new inflection point between total spatial capacity and flexibility of time-multiplexing existing resources.

Currently there is no standard method for performing power analysis of a digital design at the architecture level. One possible approach is to use tools such as PowerMill, on a complete circuit layout, to estimate power. Previous research efforts into power modeling of at the architectural level of complex systems are the Wattch project [7] and the FPGA survey by Poon et al [5]. Wattch provides a operation-centric, empirical study of modern microprocessors, following the activity of each macroscopic stage of the processors pipeline while simulating the execution of some application. The survey by Poon et al. provides an analytical exploration of a Xilinx FPGA that was independent of a given application or data set. To perform power analysis for a CGRA at the architectural level, we are using Poon's methodology and results as a baseline for our model. We are modifying it to reflect the coarse-grained nature of the CGRA, and, similarly to Wattch, coupling it with an operation-centric, empirical analysis for a suite of applications.

Research by Chandrakasan and Brodersen [6] has shown that the dynamic power consumption of arithmetic units can vary substantially based not only upon the value of the inputs, but also on the history of the inputs. For this reason, it is paramount to collect empirical data for applications of interest to be able to accurately predict the future power consumption of a CGRA.

4.1. Power-aware Architectural Simulation

To provide an empirical analysis, we need an infrastructure that allows us to study the activity levels of a CGRA executing a real application. To accomplish this we have used the Verilog Programming Language Interface (PLI) to annotate a CGRA's datapath with C routines that track either activity or power directly. One main advantage of using the PLI is that it provides visibility into the simulation and permits the monitoring functions to examine the data values of a processing element's inputs.

4.2. Creating Power Models

There are three approaches that we will use for creating a power model for a given CGRA. The first is to use published results for specific components, such as arithmetic and logic units and block memories, which are well studied for power efficient designs. A second approach, which is well suited to modeling the

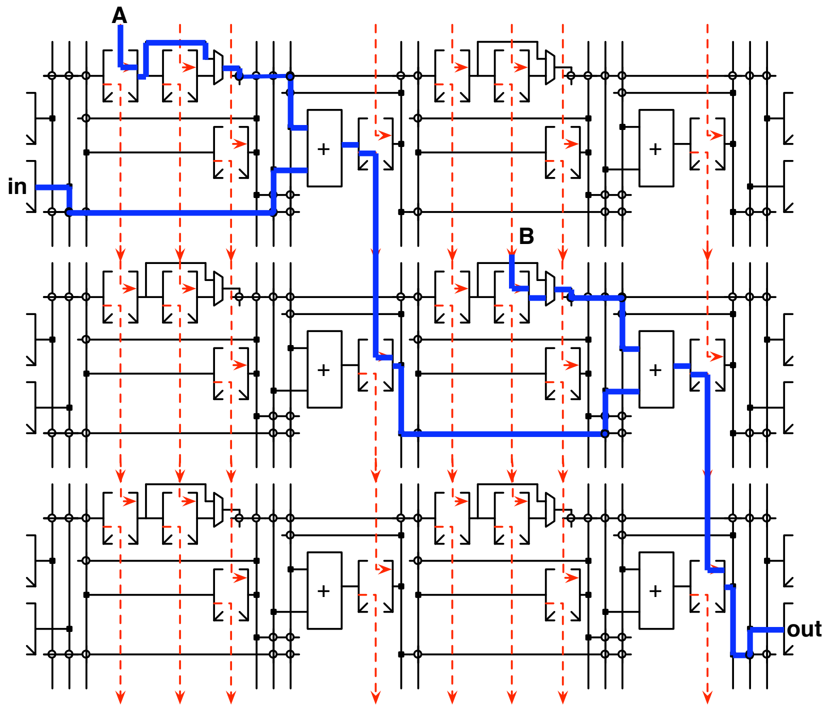


Figure 5. Visualizing a datapath graph.

interconnect, is to characterize the behavior of certain small structures in the CGRA, such as wires of different length, buffers, or switchboxes, in a specific VLSI process. The third approach, which is used for the processing elements, would be to layout components using a mixture of published results, standard cell, and custom layout techniques. For complex structures like the processing elements, this method provides the most accurate results, but is also the most labor intensive.

5. Mapping Applications to a CGRA

New architectures cannot be properly used or evaluated in a vacuum. While theoretical limits are a starting point for judging a new architecture's performance, real trade-off decisions must be made using a more realistic, workload driven evaluation. In addition, for an architecture to be useful, it must be easily programmed. A full tool chain to go from programs to execution is needed to properly evaluate the new architectures we are exploring.

To date, no tool chain currently exists to target a variety of coarse grained architectures. We intend to explore a variety of architectures and compare them on the basis of each architecture's merit, not the tool chain that targets them. Thus, it is important to have a single tool chain that can target all architectures in our study. Our goal is to build a tool chain to target a large class of CGRAs, enabling architecture exploration the same way VPR[8] allowed exploration of island-style FPGAs.

To this end, we are working on a flexible tool chain that will target a broad spectrum of configurable architectures. We ensure the flexibility we need by including a special tool to map our programs to the architecture of our choice. This is represented by the unified mapper block in Figure 3. It is designed to take the architecture-independent compiler output and generate a physical configuration for the chosen architecture.

We have chosen to use a statically scheduled dataflow graph to be our architecture independent representation. A dataflow graph represents a program as a directed graph where the nodes are operations to be performed, and the edges represent data dependencies. A simple example is shown in Figure 4. In this example, we first add inputs *in* and *a*, and the result is added to *b* to generate the final output *out*. The data dependencies are where the result of one operation is needed as the input to the next, for example the result of the *in* and *a* addition is needed by the addition with *b*. This representation is used in many compilers as an intermediate representation before creating the final program. Using a statically scheduled approach, we avoid the expensive runtime overhead necessary in dynamic dataflow execution approaches.

There are three main steps we take to map a dataflow graph to a specific architecture: scheduling, placement, and routing. The first step is scheduling. To avoid both structural and data hazards, the scheduler examines the data dependencies and the resources available in the architecture, and decides on a time for each operation to execute. We must ensure that no operation is scheduled to run before its inputs are available, and that we do not try to execute too many operations at the same time and oversubscribe our architectural resources.

The second step is placement. In this step, we choose physical execution resources to perform each operation in the dataflow graph. It is important to map operations that are dependent on each other close together while also keeping in mind the power implications of our architectures. For example, if the dataflow operation is a comparison of two numbers for equality, we would want to map it on to a power efficient comparator, even though a more power hungry ALU could also do the task. To meet these varied goals, we will use a Simulated Annealing based placement algorithm.

The final step is routing. In this step, we configure the physical interconnect to route the data from producers to consumers. This can be seen as mapping the dataflow arcs to the specific architecture. Our tool is based on the well known Pathfinder algorithm [9] for routing. Once the routing is completed, we have effectively mapped our architecture independent program into the physical architecture of our choice.

5.1. Implementing SPR

The traditional approach to scheduling, placement, routing, for devices like FPGAs, is to use independent tools that have no direct feedback, and rely on the programmer to manage their interaction. In our tool chain, these three steps are completed by a unified Schedule, Place and Route tool (SPR). While three step mapping technique is similar to the flow used for programming conventional FPGAs, one major difficulty using separate steps is that later steps must live with the decisions of the earlier steps. We are using a unified tool to explore the benefits of allowing later steps to influence earlier steps. Also, while FPGAs are statically programmed, with no notion of different operations during different cycles, one fundamental advantage of CGRAs is the coupling of function units with small instruction memories and program counters, allowing automatic time-multiplexing of the hardware. Our tool chain is constructed from the ground up to exploit this time-multiplexing for graceful performance degradation in mapping large applications to limited hardware.

The first example of how the tool integration is beneficial is in the interaction of the scheduler and the placer. As a dataflow graph is scheduled, there may be operations that have some slack around them - their inputs are available early, but their outputs are not needed until much later. This leads to a window of time where an operation can be scheduled. If we retain this window in the schedule, the placer gains some flexibility. The placer can now place the operation in both space and time. It can choose any time slice in the window to place the operation, on any available resource. This is helpful if all of the resources to execute an operation are used during one time slice in the window, but free during another.

If a placement is not possible using the current schedule and architectural resources, the placer will be able to ask the scheduler to generate a schedule with more slack in it. This corresponds to increasing the amount of hardware time-multiplexing used. As you scale back the available hardware and increase the time-multiplexing, this will lead to a graceful degradation of performance through increased latency, instead of simply a failure at the placement stage.

To allow the placer to move things in time, we treat registers as latency in the interconnect. This is a powerful idea, and means that we don't have to place registers explicitly. Instead, we use the QuickRoute[10] algorithm for latency aware routing, and registers are placed implicitly during routing.

A second example of how an integrated tool is useful lies in the ability of the router to provide feedback to the placer. Our router is based on an algorithm which tracks congestion as it produces routes. If it fails to route all connections between operations, it can provide the congestion information to the placer. The placer will then integrate the congestion into its placement cost metric, which will help it produce a better placement.

6. Testbenches and Applications

We are in the process of developing a set of applications and common kernels to drive the simulations that we will use to compare architectures.

Kernels

- FIR
- Matched filter
- 2D convolution
- FFT

Applications

- Georegistration (*e.g.* Sonoma Persistent Surveillance System via GPU [11])
- Select kernels from SWarp [12]
- K-means clustering
- Principal component analysis (PCA)

- Block-matching motion estimation
- Approximate string matching (Smith-Waterman and BLAST)

The kernels that we use, like FIR filters, FFT, and 2D convolutions, are relatively simple and known to match the strengths and weaknesses of reconfigurable architectures well, but are ubiquitous in the application domain of interest. Some of the applications, like K-means clustering, georegistration, and principle components analysis (PCA) are somewhat more complex and are specifically of interest to our collaborators at Lawrence Livermore and Los Alamos National Laboratories. Finally, applications such as block-matching motion estimation and approximate string matching are used to add breadth to our chosen application domain and generalize the conclusions of our architectural exploration.

An important issue for all reconfigurable architectures is how well they can handle potentially problematic algorithm features, like lack of regularity and unpredictability. For many simple algorithms that work well on reconfigurable architectures, there are more efficient heuristic versions that trade-off increased complexity, and in some cases less accurate results, for substantially reduced overall computational work. Reconfigurable architectures and their programming tools need to have some flexibility in this regard, because if the decrease in work is large enough, then the more complex algorithm running on a conventional processor will outperform the simpler algorithm running on any reasonable reconfigurable architecture. In addition to adding breadth to our application domain, block-matching motion estimation and BLAST provide good examples of where a heuristic can dramatically reduce the amount of computation necessary to achieve a result that is nearly as good as the brute force methods. Using these applications, we hope to ensure that the key architectural features identified by our study, will continue to be beneficial when users come up with more optimized and complex versions of their algorithms.

6.1. Macah

As described in the previous section, our architecture exploration method is intimately bound together with the SPR tool. SPR's input language is dataflow graphs. Therefore, we need some language or tool for generating these graphs. Furthermore, SPR is capable of exploring space-time trade-offs for a particular dataflow graph, but is not designed to explore more global trade-offs such as the impact of varying buffer memory sizes.

We are developing a language called Macah and a compiler for it that gives us a relatively simple method for generating a range of dataflow graphs for a particular application in a programmer-directed way. Macah is a C-like language in the tradition of Streams-C [13], NAPA-C [14] and Impulse-C [15]. These languages provide additional features that give the programmer greater control over the final implementation than plain C, but require far less effort than conventional hardware description languages.

Three of the features that help give Macah its balance of control over program implementation and ease of use are kernel blocks, streams and shiftable arrays. Kernel blocks are used to indicate which pieces of a program should be compiled into configurations for a reconfigurable coprocessor, *i.e.* a CGRA. This abstraction allows us to experiment with different coprocessor interfaces without modifying any interface code in our applications; the changes are localized in the compiler and runtime libraries. Syntactically, kernel blocks are simply C blocks marked with the new keyword "kernel" (`kernel { ... }`).

The term "stream" is used somewhat differently in different settings; in Macah, streams are first-in, first-out data channels between threads. Streams support four basic operations: creation, deletion, sending and receiving (including both blocking and non-blocking variants), and they are used to specify the I/O behavior of a kernel. In typical usage, a kernel will have one or more supporting data access functions that simply traverse data structures and either send data to a kernel or receive data from a kernel. In order to enable pipelining, we need to know the dependence relationships within and between iterations of loops in kernels. By "replacing" array and pointer expressions inside of kernels with stream sends and receives for communication with external memory, we also eliminate the need for sophisticated pointer and array subscript analysis to determine this dependence information. In a sense, Macah makes it the programmer's responsibility to ensure that there are not any "bad" inter-iteration dependencies. The stream send and receive operators are written `stream_exp <! exp;` and `lval <? stream_exp;`, respectively.

Shiftable arrays are used to efficiently describe data movement and access patterns that are common to DSP. They are just like standard C arrays, except they additionally support the shift left and shift right operators (`arr <<= n;`, `arr >>= n;`). The result of executing a shift left (right) by n operation on a shiftable array is that each of the elements of the array moves "to the left (right)" by n indices; the n right-most (left-most) elements are filled with arbitrary bits. Shiftable arrays are declared like C arrays, except using `arr[^size^]` instead of `arr[size]`. Shiftable arrays make it easier to write sliding window-type programs (like many image processing algorithms) more concisely, and somewhat simplify the compilation of such programs.

The following code illustrates the use of these features in a simple FIR filter. A more realistic implementation would include additional buffer array declarations and slightly more complicated looping code to guide the compiler in deciding how to parallelize the code for CGRAs.

```
void fir(int *I, int *O, int *C,
         int n, int nc)
{
  int Ibuf[^nc^];
  void fetchIns(int stream s) {
    for (int i = 0; i < n; i++)
      s <! I[i];
  }
  void storeOuts(int stream s) {
    for (int i = 0; i < n - nc; i++)
      O[i] <? s;
  }
  int stream is = launch(fetchIns);
  int stream os = launch(storeOuts);
  kernel {
    for (int i = 0; i < n; i++) {
      Ibuf >>= 1;
      Ibuf[0] <? is;
      int oTemp = 0;
      for (int j = 0; j < nc; j++) {
        oTemp += Ibuf[j] * C[j];
      }
      if (i >= nc)
        os <! oTemp;
    }
  }
}
```

`Ibuf` is a shiftable array that is used to store a sliding window of the input data. The (nested) functions `fetchIns` and `storeOuts` define how the input and output arrays should be accessed. The `launch` function takes a function pointer and does the appropriate stream allocation and initialization. Inside the kernel itself, each iteration of the outer loop shifts the buffered inputs over by 1, receives a new input from the input stream, calculates a single output value and sends that output out on the output stream. The conditional guard on the send operation in necessary to wait for the number of inputs received to be equal to the number of coefficients.

Each kernel in a Macah program is compiled into a dataflow graph that is then passed on to the SPR tool. The dataflow graphs contain all of the control and data dependencies needed to execute the kernel from which they were generated. As a result, we can simulate an application pre-SPR by generating a modified version the non-kernel part of the application that launches a Verilog simulator to run the kernel dataflow graph. While this pre-SPR simulation is unable to provide accurate performance estimates, the ability to functionally verify the output of the first major stage of the compiler is invaluable for rapidly debugging an application and the tool chain.

7. Related Work

FPGAs have become a viable alternative for ASICs in a variety of domains. This widespread adoption comes despite an order of magnitude reduction in speed, area, and power versus ASICs. Modern mainstream FPGA architectures are the product of decades of research aimed at reducing this penalty of configurability.

Modern FPGA architectures are comprised of look-up tables (LUTs), registers, and small memories grouped together in logic blocks. Using lookup tables as the basic logic unit provides bit-level signal manipulation and fine-grained control. Bit-level signals are routed between logic blocks by configurable routing networks. Conversely, CGRAs (coarse-grained reconfigurable arrays) contain higher-level logic blocks connected together by a configurable network. Instead of operating on bit-sized signals, these architecture route and process word-sized signals. CGRAs have been built to minimize power consumption and configuration overhead inherent in fine-grained FPGA architectures.

Because of the high overhead of having bit-level configuration, FPGA architects are integrating a number of higher-level logic blocks into their devices such as multipliers, DSP blocks, and even processors to make common functions more efficient [16]. This trend of adopting coarse-grained components into fine-grained architectures is blurring the line between the two and shows that the power and performance benefits of CGRAs are very attractive. This shows that as designers have more silicon area to use, the best choice has been to add larger, coarse-grained devices instead of strictly increasing the number of LUTs in the architecture.

Early CGRA architectures such as RaPiD [17] and Matrix [18] experimented with the appropriate mix of resources and routing structures. Matrix is a 2-dimensional matrix of ALUs with local memories. Its routing network is static, highly connected to nearest neighbors and has some bus-level resources. RaPiD is a 1-dimensional, heterogeneous mix of specialized ALUs and multipliers with a time-multiplexed, static channel routing structure. It showed that the best configuration for CGRAs is highly application-specific, thus motivating our exploration of the architecture design space and the necessity of a flexible tool chain.

Two groups are working on many-core processor architectures, AsAP and Ambric. AsAP (Asynchronous Array of Simple Processors) [19] is a many-core processor architecture that contains many low-power processors with local memories and nearest neighbor communication. By using small memories, simple architectures, and local communication, AsAP reduces power consumption and increases energy efficiency. Ambric, Inc. [20] has released a similar architecture designed to have a simple programming model and low power consumption. The Ambric architecture includes a dynamic routing network that has packet switching capabilities. Both devices can scale to include hundreds of processors on a single die while consuming little power. Unlike CGRAs these many-core architectures are designed for general purpose processing rather than domain specific datapath computing.

Hybrid sequential and spatial systems is another area of active research. Our previous work on hybrid architectures [21] shows the benefits of tightly coupling a sequential processor that can execute control dominated sequential code with a reconfigurable fabric (CGRA) that can execute regular, compute intensive sections of code quickly. MorphoSys [22], Stretch [23], and ADRES [24] each include a reasonably complex sequential processor coupled to a reconfigurable architecture that is used to extend the instruction set in some manner. These hybrid approaches are complementary to our work with developing a CGRA, providing the future possibility of integrating our CGRA into a larger hybrid system.

8. Summary

The inability for general purpose processors to double their performance every eighteen months, as previously enabled by Moore's law, has created new opportunities for spatial computing, especially for high performance application domains. Coarse-grained reconfigurable architectures are poised between ASICs and FPGAs to emerge as a new computing platform, providing near ASIC performance with FPGA-like flexibility. But, they have been plagued by challenging programming models and have been unable to differentiate themselves based solely by their power to performance ratio. Our research focuses on solving both of these problems, by designing an architecture specifically for power efficiency and with a new unified mapper that dramatically improves the task of mapping an application to an architecture.

To achieve these goals we have started by conducting a study of the power efficiency of different interconnect topologies, generalized the applicability and effectiveness of the unified mapper, and created a new C-like language that simplifies the task of describing application kernels that are targeted for CGRAs. We hope to show that high level architectural decisions can provide significant power savings. Additionally, we expect that using a unified mapper that automatically maps an application to an architecture in both space and time will significantly reduce the burden on the application programmer. Finally, we hope our work with the Macah language will make CGRAs more accessible to embedded programmers and scientists as well as hardware developers.

References

[1] A. DeHon, "The Density Advantage of Configurable Computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[2] D. Cronquist, *et al.*, "Architecture design of reconfigurable pipelined datapaths," in *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, Atlanta, 1999, pp. 23–40.

[3] S. C. Goldstein, *et al.*, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[4] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2006, pp. 21–30.

[5] K. K. W. Poon, S. J. E. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 2, pp. 279–302, 2005.

[6] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000, pp. 83–94.

[8] 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.

[9] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *ACM International Symposium on Field-Programmable Gate Arrays*. ACM Press, 1995, pp. 111–117, monterey, California, United States.

[10] S. Li and C. Ebeling, "QuickRoute: a fast routing algorithm for pipelined architectures," in *IEEE International Conference on Field-Programmable Technology*, Queensland, Australia, 2004, pp. 73–80.

[11] "Sonoma Persistent Surveillance System." [Online]. Available: http://www.llnl.gov/str/November05/Johnson.html

[12] "SWarp." [Online]. Available: http://terapix.iap.fr

[13] M. Gokhale, *et al.*, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 49–56.

[14] M. Gokhale and J. Stone, "NAPA C: Compiling for Hybrid RISC/FPGA Architecture," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[15] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[16] *Virtex-5 Family Overview - LX , LXT, and SXT Platforms*, Xilinx, Inc. [Online]. Available: http://direct.xilinx.com/bvdocs/publications/ds100.pdf

[17] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - Reconfigurable Pipelined Datapath," in *International Workshop on Field-Programmable Logic and Applications*, R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, 1996, pp. 126–135.

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

[19] Z. Yu, *et al.*, "An Asynchronous Array of Simple Processors for DSP Applications," in *IEEE International Solid-State Circuits Conference, (ISSCC '06)*, Feb. 2006.

[20] "Ambric Technology Overview." [Online]. Available: http://www.ambric.com/technology/technology-overview.php

[21] B. Ylvisaker, B. Van Essen, and C. Ebeling, "A Type Architecture for Hybrid Micro-Parallel Computers," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2006.

[22] H. Singh, *et al.*, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[23] A. Wang, "The Stretch Architecture: Raising the Level of Productivity and Compute Efficiency," Keynote Speech, 6th WorkShop on Media and Streaming Processors, December 2004.

[24] B. Mei, *et al.*, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *13th International Conference on Field Programmable Logic and Applications*, vol. 2778, Lisbon, Portugal, 2003, pp. 61–70, 2003.