Compiling for Coarse-Grained Adaptable Architectures[1]

Carl Ebeling
Department of Computer Science and Engineering
University of Washington

Technical Report UW-CSE-02-06-01

Abstract

This document describes a proposed approach to compiling high-level language programs to the Rapid configurable architecture. Since the general version of Rapid very broad, this approach should be applicable to a wide range of coarse-grained configurable architectures. The compiler is comprised of a standard compiler front-end and a backend scheduler based on a simultaneous place and route formulation of the problem. This will allow the many constraints of a configurable architecture to be solved simultaneously.

Introduction

Coarse-grained adaptable architectures have the potential to radically change the way embedded systems platforms are built. This potential has not yet been realized because coarse-grained adaptable architectures have not been widely studied or used. Until there are effective ways to program components based on these architectures, and include them in a system architecture, this situation is not likely to change. This document describes our approach to compiling programs in a high-level language to a coarse-grained configurable architecture like Rapid.

Compiling high-level language programs to FPGA-based architectures is very similar to the high-level synthesis problem. A description of the computation is compiled into a specialized hardware implementation while optimizing some objective function based on a combination of cost and performance. Since FPGAs allow the construction of arbitrary hardware structures, there are no constraints on the structure that is synthesized, other than the cost and performance constraints. There is a substantial body of research devoted to this topic, but unfortunately it does not translate well to coarse-grained configurable architectures.

Compiling to a coarse-grained configurable architecture like Rapid presents a more constrained problem because the configurable substrate is not a blank slate where all things are possible. A fixed set of function units is available: the compiler must make do with the number and combination when searching for an efficient solution. Even more important, the data interconnection network used to communicate between function units is necessarily limited in the interests of cost.

We view coarse-grained adaptable architectures like Rapid as scaled up version of VLIWs, with architectural features that allow them to be efficient. These features include distributed registers throughout the datapath, a limited data interconnect, and a configurable control architecture that takes advantage of the features of large-scale parallel computations.

The problem of compiling to an adaptable architecture is thus similar to the problem of compiling to a traditional VLIW architecture: Operations specified in the program must be scheduled to a fixed number of function units, and the data operands and results must be transferred between function units via registers. However, compiling to an adaptable architecture is much more difficult:

1. Adaptable architectures have many more function units, typically an order of magnitude more, than a VLIW. The compiler must discover and expose sufficient parallelism to make efficient use of these function units, and the scheduler must schedule them in space and time.
2. Data communication is much more constrained in an adaptable architecture because a general interconnection network like a crossbar or multi-ported register file is far too expensive for so many function units. Thus both function units and wires have to be scheduled by the compiler.

[1] This research was supported by the National Science Foundation Experimental Systems Program (EIA-9901377).

3. Sufficient data bandwidth must be provided to keep all function units operating in parallel. Instead of centralized, multi-ported register files, registers are distributed throughout the datapath to achieve high bandwidth, locality of reference and low power. In addition, small memories are distributed throughout the datapath, which serve as local data caches that supply data close to where it is used. The compiler must be able to use these registers and memories effectively.
4. Control in a coarse-grained reconfigurable architecture is typically very constrained and the compiler must generate solutions that can be implemented using the given control architecture.

We propose to develop a compiler for highly parallel coarse-grained adaptable architectures, structured as shown in Figure 1. This compiler will comprise a traditional VLIW compiler front-end that transforms a program written in a high-level language like C or Java into a control/dataflow graph, a scheduler that maps the dataflow graphs to the adaptable substrate in both time and space, and an optimizer that generates object code in the form required by the architecture. Our research will focus on the very difficult problem of scheduling control/dataflow graphs to a substrate with many interconnection and control constraints. This scheduling problem will be cast as a place-and-route problem whereby dataflow graphs are placed-and-routed onto a space-time computation substrate comprising multiple instances of the datapath graph unrolled in time. Casting the scheduling problem this way allows many difficult and interacting subproblems that arise in the context of an adaptable architecture to be solved simultaneously. These include mapping dataflow operations to datapath function units, assigning busses for data transfer, solving control constraints defined by the architecture, generating efficient time-multiplexed solutions for large problems, and performing timing optimizations like pipelining and retiming. By using an architecture-independent formulation of the problem, the algorithms we develop should be applicable to a range of coarse-grained architectures, including more traditional clustered VLIWs, which are smaller and less constrained.

Initially, this compiler will target a generalized version of the Rapid architecture, which is described in some detail in a companion document. The work we have done to date has shown that Rapid can deliver high performance with relatively low cost and power for a wide range of kernel computations encountered in embedded systems. We believe that Rapid captures the essential elements of coarse-grained adaptable architectures and thus serves as a good initial target. For example, the control architecture in Rapid can be varied from simple, expensive control, to highly optimized, efficient control. This variation will permit us to experiment with different types of control and the compiler algorithms required to take advantage of them.

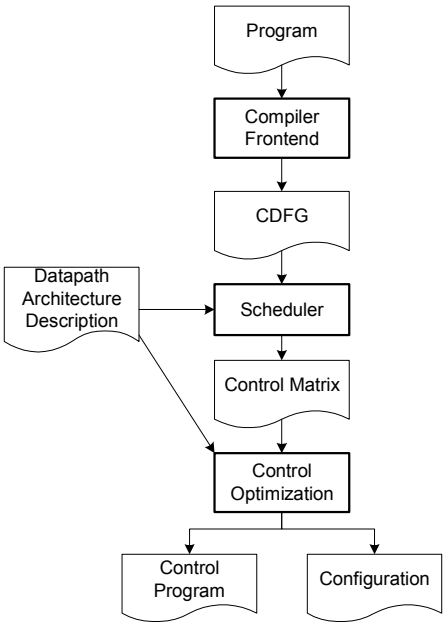

Figure 1 - Proposed tool flow

This document assumes a familiarity with the Rapid architecture as described in "The Generalized Rapid Architecture". We begin by describing how programs are written for Rapid and follow this by a description of our proposed approach for mapping dataflow graphs defined by such a program onto the adaptable architecture.

Programming Rapid

The Rapid benchmark architecture is currently programmed using a simple assembly-level language called Rapid-C. The programmer is responsible for scheduling the computation on a cycle-by-cycle basis by describing when and where each operation is performed in the datapath. Although the pipelined programming model reduces the complexity of this specification, it restricts the kinds of programs that can run on Rapid to those that can be easily pipelined. Although this covers most of the kernels of interest, it does restrict the range of algorithms used and makes some, like Viterbi, very difficult or impossible to write. Moreover, the burden of scheduling falls completely on the programmer, who becomes more of a hardware designer than a programmer. Moreover, the backend cannot fully optimize the control generation because the programmer has already done the scheduling.

Our goal is to move the scheduling and optimization into the compiler, allowing programming to be done at a higher more abstract level, and enabling better optimization of the object code. Programs will be written in a high-level language, possibly extended with constructs that allow the programmer better control of the generated control/dataflow graphs. A conventional VLIW-style compiler front-end will be used to produce a control/dataflow graph for the program, which is passed to the scheduler and optimizer. Although the focus of the proposed research is on this backend, we will investigate ways to perform program transformations in the compiler front-end to make the control/dataflow graph more amenable to scheduling.

A program execution comprises the execution of a sequence of dataflow graphs, as determined by the control flow. We rely on the compiler to produce large dataflow graphs that can be mapped efficiently to a large datapath to achieve a high degree of parallelism. The scheduler must perform two related tasks: First, it schedules individual dataflow graphs onto the datapath. Second, it must "stitch" these dataflow graph executions together to implement the communication between them.

The compiler cannot be expected to take arbitrary C programs and produce highly parallel dataflow graphs that can be mapped to the reconfigurable datapath. The programmer must understand the capability and limitations of the hardware. In particular, the programmer must understand the computation model, which relies on keeping the data close to the operators, and not global memory, which cannot be accessed efficiently. The intent is to allow the programmer to describe the computation as a set of dataflow graphs described implicitly as a C program. The compiler first transforms the program into a control/dataflow graph that is then scheduled onto the parallel hardware. The following program is an example of a kernel computation that can be executed very efficiently on a highly parallel reconfigurable datapath.

```
#define M 16
#define N 8
matmult ( StreamIn AStream, StreamIn BStream, StreamOut outStream) {
  int r, c;
  int i;
  // Datapath registers and memories
  Register int A;
  Register int Sum[N];
  Register int B[M][N];          // N datapath memories of length M

  // Initialize the internal B array
  for (r=0; r<M; r++) {
    for (c=0; c<N; c++) {
      B[r][c] = BStream++;
    }
  }
  for (i=0; i<N; i++) {
    for (c=0; c<N; c++) {
      Sum[c] = 0;
    }
    for (r=0; r<M; r++) {
      A = AStream++;
      for (c=0; c<N; c++) {
        Sum[c] += A * B[r][c];
      }
    }
    // Write back results
    for (c=0; c<N; c++) {
      outStream++ = Sum[c];
    }
  }
}
```

The AStream, BStream and outStream variables refer to streaming input and output ports connected to memory, whose address generators are defined elsewhere. The A and Sum variables refer to values local in the datapath and B refers to a set of N memories in the datapath used to store the B matrix. The first loop nest initializes the B memories, one value at a time, from BStream. The **i** loop nest does the matrix multiply, producing one row of the result matrix at a time. The body of the **r** loop is a single dataflow graph when the inner loop is unrolled and can be executed in a single cycle if there are sufficient resources in the datapath.

This program illustrates some of the difficult challenges for the compiler. Data dependence analysis can determine that the c loop at the end that writes one row of results to the output stream can be overlapped with the execution of the beginning of the next iteration of the i loop if the Sum variables are renamed. This doubles the performance, assuming that the dataflow graphs of the overlapped computations can be scheduled together, which turns out to be straightforward. Typically, many matrix multiplies are executed in a loop, with different A and B matrices. An analysis of the computation can determine that the initialization of the B matrix for the next matrix multiply can be overlapped with the computation of the current matrix multiply if the B matrix is renamed. This doubles the performance yet again without increasing the memory bandwidth. We will explore the data dependence analysis and renaming that allows component dataflow graphs in the CDFG to be scheduled in together. This technique is analogous to software pipelining [21], except that entire loops are scheduled instead of instructions.

This program illustrates the use of language constructs that allow the user to reference datapath resources like I/O streams and datapath registers and memories, and shows the programming style that exposes the parallelism of the computation as implicit dataflow graphs embedded in a control flow graph. In this case the programmer has written the program to expose the parallelism, but we will also explore conventional loop transformation techniques that can extract parallelism from nested loops. A conventional VLIW compiler front-end will be used to transform programs into a control/dataflow graph that is passed to the scheduler and optimizer. This front-end will perform the control flow and data dependence analysis that will allow independent dataflow graphs to be scheduled together. In addition, common compiler optimizations like if-conversion and loop transformation will be used to push some of

the control flow into the dataflow graphs and to generate dataflow graphs that match the underlying computation substrate.

Program Dataflow Graphs

The dataflow graphs produced by the compiler front end must be scheduled onto the reconfigurable datapath. As described in the architecture overview, this scheduling must satisfy the data dependence constraints in the dataflow graph, the function unit and data interconnect constraints of the datapath, and the constraints of the control architecture. The scheduling problem is formulated as a traditional place-and-route problem, where the dataflow graph is place and routed onto an execution graph that represents the execution resources of the datapath in both space and time. Solving the resulting place-and-route problem solves the problem of assigning operators to function units to maximize parallelism, and the problem of transferring data values in the tightly constrained data interconnect network. In addition, the place-and-route solution automatically performs register assignment, pipelines the dataflow graph, and determines how to use memories to time-multiplex hardware resources over a large dataflow graph.

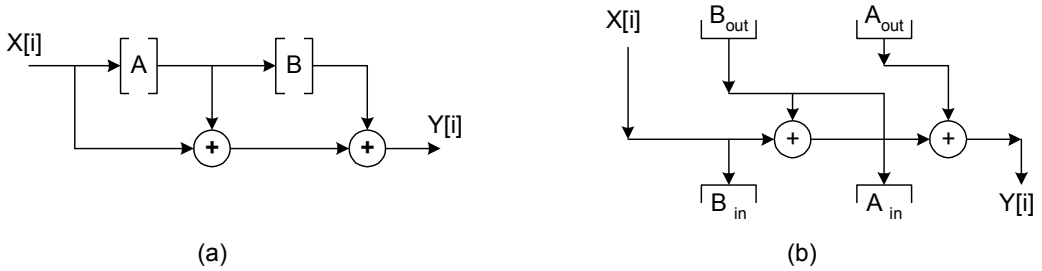

Figure 2 – (a) A simple dataflow circuit for adding all subsequences of 3 numbers in the input stream. (b) The dataflow graph is a single execution of this circuit.

Figure 2 (a) shows a dataflow circuit representation of a simple computation that computes the sum of all input subsequences of length 3. The dataflow graph in Figure 2 (b) represents a single execution of this dataflow circuit. A dataflow graph is a DAG comprised of nodes that perform combinational operations and edges that describe the flow of data between the nodes. There are no registers in a dataflow graph. The dataflow graph inputs (live ins) flow are operated on by operators in the dataflow graph to produce a set of outputs (live outs). These inputs are outputs are stored in registers that provide storage for values that are produced by one dataflow graph and consumed by another. These registers may be temporary registers in the datapath used to forward results from one dataflow graph to the next, locations in local memory that provide longer term storage, or registers associated with input and output ports.

The Rapid Datapath Graph

The Rapid datapath is represented by a circuit graph comprised of function units, memories, registers, multiplexers and the wires that connect them. This graph represents only the datapath of the Rapid array; the control is represented separately and is determined as a side effect of the process of scheduling the computation to the array. This control is optimized and generated by later steps in the mapping process. Figure 3 gives the graph for a simple datapath that has just two function units and some registers, with two input ports at the left and two output ports at the right. The interconnect multiplexers are shown as circles with the line in the circle aligned with the shared multiplexer output: Only one circle on this line can be connected at a time. This datapath, with an assignment of values to the control signals, implements a specific computation that is performed during a single clock cycle. Dynamically controlled connections can change from one clock cycle to the next, while statically configured connections remain the same for the entire computation. For these examples, we will assume all multiplexers are dynamic.

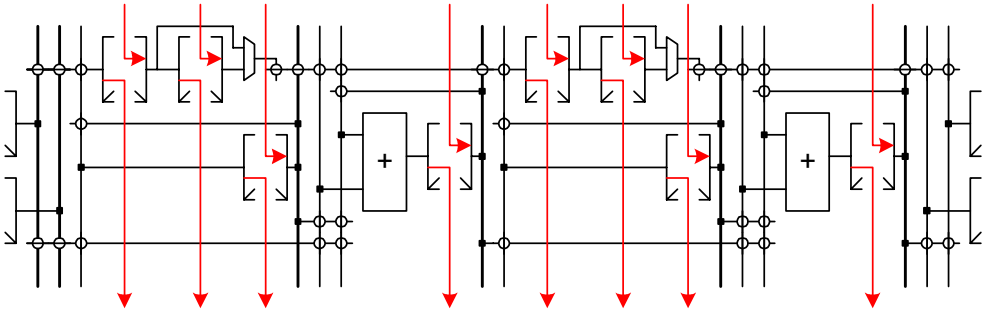

Figure 3 - Example of a simple datapath graph

Registers in the datapath graph are split to indicate that the datapath graph is a combinational circuit during one clock cycle. At the beginning of a clock cycle, the register outputs are valid and flow through the dataflow graph implemented by the connections. The outputs are then latched into the registers at the end of the clock cycle. The red arrows show that the values written to the registers are forwarded to a new instance of the datapath graph in the next clock cycle, which is a copy of the datapath graph with a (typically) different set of connections.

Scheduling Dataflow Graphs

A dataflow graph is executed by the datapath over several cycles by setting the connections such that the dataflow graph is embedded in the datapath graph, and is thus directly executed by the datapath. Such an embedding for the dataflow graph of Figure 2 (b) in the datapath graph of Figure 3 is shown in Figure 4 on the next page. The datapath has been copied three times since it takes 3 clock cycles to execute the dataflow graph, with the first cycle at the top. Note that the register inputs from one clock cycle feed into the registers outputs of the next clock cycle. The execution of the datapath thus forms a large computational substrate in space and time. Scheduling is performed by mapping dataflow graphs to this space/time substrate.

The bold blue lines show how the dataflow graph has been mapped to the unrolled datapath graph. The value of X[i] and A_{out} are added in the first clock cycle, but the result value is forwarded via the pipeline register to the next clock cycle, which adds the value of B_{out} to it, and forwards the result to the next clock cycle via a second pipeline register and thence to the output. The labels show where the A_{out} and B_{out} inputs have been forwarded through registers from some previous dataflow graph, and how the outputs A_{in} and B_{in} have been forwarded to some next dataflow graph The assumption is that the previously executed dataflow graph has deposited the appropriate values into these registers, and the next dataflow graph will take the values from these registers.

Note that the graph shown in blue is entirely combinational. The pipeline registers that are inserted on the result path are not part of the dataflow graph, but are inserted only to allow the dataflow graph to span more than one clock cycle. In this case, the pipelining is enforced by the pipeline register on the function unit outputs. In the more general case, the function unit output may have the option of passing to the input of another function unit without a pipeline register. In this case, the scheduler must decide whether or not to insert a pipeline register.

Since the circuit graph is configurable, it may contain paths whose delay exceeds the clock cycle. That is, since components may be composed differently within a single clock cycle, it may be possible to configure a path that has too much delay. Although this is not possible in this example because each function unit has an associated pipeline register, in general the place and route scheduler must ensure that all paths meet the clock cycle constraint.

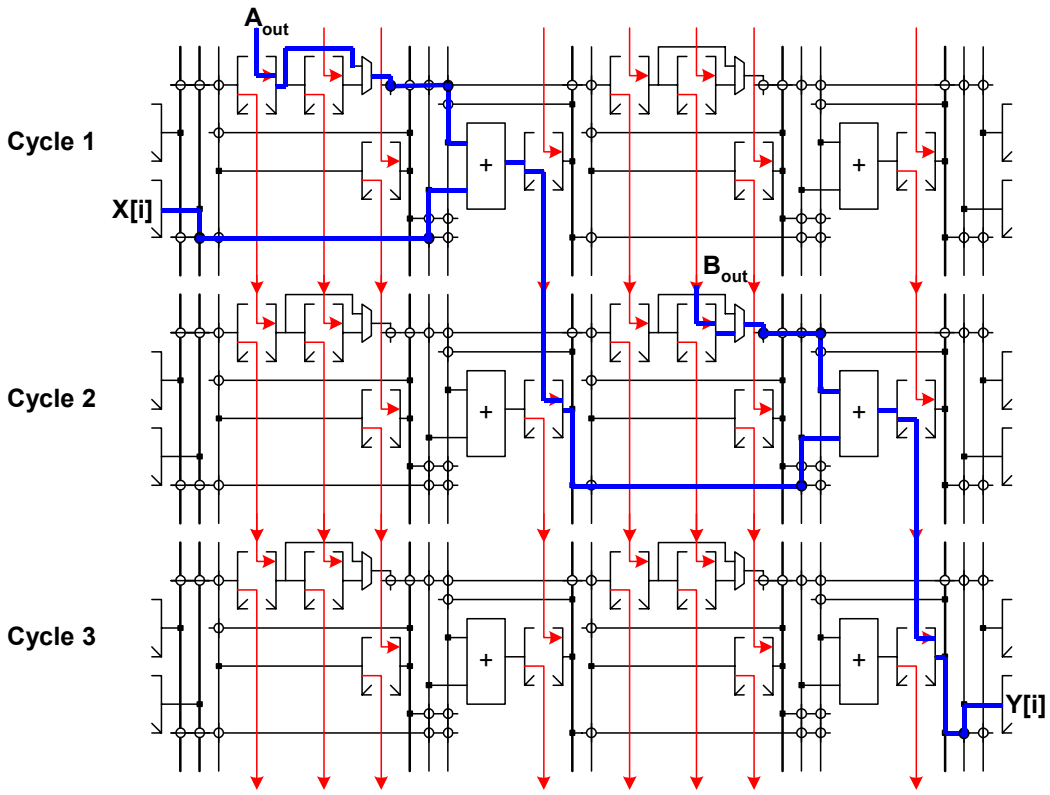

Figure 4 - The dataflow graph of Figure 2 mapped to the datapath over three cycles

Dataflow graphs are not executed in isolation, but get their inputs from a previous dataflow graph and send their results to a next dataflow graph. When a dataflow graph is executed in a loop, it forwards data to the next invocation of itself, as shown in Figure 2. A second iteration of this dataflow graph is shown in Figure 5 with bold purple lines. The communication of the outputs of the first iteration to the inputs of the second iteration is shown using bold red lines.

An indefinite number of executions of the dataflow graph can be executed on successive clock cycles, with each dataflow graph overlapping its neighbors. This can be represented, as in iterative modulo scheduling, by folding the datapath graph back on itself. In this case, an execution of the dataflow graph starts on every cycle; that is, the iteration interval is one. Thus only one copy of the datapath graph is required, as shown in Figure 6, where values forwarded across the bottom wrap back to the top of the datapath graph. When the datapath graph is folded back on itself, different parts of the mapped dataflow graph correspond to different iterations. This is shown in Figure 6 using different colors as well as time superscripts. The blue part corresponds to the iteration starting on this cycle, the purple part corresponds to the iteration that started on the previous cycle, and the green part started 2 cycles before. The bold red lines show where values are forwarded from one dataflow graph to the next.

A dataflow graph is scheduled to the datapath by constructing a sufficiently large substrate by unrolling the datapath graph. If the dataflow graph is executed in a loop, then this substrate is wrapped by connecting the outputs at the bottom back to the inputs at the top. The simultaneous place and route algorithm is then used to map the given dataflow graph to this substrate. If the problem cannot be solved, then the size of the substrate is increased by adding another copy of the datapath graph, and a new attempt is made.

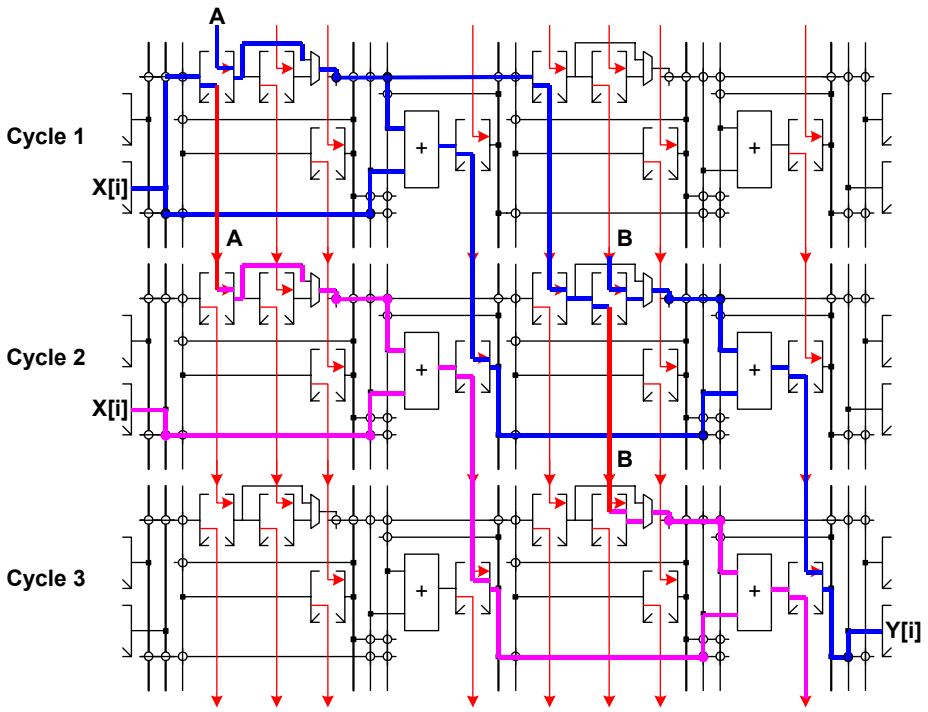

Figure 5 - The dataflow graph of Figure 2 mapped to the datapath over three cycles

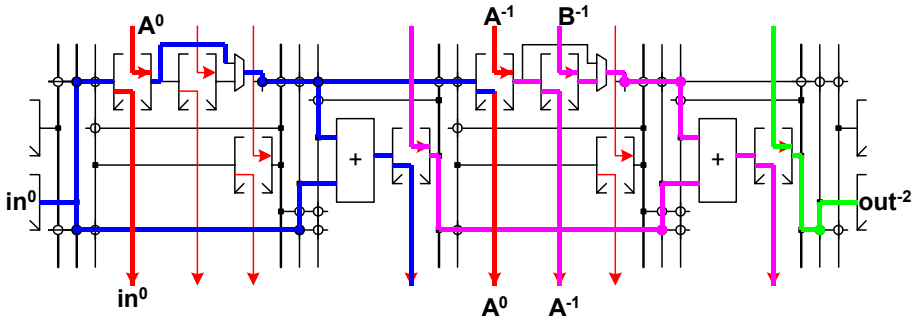

Figure 6 – One copy of the datapath graph with register edges wrapped to form three clock cycles

An Aside on Timing Notation

It is important to be precise about the meaning of names, especially with respect to the meaning of subscripts. A name can refer to a data value in the application, the value in a register, or the register or wire itself. For example, X could refer to a value in an array from the application, or to an input or output of the dataflow graph, or to a specific register or wire in the datapath. Complicating the situation is the fact that a dataflow graph may be executed many times and we need to distinguish each execution, and particularly the values in each execution from each other.

We will name inputs, outputs and internal signals of a dataflow with simple names, possibly subscripted where signals are related. If a dataflow graph is executed multiple times, for example, in a loop, we will use a superscript on the signal name to denote the iteration number of the DFG. Thus, A and X_i refer to signals in the dataflow graph. The value of these signals is determined by how the registers supplying those signals were written. We don't really care what these values are – what we care about is relationship between signals in the dataflow graph and between dataflow graphs. The names A^t and X^t_i refer to the signals A and X_i at time t, that is, during the t'th iteration of the dataflow graph. We use subscripted names for signals that are assigned to each other from one dataflow graph to the next. The value of X_i at time t becomes the value of X_{i-1} at time t+1. More succinctly, $X^t_i = X^{t+1}_{i-1}$ and thus it is easy to check whether two signals have the same value – the values are equal if their subscript + superscript add to

the same number. When a dataflow graph is mapped to the datapath graph, possibly duplicated and wrapped from bottom to top, the signal names are used to annotate the wires and register inputs and outputs of the datapath graph. In so doing, the superscripts denote which iteration each part of the mapped dataflow graph belongs to. If a signal is forwarded from the bottom of the datapath graph back to the top, the superscript is decremented, since this signal is now being used in the previous iteration of the dataflow graph, that is, the DFG that began at the previous time. Again, when a signal from one iteration is forwarded to another iteration, the identity $X^{t}_{i} = X^{t+1}_{i-1}$ is used to transform the name from one iteration space to another. (If subscripts are not used to denote time-related signals, the identities can be written from the relationships between values in the dataflow graph.) Dataflow graph operators can similarly be annotated with a time superscript to denote which iteration they belong to.

Time-Multiplexing

Many configurable architectures are constrained in the size of the dataflow graphs that can be implemented. If a circuit dataflow graph is too large to fit, then it must be redesigned so that it does. Some architectures have been designed that support the concept of "virtualized" hardware, where pages of hardware resources are instantiated by quickly swapping configurations. However, the ability to change a large number of configuration bits is very expensive in both area and power, and mapping to such architectures is difficult. The Rapid architecture support time-multiplexing via the memories included in the datapath. As a dataflow graph is executed over many cycles, the temporary values are stored in memories used as a fixed-length buffer. If the dataflow graph is uniform, then an efficient time-multiplexing solution can be found so that each cycle repeats the same (or almost the same) instruction using different data stored in the memories. That is, each execution of the datapath implements a section of the dataflow graph, and since the dataflow graph is regular, each section is executed using the same, or very similar, instructions.

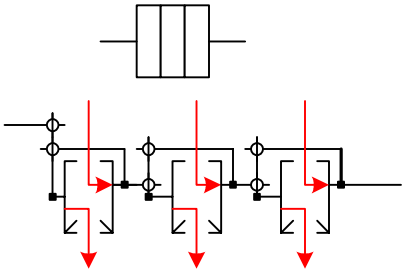

Figure 7 – Datapath representation of a memory module used as a fixed-length buffer

Using several cycles to execute a dataflow graph requires storage for the values which are not being used, but must be held for the next iteration. This storage can be provided by datapath registers, if only a few are needed. In general, especially for large dataflow graphs, this is insufficient, and storage in the form of a fixed-size buffer, which we call a CLSR (Configurable Length Shift Register), is a more efficient way to provide this. A CLSR of size N is just a shift register that holds N values and shifts each time it is read/written. A CLSR is implemented efficiently in the datapath using a memory along with a single address pointer used for both read and write (implemented as write after read). This pointer is incremented modulo the size of the buffer on each read/write. This representation is shown in Figure 7 for a buffer with three entries. The memory imposes an underlying constraint that all of the feedback muxes are set the same, to either hold or to shift, and this constraint must be included as part of the scheduling problem. In general, the size of the buffer may not be known and an additional multiplexer must be included to give the scheduler the ability to select the appropriate buffer size. It may be the case that a value needs to be shifted into the buffer before a value needs to be shifted out, or equivalently that a value needs to be shifted out before one is ready to be shifted in. This case can be handled easily by using a datapath register in addition to the memory implementing the buffer to absorb this extra value. It is also possible to use memories that have separate read and write addresses.

Figure 8 gives a dataflow graph that has an initiation interval of three clock cycles for a datapath with only two adders. We will add to our previous datapath a buffer of size 2 that will allow three-way time-multiplexing. Figure 9 shows three copies of this new datapath graph that forms the substrate for a loop with an initiation interval of three clock cycles. The bold blue and red lines show how the dataflow graph of Figure 8 is mapped to this datapath substrate. The dataflow graph is mapped over 7 cycles, even though a new execution is started every 3 cycles. The

left three additions of the dataflow graph are implemented vertically in the left part of the datapath in the first three cycles, shown in blue, and the rightmost three additions are implemented in the right half in the next three cycles, shown in purple, with the output written on the 7th cycle, shown in green. The forwarding of values from one dataflow graph to the next is shown in red. Note that the two halves of the datapath operate almost identically, which greatly reduces the control complexity by allowing control signals to be shared between the two halves. Note also that the constraints on buffer controls are satisfied by this schedule.

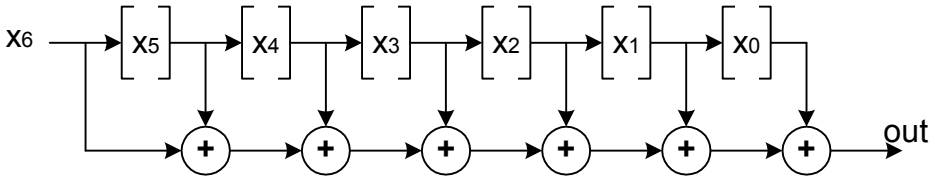

Figure 8 - This dataflow graph adds 7 values and requires three clock cycles.

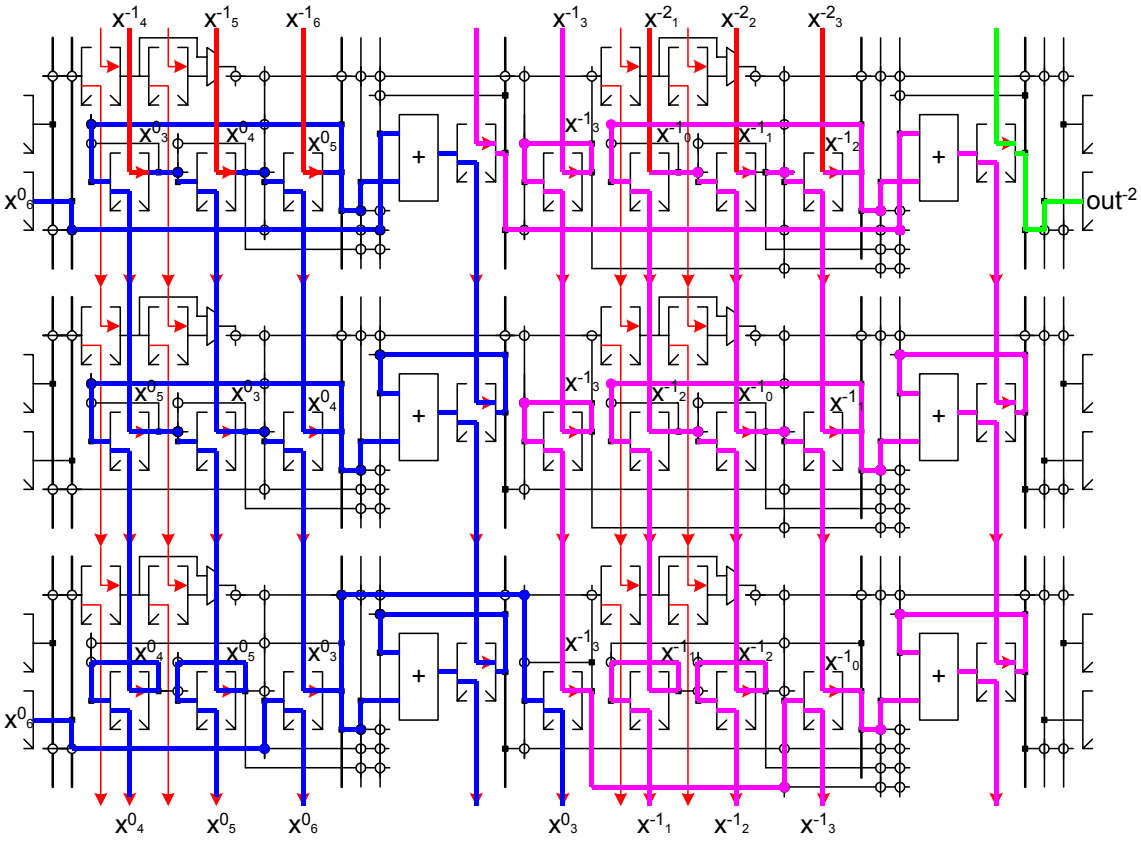

Figure 9 – The dataflow graph of Figure 8 mapped to three copies of datapath graph using a loop initiation interval of size three.

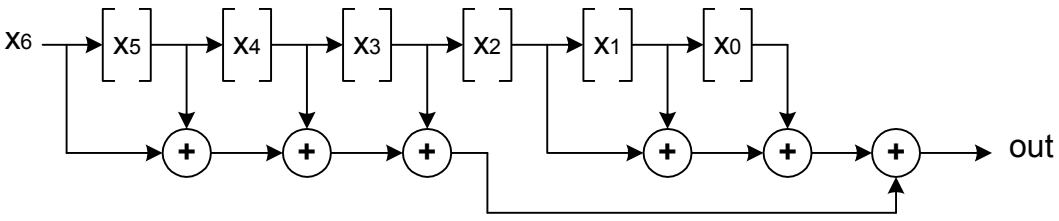

Figure 10 – The dataflow graph of Figure 8 modified to decrease the latency.

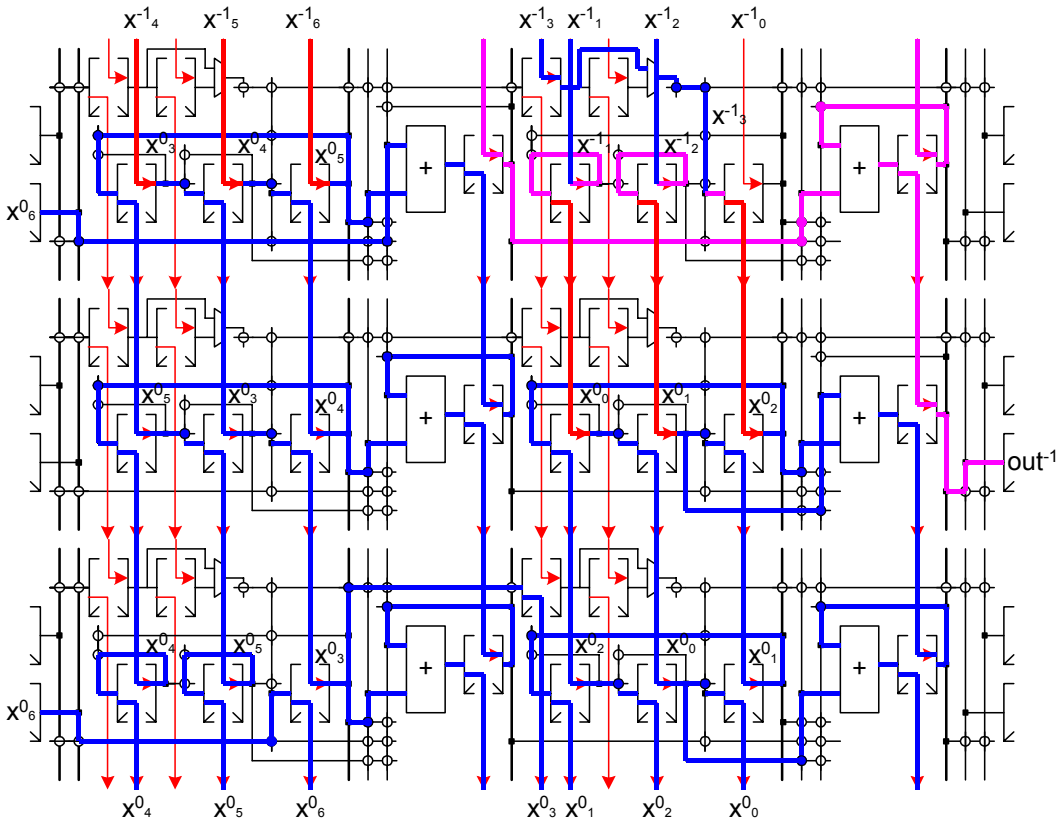

Figure 11 – The dataflow graph of Figure 10 mapped to the unrolled datapath substrate.

This is an example where changing the dataflow graph is necessary to achieve a different mapping to the datapath graph. The dataflow graph of Figure 8 has been rearranged in Figure 10 by changing the order in which the additions are performed. The result of mapping this revised dataflow graph is shown in Figure 11. The programmer can do this by writing the code differently, or the compiler can make the change assuming that the transformation does not change what the dataflow graph computes. The programmer must allow the compiler to make these changes since transformations based on arithmetic identities may change the result of the computation because operators are applied in a different order.

Other dataflow graph transformations can result in different and possibly better mappings. For example, if the order of adds is reversed in this example, an even more efficient solution can be found. It may be possible for the compiler to generate several different possible dataflow graphs, using the scheduler to determine which are feasible and of those, which are most efficient.

Mapping to Function Units

In many cases, a function unit can perform different operations as determined by its control values. For example, an ALU can perform a variety of arithmetic and logical functions. When placement is performed, an operator can be mapped to any function unit that includes it as one of its allowable functions. The placement then assigns a value to the function unit control signals. One way to include this information in the place and route scheduler is to replace the ALU with an array of the functions that it can perform, with extra multiplexers that allow the implementation to choose exactly one. This construction has been used in Pathfinder to allow architecture independent descriptions of FPGA implementation.

Function units can also be pipelined. For example, multipliers are often pipelined to achieve higher throughput. This can be modeled easily by placing an extra register on the output of the multiplier in the dataflow graph that enforces the extra clock cycle.

Mapping to Memories

Memories are used in a variety of ways. Most commonly, they are used as CLSRs, which can be represented in the datapath graph as already described. They can also be used as lookup tables, that is for random reads, in which case an address is presented and data is returned. In this case, the memory is represented as a register, which has an address as input, and data as output. Memory writes are not so easily represented. In this case the memory operator has two inputs, data and address, and has a data dependence edge to all subsequent reads or writes such that the resulting partial order on operators forces the correct ordering on memory operations. These data dependence edges are not part of the datapath graph, but are virtual edges used only to maintain correct ordering of memory operations.

Memories can also be addressed through a function that permutes the address bits to generate address sequences useful for dataflow graphs found in Viterbi and FFT computations. This does not change how the dataflow graphs map to the memories, however, since the address transformation is not part of the dataflow graph.

Predicated Execution

Predicated execution is used to move control operations in the control flow graph into the dataflow graph, thereby avoiding control hazards and increasing the size of the dataflow graphs mapped to the datapath. The compiler performs this transformation via if-conversion. The resulting dataflow graph is augmented with edges that are not data, but control. For example, the status result of an ALU may be used to control a multiplexer or modify the opcode of a function unit. There are different ways predicated execution can be handled. One way is to explicitly include predicated operators in the dataflow graph along with a way to map these operators to the datapath operators. While this makes mapping straightforward, the knowledge required to understand how predicated operators are mapped to the datapath may be extensive and difficult to specify and maintain. It may also be possible to handle predicated execution via a superposition technique. For example, if predication chooses between an add and a subtract, then both operators are instantiated, but may be superimposed on the same function unit if the function unit supports both operators. This allows multiple different types of mappings within the place and route process. If two function units are used, then their outputs must at some point be merged, which can also be seen as superposition, with a multiplexer used to perform the superposition. The construction described for mapping function units can be used to achieve this.

Stitching Data Flow Graphs

Thus far, we have only described scheduling single dataflow graphs in isolation, with the possibility that the dataflow graph is executed in a loop, with loop-carried dependencies solved as part of the place and route scheduling. A dataflow graph typically communicates with other dataflow graphs, getting inputs from previous dataflow graphs and sending outputs to succeeding dataflow graphs. This scheduling is done independently for each dataflow graph in the control/dataflow graph, but then the separate dataflow graph executions must be "stitched" together so the results of one are transferred to the next. If the communicating dataflow graphs are executed sequentially, then they can be stitched together by scheduling them together so that the communication is performed. This can be done by concatenating the substrates for each dataflow graph, and solving the I/O constraints between the two using place and route. In the absence of control flow, the dataflow graphs can overlap in time as long as the data dependence constraints between them are satisfied, as illustrated in Figure 5. Where control flow separates two dataflow graphs, that is, where the successor relationship is determined at runtime, the dataflow graphs cannot be overlapped unless operations executed speculatively do not affect the computation no matter which path is taken.

Of course, by transitivity, this leads to the situation where all the dataflow graphs must be scheduled together, which is intractable for large problems. One possibility is to schedule the dataflow graphs one at a time, in order of importance, with previously scheduled dataflow graphs providing input/output constraints to the following schedules. For example, a loop would be scheduled first since it is executed many times, followed by the scheduling of the initialization dataflow graph. This may lead to expanded schedules for the less important dataflow graphs, that is, schedules that are longer than necessary, but it does partition the problem into tractable subproblems.

This does not address the problem of control that is not sequential. For example, if a branch is executed, then values may be forwarded to two different dataflow graphs at the branch point, and may be received from two different dataflow graphs at the merge point. In additions, if values are forwarded to dataflow graphs that are not adjacent in the control flow, then the communication must be added to the intervening dataflow graphs and scheduled concurrently.

Scheduling for Optimized Control

In the forgoing description, we have shown that the scheduling problem for the Rapid configurable architecture can be formulated as a place and route problem that assigns dataflow graph operations and edges to an unrolled datapath graph function units and wires in both time and space. Representing the problem this way allows many difficult problems to be solved simultaneously. It is clear that the problem of scheduling operations and routing data values can be solved using a place and route solution even where time multiplexing is required. But the place and route based scheduling must solve more than just data dependence and interconnection constraints. It must find a solution that allows the control to be generated successfully. Soft control is optimized after the scheduler has decided on a schedule, and how well this optimization does depends on the schedule. Thus the scheduler must have as part of its cost function an evaluation of how amenable the schedule is to soft control optimization. The components of this evaluation must include the different control optimizations that are possible, as described below. Note that the scheduler must solve the constraints defined by hard control, just like it solves the interconnection constraints. Hard control is configurable control and must be set to the same value for the entire computation, while soft control can be changed from cycle to cycle. Thus even though the scheduler can decide how to set the hard control, that decision must be the same for every cycle, that is either 0 or 1 (along with don't cares). Depending on the number of hard control signals and the flexibility of the interconnect network, this constraint may be more or less difficult to solve. However, it is a hard constraint, just like the routing constraints – either the solution passes or fails.

The following discussion assumes that the reader is thoroughly familiar with the Rapid control architecture described in the document "The Generalized Rapid Architecture Definition". The figure that describes how control is generated in the Rapid datapath is reproduced here as Figure 12.

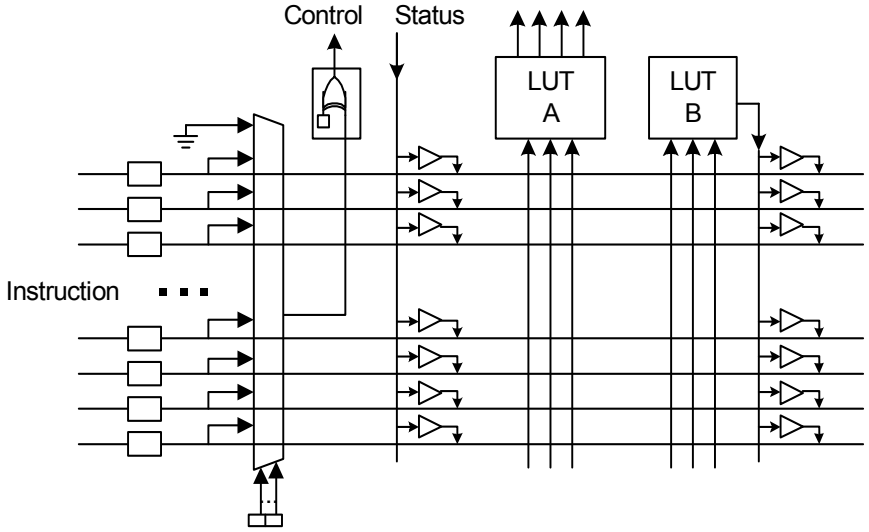

Figure 12 – Components of the Rapid control path.

The Control Matrix

The result of scheduling is a control matrix, which contains an entry for every soft control signal in the datapath for every point in time. This matrix is illustrated in Figure 13. Each column corresponds to a control signal, or group of control signals in the case of symbolic control. Each row corresponds to a clock cycle. The control flow graph, illustrated using arrows to the left of the matrix, is overlaid on this control matrix and determines in what order the different rows are executed. The usual way to implement control is to store this matrix in instruction memory and issue a different instruction on each cycle. This is exactly what we want to accomplish, but we do not want to store the entire matrix or deliver a very long instruction, which might be >1000 bits every clock cycle. The Rapid control architecture takes advantage of the redundancy in this matrix to reduce the instruction size by two orders of magnitude, thereby reducing the size of instruction memory and the number of bits that must be delivered every cycle.

			Control Signals									
1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	0	x	x	x	i3	x	x	x	x	0
1	1	1	0	x	x	x	i0	0	x	x	x	0
1	1	1	0	x	x	i0	x	1	x	0	x	1
0	1	0	0	x	x	i0	x	1	0	1	1	f1
1	1	1	0	x	1	i1	x	0	0	1	0	f1
1	1	1	0	x	1	i1	x	1	0	0	s1	f1
0	1	0	0	x	0	i0	i1	1	1	1	x	0
1	1	1	0	x	x	i1	i2	0	1	1	x	0
1	1	1	0	x	1	i0	i1	1	1	0	s1	0
1	1	1	0	x	1	i1	i2	1	1	1	x	x
0	1	0	0	x	x	x	i3	1	1	1	0	x
1	1	1	0	x	1	x	i0	0	1	1	0	x
1	1	1	0	x	1	x	x	1	1	0	1	x
0	1	0	0	x	0	x	x	1	0	1	1	x
1	1	1	0	x	x	x	x	0	0	1	0	x
1	1	1	0	x	x	x	x	1	0	0	s1	x
1	1	1	0	x	1	i0	i1	1	0	1	s1	x
0	1	0	0	x	0	i2	i3	1	0	1	0	f1
1	1	1	0	x	1	i0	i1	0	1	1	0	f1
1	1	1	0	x	x	i0	i1	1	1	0	0	f1
1	1	1	0	x	x	i2	i3	1	1	1	0	0
0	1	0	0	x	x	i2	i3	1	1	1	0	0
1	1	1	0	x	1	x	i0	0	1	1	1	1
0	1	0	0	x	0	x	i1	1	0	0	1	1
1	1	1	0	x	x	x	x	0	0	1	0	0
1	1	1	0	x	x	x	x	1	0	0	0	0
1	1	1	0	x	1	x	x	1	0	1	0	0

Figure 13 - Example control matrix

Entries in the control matrix are one of the following values:

- 0/1 : The control signal is set to 0 or 1.
- i# : This refers to an input of a multiplexer as a symbolic value. A LUT is used to map a selected set of instruction bits to this value. For multiplexers without a mapping LUTs, this control is 0 or 1.
- s# : This refers to a specific status bit generated in the datapath. This bit must be routed to the control signal on this cycle.
- f# : This refers to a control register in the control path used to implement a FSM.
- X : The control signal has no effect on this cycle and can be set to any value. It is possible for a symbolic value to be a partial don't care. This happens when an input has been routed to more than one input. In this case the symbolic value will appear as (i1 + i3), for example. We will ignore this possibility, at least initially.

The goal of optimization is to compress this matrix to reduce both the size of instruction memory and the number of bits that must be delivered every cycle. The amount of compression possible depends on the amount of redundancy in the matrix. We have found that for computations typical of embedded applications, such as signal and image

processing algorithms, there is a substantial amount of redundancy in the control matrix. Most programs are relatively short, but contain nested loops that iterate many times, perhaps even indefinitely for certain streaming filter computations. This means that the control matrix is very wide but not very deep. This also means that the chance of finding repeated rows in the matrix that can be thought of as "macro-instructions" is fairly low. This may happen if an inner loop is unrolled, but this case can be handled via an instruction count attached to each datapath instruction that repeats the instruction a given number of cycles. This count can also be used to replace loop control for inner loops comprising a single instruction. This simple row compression can be performed first and does not affect column compression.

We will focus instead on compressing the matrix by deleting redundant or unnecessary columns. We will rely on a scheduler that makes the matrix as redundant as possible, leading to substantial optimization. Our goal is to reduce the number of unique columns in the matrix to the number of available instruction bits. Each instruction bit can then be used to drive a separate column. Once this goal has been reached, further optimization is unnecessary. If a program is too complex or the scheduler cannot produce a control matrix that can be sufficiently optimized, the compilation fails and the user must find a different way to write the computation.

Don't Care Analysis

In a large reconfigurable datapath, full utilization of resources is not possible. In particular, more busses are provided than are used on any particular clock cycle. Unused datapath resources appear in the control matrix as don't care control signal values. These X's are valuable, because they increase the amount of redundancy in the control matrix and allow greater optimization. The scheduler must therefore perform a complete don't care analysis based on the schedule it generates. In addition, a partial analysis is necessary during scheduling in order to generate good schedules. This section describes how to generate X's in the control matrix based on an analysis of the resources the program uses on each clock cycle.

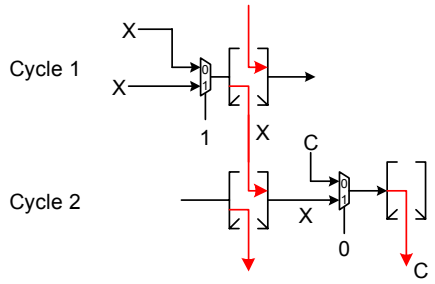

Figure 14 – Space-time datapath fragment from illustrating combinational and sequential don't care analysis

Don't care signals in the control matrix are all those signals that cannot affect the result of the program. Don't cares are classified as either combinational or sequential, and both are illustrated in Figure 14. Combinational don't care control signals do not affect the result of any value that persists past the end of the current clock cycle, that is, any value that is written to a register or affects any state element such as memory or the input and output ports. That is, any value that cannot reach one of the red arrows in Figure 14 that forwards values in one cycle to the next, is a don't care. In the second cycle of Figure 14, multiplexer input 0 is labeled C (care) because it can reach the input of the register. Input 1 is labeled X (don't care) because it cannot affect any register value. Combinational don't care analysis marks the circuit graph backwards from the inputs of all the state elements, discovering those values and elements that are used. A depth-first search backwards through the datapath graph is used to mark all reachable function units or multiplexers from these inputs. All unmarked units are not used in the current clock cycle and their control values can be set to X.

Sequential don't care control signals are those that do not affect any value used on the *subsequent* clock cycle. If a register value is not used in a clock cycle, and the register is loaded with a new value, then the value written to the register on the previous cycle is a don't care, and does not participate in the combinational don't care marking in that cycle. This situation is shown in Figure 14 where the don't care data value on input 1 of the multiplexer in Cycle 2 propagates back to the previous cycle, causing the multiplexer input 1 to be an X instead of a care value. Note that combinational don't care analysis will determine that input 0 is a don't care data value as well.

Don't care analysis thus proceeds as follows: combinational don't care analysis is performed on the last instruction of the program under the assumption that all output data values are cares, but that no other data values are. (This assumes that after the program completes, all register values are discarded.) This marking is then propagated back through the register arrows to the previous instruction. Any state elements for which the arrows cannot be specifically analyzed, for example, memories, are marked as cares. The marking process continues in this way until the entire control matrix has been marked. All unmarked entries are don't cares.

Control flow complicates the analysis somewhat. Where following a red arrow backwards encounters a control flow fan in (e.g. a branch to the following instruction), then both source instructions are marked. If a control flow fanout is encountered (e.g. the previous instruction is a branch), then the instruction is marked, possibly adding to marks already present. That is, a data value is marked as a care if either following instruction marks it. The search backward through the program is therefore done in breadth-first order.

Control Matrix Optimizations

The most important optimization is the division of soft control into static and dynamic control. Static control is the soft control which does not need to be changed during the computation and thus does not need to be generated on the fly by the control path. The static control comprises all those signals that are constant, that is, either 0/X, or 1/X. The first optimization is to remove all constant columns from the control matrix. These control signals can be soft-configured and do not require any instruction bits. The control optimization cannot increase the amount of static control since this is determined by decisions made by the scheduler. Thus the scheduler must have as part of its cost function the number of static control signals and use this to find more efficient schedules. An example of scheduling to increase static control is to use different ALUs for different operators. That is, only adds would be mapped to one ALU and only subtracts mapped to a second ALU. The result is that the controls of both ALUs are static, and there is a good chance that the input multiplexer controls are static as well. This is the reverse of the optimization performed by synthesis algorithms, which attempt instead to reduce the number of function units at the expense of increased control.

Control Signal Sharing

A key feature of the Rapid architecture is the ability to drive an arbitrary number of control signals using a single instruction bit. The next optimization discovers control signals that can share the same instruction bit. Two control signals can share the same instruction bit if they are *equivalent*, that is, if they are either the same for all instructions, or different for all instructions, ignoring cases where either signal is an X. (This optimization is presented first even though it is performed after pipelined signals have been discovered, symbolic control signals assigned and control functions have been decomposed.)

Determining which signals are equivalent is complicated by the fact that the equivalency property is not transitive. That is, signal A may be equivalent to either signal B or signal C, but not both at the same time, because some X value in A must be 0 in one case and 1 in the other. This means that B and C are not equivalent and two instruction bits are required, not just one. While it is easy to determine whether any two signals are equivalent, it is difficult (NP-hard) to determine how to partition the signals into sets of equivalent signals to minimize the number of sets. The equivalence relation defines a graph with an edge between any two signals that are equivalent. The problem of finding the minimum number of equivalent sets is that of finding the minimum number of cliques that cover the graph.

An efficient approximation algorithm can be defined based on the notion of dominance. Control signal A *dominates* signal B if there is no X in A corresponding to a 0 or 1 in B. That is, A and B can be made identical by assigning values to X's only in signal B. Dominance is a useful property because it establishes a hierarchy on the signals, which can be used to partition the control signals. If signal A dominates B and C, then A, B and C can be made identical by some assignment of X's in B and C. The dominance property defines a DAG on the control signals, where there is an edge from signal A to signal B if A dominates B. (Where A dominates B and B dominates A, the two signals can be collapsed into a single signal.) The problem of minimizing the number of sets of equivalent signals then becomes the problem of finding those signals with no in-edges. These then define the set of trees that completely cover the DAG, where each tree is a set of equivalent signals. An approximation algorithm based on clique covering of the equivalence graph will probably give a better result, but it will depend on what the control matrix looks like in practice.

The operation of checking whether two signals are equivalent must be fast. This can be done using the following bit-vector representation for each signal. Let S be the vector for signal S defined by the column in the control matrix. Then ONE_S is a bit vector where $ONE_S[i] = 1$ if S[i] = 1, and 0 otherwise, and $ZERO_S$ is a bit vector where $ZERO_S[i] = 1$ if S[i] = 0, and 0 otherwise, and DC_S the bit vector where $DC_S[i] = 1$ if S[i] = X and 0 otherwise. The following equation can be used to perform the equivalence test:

$$A \equiv B, \; if \; ((ONE_A \wedge \neg DC_B) = (ONE_B \wedge \neg DC_A)) \wedge ((ZERO_A \wedge \neg DC_B) = (ZERO_B \wedge \neg DC_A))$$ The X's in A that must be assigned to 1 is given by $ONE_B \wedge DC_A$, and $ZERO_B \wedge DC_A$ gives the X's that must be assigned to 0. These operations can be performed very efficiently using the bit vector representation.

To include control sharing as part of the scheduler cost function, it must keep an estimate of the number of unique (non-equivalent) signals, which is difficult because of the X values that keep equivalency from being transitive. Finding a fast way to keep track of the number of equivalent signals, or even an estimate, will be necessary to include this in the place and route cost function.

Control Signal Pipelining

When a dataflow graph is mapped to the datapath, it is usually pipelined in some fashion. This means that the dataflow graph starts in one cycle, and continues execution on successive cycles. This means that the control signals for the dataflow graph are skewed in time, so that control signals that occur later are simply delayed versions of an earlier control signal. This is illustrated in the control matrix of Figure 13 in the signals of column 3, 7 and 9. A limited amount of pipelining can be inserted on the control signals in the control path to skew control signals with respect to each other. Columns can thus be shifted earlier in time if that creates new opportunities for control signal sharing.

A column shift can only be performed if it satisfies the control flow. That is, when a control value in the control vector is shifted earlier, it must be shifted to all instructions that branch to the current instruction. This means that all instructions that follow an instruction must have the same control value or an X. When a column is shifted earlier in time, X's are inserted at the end of the vector, and values are shifted out of the beginning of the vector. The X's reflect that no control signals are needed after the last instruction, but values shifted out the beginning of the vector are lost. If these values are X's, or the default reset value, then the shift if legal. Otherwise it is not, since the lost value cannot be generated. (It may be possible to insert a prolog to the program that allows these values to be generated.)

It is not clear how best to optimize the sharing of control signals via pipelining. The following is a greedy algorithm that attempts to increase the number of equivalent signals in the hopes that a smaller cover can be found. First, the equivalence graph is found which contains an edge between two signals if they are equivalent. We then process the connected components one at a time, starting with the smallest connected components, which are single signals that are not equivalent to any other signals. For each connected component, shift all the signals by the amount that adds the most edges to the graph. As edges are added, the connected components must be resorted by size. The algorithm then proceeds through all the connected components of the equivalence graph. Note that if no edges are added to the equivalence graph, then the size of the connected component does not change and will not be processed again, unless an edge is added to it from another connected component. That is, the algorithm is greedy, not iterative. Note that decreasing the number of connected components does not necessarily decrease the number of non-equivalent control signals.

It is crucial to include the effect of control signal pipelining in the scheduler cost function because it can have a very large effect on the control complexity. One possible way to include the effect of pipelining without paying for a complete recomputation for each scheduling decision is to recompute the exact value every once in a while and then estimate how the cost changes until a new recomputation is triggered.

Symbolic Signal Assignment

Symbolic values in the control matrix must be assigned constant values, which are generated by the control path and mapped to the final value used in the datapath. These values can be assigned arbitrarily to maximize the amount of static and shared signal optimization.

Control Signal FSMs

There are LUTs in the control path which can be used to implement simple FSMs. A typical use of such an FSM is to allow a control value to shift from one signal to another under the control of a pair of instruction bits. If 16 signals are operated in the way, it would require 16 different instruction bits instead of just two. Recognizing where this control is possible is not trivial and is probably cannot be a part of the place and route cost function.

Summary

The amount of compression possible depends on the amount of redundancy in the control matrix, which depends on the details of the schedule. Thus the scheduler must incorporate an understanding of the effect of decisions on the ability of the control optimizer to minimize the instruction size. We have described the different control optimizations that are possible and described some initial algorithms for performing these optimizations. However, the scheduler must have some knowledge of the cost of schedules that it produces in terms of the amount of optimization that can be performed on the associate control matrices. This knowledge will be built into the objective function used by the scheduler. Exactly how to include this knowledge in an efficient way is the topic of further research.

Related Work

Considerable research has been done in the area of high-level synthesis, which is concerned with compiling high-level algorithmic descriptions to hardware implementations [10, 25, 26]. The key problems in this area are resource allocation and scheduling. However, traditional high-level synthesis is free to construct an arbitrary hardware implementation, while an adaptable architecture presents a fixed substrate with a fixed set of function units and connections. Although the problems have similar characteristics and we hope to leverage results from this area, solutions do not apply directly. (Since FPGAs present a blank slate that can be used to implement arbitrary hardware structures, high-level synthesis has been applied in the context of fine-grained FGPA architectures. [5, 6, 17])

Compiling to a coarse-grained architecture is very much like compiling to VLIW architectures, since both represent fixed architectures that constrain how the mapping is done. In fact, we intend to leverage VLIW compiler techniques [30, , taking particular advantage of predicated execution to push control flow into the dataflow graphs. However, adaptable architectures have many more parallel function units, a much more constrained interconnection network, contain special resources like embedded memories, and make use of optimized control structures to reduce cost and power consumption. These features make the scheduling problem much more difficult, although ideas like iterative modulo scheduling and software pipelining do apply.

Some of the research on high-level languages for FGPA design has focused on structural languages that allow the designer tight control of the generated architecture [3, 4]. Other languages have been defined that provide constructs that facilitate compiling to hardware, which we intend to do as well.

Many researchers have proposed adding function units based on FPGAs to traditional processor architectures [2, 31, 34, 36]. These function units are configured to execute a sets of instructions in the program that are executed frequently, which corresponds to adding special instructions to the ISA. However, only relatively small parts of the program are implemented this way, and the whole issue of control is finessed.

The place and route problem has been studied extensively. In the context of FPGA architectures, which is most relevant here, simulated annealing is the algorithm of choice for placement, and negotiated routing algorithms like Pathfinder [7, 11, 13] are the algorithms of choice for routing. Not much work has been done on simultaneous place and route [28], largely because it is prohibitively expensive for large problems like FGPA place and route. We expect it to be applicable to scheduling coarse-grained architectures because the problem size is much, much smaller.

Simulated annealing, which is used by almost all placement tools, was proposed by Devadas [37] as a means to optimize scheduling in the context of high-level synthesis. However, only cost and power constraints are included in the objective function since routing is not an issue for high-level synthesis.

References

1. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference on Principles of Programming Languages*, pages 177--189, Austin, Tex., Jan. 24--26, 1983.
2. C. Alippi et al., A DAG-Based Design Approach for Reconfigurable VLIW Processors, in *Proc. IEEE Design and Test Conf. in Europe*, Munich, Germany, 1999, pp. 778-780.
3. M. Aubury, I. Page, G. Randall, J. Saul, R. Watts: Handel-C Language Reference Guide, Oxford University Computing Laboratory, 1996.
4. P. Bellows and B. Hutchings. JHDL - an HDL for reconfigurable systems. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175—184, Napa, 1998.
5. T.Callahan and J.Wawrzynek. Instruction level parallelism for reconfigurable computing. In *Proc. 8th International Workshop on Field-Programmable Logic and Applications* (FPL 1998), September 1998.
6. E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for Reconfigurable execution (SCORE). In *Conference on Field Programmable Logic and Applications* (FPL 2000), pp. 605-614.
7. P. Chan, M. Schlag, C. Ebeling and L. McMurchie. Distributed-memory parallel routing for field-programmable gate arrays, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19, No. 8, pp. 850-862, Aug. 2000
8. D. Cronquist, P. Franklin, S. Berg, and C. Ebeling. Specifying and Compiling Applications for RaPiD, In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1998.
9. D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths, In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, Atlanta, pp. 23-40 , April 1999.
10. G. Doncev, M. Leeser, and S. Tarafdar. High level synthesis for designing custom computing hardware, in *Symposium on Field-Programmable Custom Computing*, April 1998.
11. C. Ebeling, L. McMurchie, S. Hauck, and S. Burns. Placement and Routing Tools for the Triptych FPGA, *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 473-482, December 1995.
12. C. Ebeling, D. Cronquist, P. Franklin, J. Secosky, and S. Berg. Mapping Applications to the RaPiD Configurable Architecture, In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1997.
13. C. Ebeling and L. McMurchie. Pathfinder: A Negotiation-Based Router for Routing-Constrained FPGAs, In *Proceedings of the Third ACM Symposium on Field-Programmable Gate Arrays*, Monterey, pp. 111-117, February 1995.
14. C. Fisher, K. Rennie, G. Xing, S. Berg, K. Bolding, J. Naegle, D. Parshall, D. Portnov, A. Sulejmanpasic, and C. Ebeling. "An Emulator for Exploring RaPiD Configurable Computing Architectures," In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (FPL 2001), Belfast, pp. 17-26, August, 2001.
15. S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer, PipeRench: A Coprocessor for Streaming Multimedia Acceleration, In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.
16. M.B. Gokhale, et al. Napa C: Compiling for a Hybrid RISC/FPGA Architecture, *FCCM 98*.
17. J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor, in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1997, pp. 12-21.
18. M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, An Efficient General Cooling Schedule for Simulated Annealing, Proc. *ICCAD-86*, Santa Clara, 381-384, 1986.

19. W. Hwu, R. Hank, D. Gallagher, S. Mahlke, D. Lavery, G. Haab, J. Gyllenhaal, and D. August. Compiler Technology for Future Microprocessors, *Proceedings of the IEEE*, December 1995, pp. 1625-1640.

20. S. Kirpatrick, C. Gelatt, and M. Vecchi (1983) Optimization by simulated annealing. Science 220, pp. 671-680.

21. M. Lam, Software pipelining : an effective scheduling technique for VLIW machines, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, 1988, pp. 318—328.

22. W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar and S. Amarasinghe, Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine, *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII), Oct. 1998.

23. G. Lu, H. Singh, M. Lee, N. Bagherzadeh, F. Kurdahi and E. Filho, The MorphoSys Parallel Reconfigurable System, *European Conference on Parallel Processing*, pp. 727-734, 1999.

24. S. A. Mahlke et. al., Effective compiler support for predicated execution using the hyperblock, in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45-54, December 1992.

25. M. C. McFarland, A. C. Parker, and R. Camposano, The high-level synthesis of digital systems, in *Proceedings of the IEEE*, vol. 78, pp. 301—318, 1990.

26. G. D. Micheli, Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.

27. E. Mirsky and A. Dehon, MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources, In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157-166, 1996.

28. S. Nag, and R. Rutenbar. Performance-driven simultaneous place and route for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(5):499—518, June 1998.

29. R. Nagarajan, K. Sankaralingam, D. Burger and S. Keckler, A Design Space Evaluation of Grid Processor Architectures, In Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, 2001.

30. B. Rau, Iterative modulo scheduling: An algorithm for software pipelined loops, in *Proc. of 27th Annual International Symposium on Microarchitecture*, pp. 63--74, Dec.1994, San Jose.

31. R Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings MICRO-27*, pages 172—180, 1994.

32. W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. High-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 69-78, Feb. 1999.

33. G. Weaver, B. Cahoon, J. Moss, K. McKinley, E. Wright, J. Burrill. The Common Language Encoding Form (CLEF) Design Document, University of Massachusetts at Amherst Technical Report TR 97-58.

34. R. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. FCCM, 1996.

35. M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

36. Z. Ye, P. Banerjee, S. Hauck, and A. Moshovos. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Unit, in *Proceedings International Symposium on Computer Architecture*, Toronto, CANADA, June 2000.

37. S. Devadas and R. Newton. Algorithms for Hardware Allocation in Data Path Synthesis, *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 7, July 1989.