Queue Machines: Hardware Compilation in Hardware

Herman Schmit, Benjamin Levine and Benjamin Ylvisaker
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{herman,blevine}@ece.cmu.edu

Abstract - *In this paper, we hypothesize that reconfigurable computing is not more widely used because of the logistical difficulties caused by the close coupling of applications and hardware platforms. As an alternative, we propose computing machines that use a single, serial instruction representation for the entire reconfigurable computing application. We show how it is possible to convert, at runtime, the parallel portions of the application into a spatial representation suitable for execution on a reconfigurable fabric. The conversion to spatial representation is facilitated by the use of an instruction set architecture based on an operand queue. We describe techniques to generate code for queue machines and hardware virtualization techniques necessary to allow any application to execute on any platform.*

I. Introduction

To motivate this work, we will first consider the difficulties of a software developer creating applications for a reconfigurable computing platform. We will assume that the target reconfigurable computing platform looks like most such systems described in the literature: a microprocessor-based component integrated with an FPGA-based component. The application representation now consists of at least two components: one corresponding to the microprocessor system, and one corresponding to the FPGA-based system. Both the microprocessor and FPGA components require significant interface code. The overhead imposed by the interface code means that parallel sections of code must last longer (e.g. have more loop iterations) to justify the setup and communication time. The heterogeneity of the application code as well as the dependence on the interface code makes this software very tightly coupled to the hardware platform. Any variation in the hardware platform will make the application difficult to run on the hardware. Upgrading the hardware platform will require re-compilation of all applications, which is not desirable from a business or logistical perspective, and could be technically impossible. No software developer would invest time and effort developing for such a platform if it creates such logistical nightmares.

As an alternative, we propose an approach that streamlines these logistics by using a single representation for the entire application. This representation will look like conventional software in that it will be a sequence of instructions with a serial semantic. The application is initially executed in a serial mode, but when the serial machine encounters a loop with parallelism, a spatial representation of this loop body is created for execution on reconfigurable hardware in subsequent loop iterations. As illustrated in Figure 1, such an architecture would have a serial execution engine and a reconfigurable hardware component, which we call the spatial execution engine. Both engines share access to architectural resources like the memory space. A "compilation engine" would create spatial implementations from the serial code. When the compilation of a spatial portion of code was complete, the serial engine would suspend and the spatial engine would continue the execution of that portion of code. When that loop exits, control would return to the serial engine.

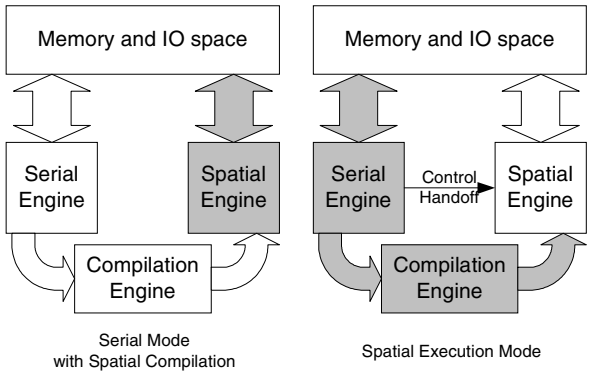


Fig. 1. Modes of execution in integrated machine

This approach means there is only a single representation for the application, using a single model of computation. Conversion to a spatial representation takes place in the hardware itself on every execution, which means that there are no binary management or compatibility issues. Finally, since there is one representation of the executable and a shared interface to the remaining components of the architecture, there are no interface issues. Compilation of programs is easier because there is one architectural view of memory and I/O, there is no interface synthesis necessary, and the entire program, including the parallel portions, is expressed as a sequence of instructions.

A. Illustrative Example

This vision seems fantastically difficult to achieve, especially considering the inefficiency that is often attributed to FPGA compilation. This section will give an example of what

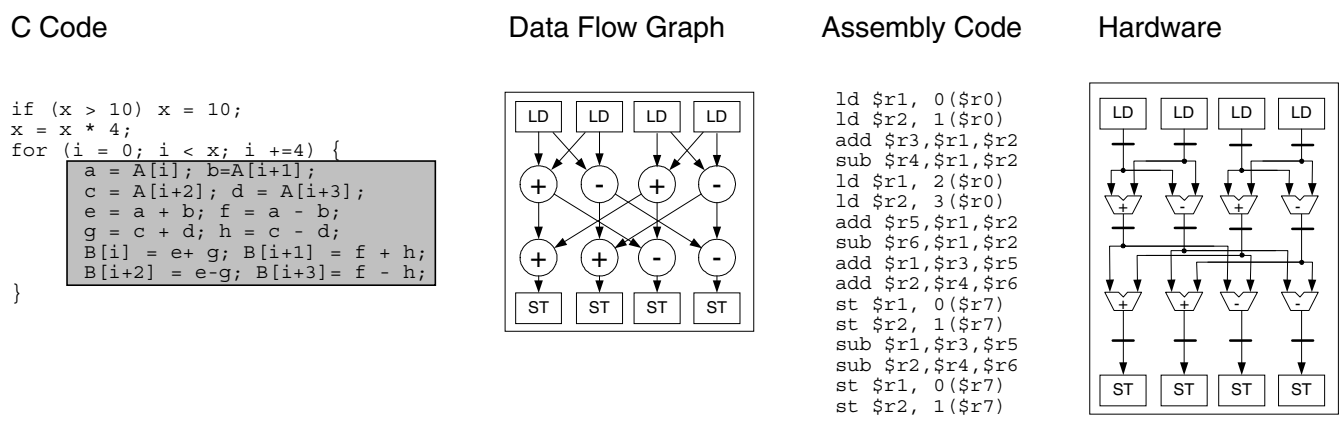


Fig. 2. Initial C code for a simple loop, the data flow graph for the loop body, and the assembly code generated by the compiler. The hardware required to implement this block "looks like" the data flow graph, but is hard to recover from the assembly language.

we want to achieve, and simultaneously illustrate the difficulties accomplishing this goal using a typical instruction set architecture (ISA). The subsequent sections of this paper will illustrate how this is possible using a different kind of ISA.

The C code in Figure 2 consists of some setup code and a loop. Inside the loop, a calculation is performed on four values read from memory. There are no memory-carried dependencies or loop-carried dependencies in this loop. The data flow derived from this code shows all eight memory accesses and eight arithmetic operations as nodes connected by dependencies. Our objective is to have the serial engine execute the code outside the **for** loop, then execute one iteration of the loop body while the compilation engine creates the hardware structure at right in this figure. On subsequent loop iterations, the hardware design is executed to perform the loop body. Assuming there are adequate hardware resources and memory ports, the hardware can initiate and complete one loop iteration per cycle. Once the loop is done, control passes back to the serial engine.

The structure of the desired hardware design strongly resembles the structure of the data flow graph, with registers inserted to pipeline the computation, and functional units replacing the operators. The dependency arcs in the DFG are associated with wires connecting registers and functional units. The DFG is created by the compiler, which uses it to generate machine code. In the example, we show MIPS-like code for this loop body. Unfortunately, the DFG structure is almost impossible to recover from this machine code. The compiler has done a number of things that obfuscate the structure of the hardware. First, it has re-ordered code to minimize register usage. This re-ordering pushes independent instructions away from each other. For example, every other instruction in the loop body depends somehow on the load instructions. Yet to save a pair of registers, the second pair of load instructions is not performed until after the first pair of additions and subtractions. This obscures the fact that the load instructions should all be executed as soon as possible to enable the execution of the maximum number of additions and subtractions. In order to further conserve registers, the compiler assigns one register to multiple arcs in the DFG. In order to recover the dependency information, a compilation engine would have to be able to look through all instructions in the loop body to discover the last use of an input operand. Finally, in this assembly language code example, six architectural registers are used. If the loop body was significantly larger, it would likely use more registers. At some point it might exceed the number of registers provided by the architecture, in which case it would "spill" the registers to memory. This register spill further obscures the data dependencies, and places a final limit on the size of the hardware block we could possibly reconstruct from such code.

While the DFG has the hardware structure embedded, the realities of the ISA forces the compiler to destroy the structure. This paper will demonstrate that given the proper instruction set architecture and techniques of hardware virtualization, the structure of a hardware design can be preserved, even in an ISA with sequential semantics. The key ISA innovation is the use of a an operand queue, rather than a stack or register file, to express the dependencies between operations in the computation.

The next section will describe the basic queue machine in terms of its sequential semantics. Section III will describe how it is possible to generate queue machine code from a general data flow graph, and Section IV will describe an algorithm to "place-and-route" a queue machine representation of an algorithm on a virtual datapath fabric. Section V will also discuss how this virtual datapath fabric can be executed on a physically constrained hardware fabric.

II. Queue Machines

Our objective is to create an ISA that can be used to capture complete applications. The applications we are interested in capturing do not run fast enough on conventional processors, and contain segments of code with substantial parallelism.

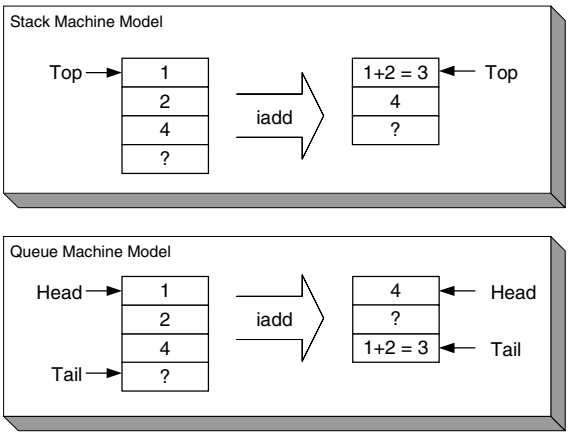


Fig. 3. Stack and Queue Machine Models

These are the target applications for reconfigurable computing platforms. Like any ISA, we need mechanisms to express control flow and mechanisms to express data flow. An ISA designed for reconfigurable applications must, in addition, facilitate detection of highly parallel loops and conversion of the data flow in the program into a spatial representation.

In order to enable the compilation engine to identify portions of code for hardware compilation, we require instructions that indicate the boundaries of parallelizable loops. Our machine uses two instructions for this purpose: **`loopbegin`** and **`loopend`**. This **`loopbegin`** instruction initializes and increments an index variable for the loop, using a specified minimum and maximum bound for the loop index. Between the **`loopbegin`** and **`loopend`** instructions, there can be no other control flow instructions like branches or other nested loops. If any of these instructions are encountered, the compilation to hardware is aborted. These instructions have a clear sequential semantic and can be executed serially on the first iteration while compilation of the loop body is performed.

In addition to a mechanism for expressing and identifying the portions of code that map to hardware, we need a mechanism for representing the data flow of the computation. In our machine, we are going to use an operand queue for expressing the data flow for a program segment. A machine that uses an operand queue for expressing the structure of the computation is called a *queue machine*. An operand queue is similar to the more familiar operand stack. A machine with an operand stack works by obtaining input operands by popping them off a stack. The result (or results) of the operation are pushed back on the top of the stack. The queue machine works similarly, except that the inputs for an operation are read from the head of the queue, and the results are pushed onto the tail of that queue (which is at the other end of the queue from the head). The two machines are compared in operation in Figure 3.

Machines with a single operand stack can be used to represent acyclic data flow graphs that either: have a undirected covering graph that is a tree, or have an undirected covering graph that contains at most one cycle with that cycle having a directed Hamiltonian path [3]. This is an advantage for a compiler, as most expressions are parsed as trees and convert to stack operand form easily. However, this limits the class of data flow graphs that one can successfully express. Ideally, we want a machine that can capture any directed acyclic graph (DAG). Stack machines such as the JVM [9] have stack manipulation instructions that can, for example, pop an operand off the stack and push two copies of that operand on the stack. This extends the capabilities of the stack so that a subtree can have multiple fanouts, but it does not make it possible to capture general data flow graphs without the use of a separate memory or scratch space.

Programs for queue machines can represent all data flow graphs that have an embedding that is a *directed arched leveled-planar graph* [3]. We will consider a subset of this class of graphs, DAGs with an embedding that is a leveled-planar graph. A leveled graph is one in which there exists a mapping **A** of vertices to integers such that if there is an arc from vertex u to vertex v, then $\mathbf{A}(u) = \mathbf{A}(v)+1$ for all arcs in the graph. Visually, a leveled graph can be drawn with the operations in rows such that every arc goes from an operators in one row to an operator in the next row. No arc skips a row. An example of a leveled graph is shown in Figure 5b. Planarity means that it is possible to draw the data flow graph without any dependency arc crossing another dependency arc. A leveled-planar graph is a graph that is both leveled and planar. Examples of leveled-planar graphs are shown in Figure 5c and Figure 5d. The class of leveled-planar data flow graphs includes all rooted directed trees, so the queue machine is as useful for a compiler as a stack machine. Section III will discuss how all data flow graphs can be converted to a planar representation through the use of queue manipulation instructions.

The sequence of instructions on a queue machine has a number of advantages for parallel and spatial execution. To illustrate this, consider the stack and queue instruction schedules for the eight-operation data flow graph in Figure 4. The stack machine does a depth-first traversal of this graph (which is a rooted tree), and the queue machine does a breadth-first traversal of the graph. There are some sets of instructions in this graph that are independent and can execute simultaneously. In the queue schedule, the independent instructions such as Op_1, Op_2, Op_3, and Op_4, are adjacent to each other in the code. In the stack machine, dependent (rather than independent) instructions are close and as a result, independent instructions are sometimes forced away from each other. As the size of the graph grows, so will the distance between independent instructions in a stack machine representation.

The representation of a data flow graph in a queue machine forces all independent instructions at the same level in the graph to be in one contiguous block of code. This fact has benefits when trying to convert this program to a spatial exe-

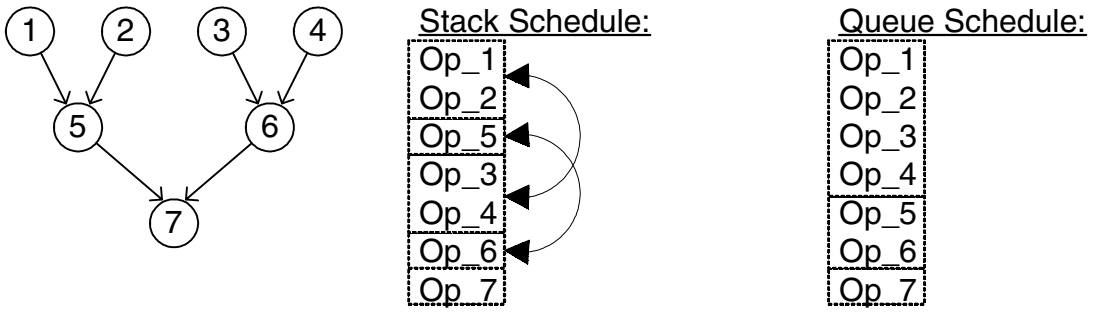


Fig. 4. Stack and Queue Machine Schedules: A tree DFG scheduled on a stack machine and queue machine. The independent operations are distant from each other in the stack schedule, and consecutive in the queue schedule.

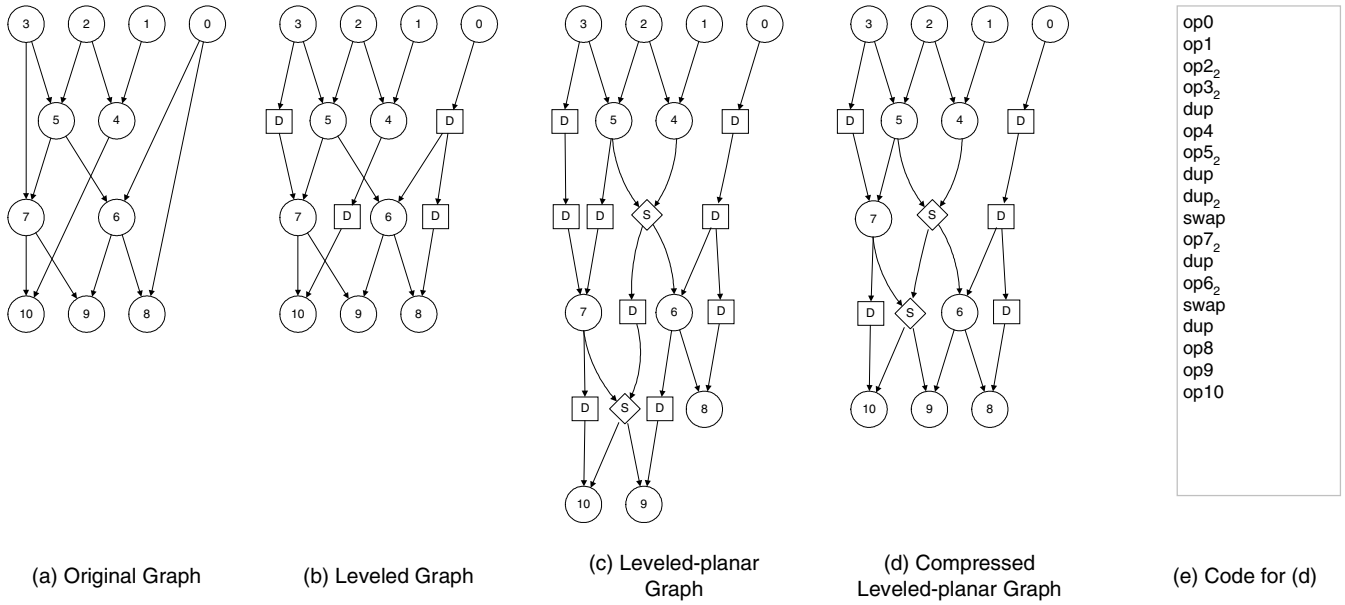


Fig. 5. Graph Transformations: (a) general DAG, (b) leveled DAG, (c) leveled-planar, (d), compressed level-planar graph, and (e) code generated by traversal of part (d).

cution model. If the graph in Figure 4 were mapped to an architecture like PipeRench [1], parallel operations like 1-4 and 5-6 would be placed on a single stripe, assuming adequate hardware resources. The sequence of instructions in the queue machine parallels this structure exactly. In Section IV we will describe how the levels of the graph can be recovered from one serial execution of the code, and how different independent instructions can be assigned to functional units on a reconfigurable fabric.

III. Queue Machine Code Generation

Generating queue machine code for a basic block of a program involves creating a data flow graph for that basic block, converting a data flow graph into a level-planar data flow graph, and then performing a breadth-first traversal of the graph. The first phase, creating the data flow graph, is a general compiler topic. The conversion of the data flow graph to a level-planar graph can be broken into a levelization phase followed by a planarization phase. To illustrate the process we will use an example shown in Figure 5. In order to complete the mapping we will need two special operators: a duplicate operator for levelization (**dup**), and a swap operator for planarization (**swap**).

Levelization is easy. A topological sort of the graph is performed, as shown in Figure 5(a), so that all nodes that have a maximum path of length n from the source node or nodes in the graph are on the nth level. In a topologically sorted graph, there can be arcs that span one or more levels. For example, in Figure 5(a), arcs from operation 0, 3, and 4 span more than one level. Special operators, called duplicate instructions or (**dup**), are added to make all arcs go from one level to the subsequent level. In the queue machine, the semantics of the **dup** operator is to take an operand from the head of the queue, and write it to the tail of the queue. A variation of this operator, $\mathbf{dup_2}$, places two copies of the input operand on the tail of the queue.[1] Figure 5(b) shows a levelized version of the

graph that has been obtained by adding three **dup** instructions and one $\mathbf{dup_2}$ instruction.

This level graph is not planar. In our drawing in Figure 5(b), two arcs cross. We first attempt to find an ordering of nodes on each level that produces a planar embedding. We use the algorithm described in [5] to do this. Assuming there is no planar embedding, the crossings can be replaced by a **swap** operator, shown as a diamond with an "S" in Figure 5(c). In the queue machine, a **swap** instruction reads two operands, first *x* and then *y*, off the head of the queue, and writes *y* and then *x* on to the tail of the queue. We use a heuristic algorithm to minimize the number of arc crossings in the drawn graph, and then we replace each crossing with a **swap** instruction. The **swap** operator may make the graph planar, but can destroy the level property. Therefore, the process of creating a level-planar graph requires a re-levelization phase after **swap** instructions are used to create planarity. Figure 5(c) shows a level-planar graph generated from Figure 5(b) using two **swap** operators and five **dup** operators. After planarizing the level graph, the graph can usually be compressed by looking for cut sets of the graph composed entirely of **dup** nodes. The removal of these **dup** nodes results in a compressed level-planar graph shown in Figure 5(d).

The code sequence in Figure 5(e) is obtained by a breadth-first traversal of the graph in Figure 5(d). In this case the code length of the is 18 instructions, which means that almost half of the instructions are ones that manipulate the queue only (**swap** and **dup**).

IV. Hardware Compilation from Queue Machine Code

Compilation of this code at run time to a spatial representation requires an algorithm to generate placement and routing information for each instruction in the graph. To deal with the general case, we present an algorithm that targets an infinitely large fabric of functional units. If the virtual hardware design created by this algorithm is too large for the physical fabric, we will use hardware virtualization techniques to execute it on the hardware that exists. We will first describe the generation of the virtual and unbounded hardware.

Our virtual hardware target consists of an unbounded, two-dimensional matrix of functional units. Each functional unit is capable of executing every loop-legal instruction, including memory loads and stores. We assume that the data flow graph has operators that consume no more than two operands. The interconnections of these fabric elements is as shown in Figure 6. Each row in the fabric obtains operands from the interconnection network, which connects the outputs of the previous row of functional units. There are no connections between functional units on the same row, therefore they can only execute instructions that are independent of each other. The outputs of the functional units connect to registers, and the functional units can be arbitrarily pipelined. We first assume the interconnection network is complete.

1. We use the same convention for all queue instructions. A subscript means that multiple copies of the result of the operation are placed on the tail of the queue.

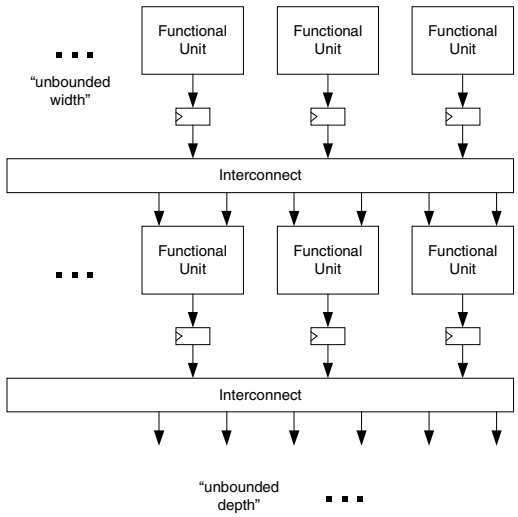


Fig. 6. Hardware Fabric Block Diagram: This is our virtual hardware target, unbounded in both dimensions.

```
row = 0
this_q = 0, next_q = 0
foreach op {
    if (this_q – inputs(op) < 0) {
        row ++
        place(op,row)
        this_q = next_q – inputs(op)
        next_q = outputs(op)
    } else {
        place(op,row)
        this_q -= inputs(op)
        next_q += outputs(op)
    }
```

Fig. 7. Row placement algorithm.

To create a placement for a code segment, one must determine, for each instruction, the row and column of the functional unit assigned to that instruction. The row is determined on the first serial execution of the loop body by monitoring the status of the queue. We need to know the number of input and output operands from each instruction. We keep track of two variables: **this_q**, which indicates the number of operands remaining for this row of instructions; and **next_q**, which indicates the number of operands in the next row of instructions. The pseudo-code for determining the row placement is shown in Figure 7. These variables determine dependencies between instructions. Figure 8 shows the process of hardware compilation for the example code from Figure 5.

Instruction	Operand in	Operand out	Queue Length this_q	Queue Length next_q	Action
op0	0	1	0	0 -> 1	row = 0
op1	0	1	0	1 -> 2	row = 0
$op2_2$	0	2	0	2 -> 4	row = 0
$op3_2$	0	2	0	4 -> 6	row = 0
dup	1	1	6 - 1 = 5	1	row = 1
op4	2	1	5 -> 3	1 -> 2	row = 1
$op5_2$	2	1	3 -> 1	2 -> 3	row = 1
dup	1	1	1 -> 0	3 -> 4	row = 1
dup_2	1	2	4 - 1 = 3	2	row = 2
swap	2	2	3 -> 2	2 -> 4	row = 2
$op7_2$	2	2	2 -> 0	4 -> 6	row = 2
dup	1	1	6 - 1 = 5	1	row = 3
$op6_2$	2	2	5 -> 3	1 -> 3	row = 3
swap	2	2	3 -> 1	3 -> 5	row = 3
dup	1	1	1 -> 0	5 -> 6	row = 3
op8	2	0	6 - 2 = 4	0	row = 4
op9	2	0	4 -> 2	0	row = 4
op10	2	0	2 -> 0	0	row = 4

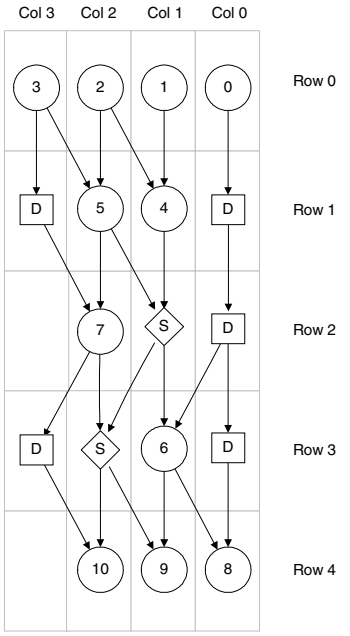


Fig. 8. Hardware Compilation Example.

There are a number of options for determining the column placement within the row. The simplest technique is to have a counter, which is initialized to zero when a new row is allocated. This counter is incremented by one every time a new instruction is mapped without allocating a new row. We call this approach the right-justified row placement algorithm. This approach is illustrated in the example in Figure 8.

In this example, the interconnect requirements for the fabric are local. A functional unit at (*x,y*) need only obtain operands from the functional unit directly above itself (*x,y-1*), or above and one column to either the right or left (*x-1,y-1*) or (*x+1, y-1*). Unfortunately, this is not true in general. For example, in Figure 9, one row contains a number of store instructions (which generate no output). Using the right-justified row placement algorithm, the placement shown in Figure 9 would require an interconnection network where the functional unit at (*x*,*y*) can access functional units at (*x*-1,*y*+*k*), where 0<*k*<5. In general, assuming there is no bound on the fanin or fanout of instructions, there is no bound on the length of the interconnect between rows.

There are two potential solutions to this problem. The first solution is to use a more sophisticated column placement algorithm. One such algorithm would assign a column to an instruction by averaging the column assignments of the instructions that generate the instructions operands. This calculation simply requires an adder. In the example in Figure 9, this approach has reduced the length of the longest interconnect to a span of two columns. In the worst case this algorithm reduces the length of the longest interconnect to the

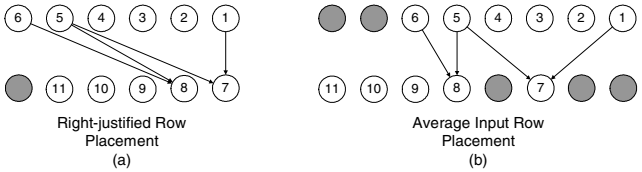


Fig. 9. Row Placement Options

width of the previous row. This approach is also not as efficient in utilizing functional units, as illustrated in Figure 9.

The second solution to limiting the interconnect length is to make the placement explicit in the original code by adding adequate **dup** and **nop** instructions where necessary. The **nop** instructions are fillers. They do not affect the queue. They only serve to consume a column and space the instructions correctly to enforce local connections. Figure 10 shows the worst case scenario from Figure 9 with additional operators added to assure the interconnection network is local. The nop instructions are shown as grey circles. This approach bloats the serial code, hurting the performance of the initial serial execution, but can be used to significantly improve performance, decrease the area required by the hardware fabric, and make it much easier to virtualize this hardware, as we will discuss in Section V.

Table I shows, for a number of different kernels, the number of operators, the number of instructions required to generate a valid queue machine representations (the number of nodes in the level-planar graph), and the number of instructions required to generate an interconnect limited version of

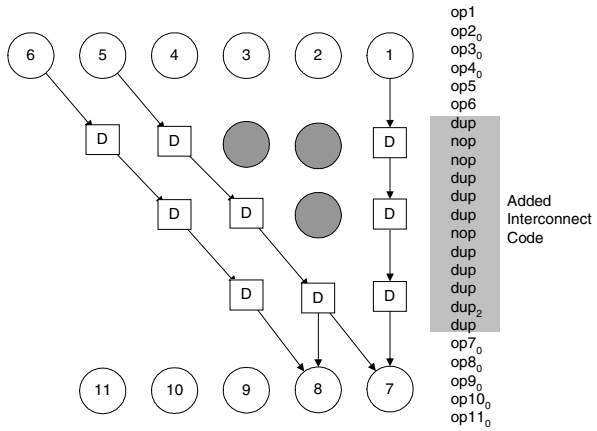


Fig. 10. Explicit code for limited interconnect

this level-planar graph, where only nearest column connections are required. The instruction set for this machine supports 16 bit additions and subtractions, a 16 bit x 4 bit multiplication, a 6 bit table lookup, memory load and store, and various logical operations.

Transforming the graph to a level-planar representation requires a significant increase in the number of instructions in the representation. It should be considered that because the instructions have no explicit operand specification (such as three register file addresses) the overall size of the executable may not be much bigger than one that uses a register file. For example, the queue machine instructions in these examples could be encoded in a byte, while the register file machine, because some of these applications have almost 64 live variables at a time would require at least 18 bits for operand specification (three operands, six bits each) and six to eight more bits to specify the operation. Thus, the register file operands would be at least three times as large as a queue machine instruction. Undoubtedly, the queue machine representations would be significantly more dense than the configuration bitstream for an FPGA or an architecture such as PipeRench. Finally, this bloated code is only required when the applications developer wants that section of code to be mapped to parallel hardware. If this is a serial portion of the application, the general queue machine structure can be used. The results in Table I could be significantly improved by implementing a more sophisticated compression phase that executes after the initial transformation to a level-planar representation.

Some applications have complex, non-local interconnect that makes the queue machine implementation particularly large. The best example of this is the **fft8_iterative**, which has some stages that are local, and other stages that require connections from one extreme edge of the row to the other edge. There is no way to reduce this connectivity within the algorithm. Other application kernels, like **rc6** and **dct1** have much lower connectivity requirements and therefore require a smaller level-planar graph.

TABLE I Operation Counts and Depth of Application Graphs

	Original DFG		Level-planar Graph (LPG)		Limited Interconnect LPG	
Kernel	Ops	Depth	Ops	Depth	Ops	Depth
dct1	122	18	537	49	568	49
fft8_iterative	88	7	909	41	945	41
haar16	124	6	918	17	940	17
rc6	74	23	330	42	338	42
idea	303	160	1462	235	1462	235
popcount	31	5	229	24	237	28

On average, making an explicit interconnect limited version of the graph only requires 2% more instructions than the general queue machine expression. This is lower than one might initially expect, but it must be considered that what usually causes wide fanouts is operations that have zero inputs or zero outputs, such as loads or stores. In these applications kernels, there are few intermediate loads or stores. Almost all the data must be acquired before we begin operations and most data to be written back into the memory is ready at the same time. Thus, most loads and stores are forced towards the top or bottom of the DFG, and as a result the unfavorable scenario in Figure 9 rarely occurs. This result leads us to believe that writing queue machine code that is interconnected limited is a worthwhile trade-off, considering the reduced hardware costs and the benefits to virtualization, as described in the following section.

V. Hardware Virtualization

The virtual hardware design may be too large for the physical device it is running on. In addition, different variations of the hardware platform might have different hardware capacities. Hardware virtualization techniques are necessary to execute these designs on a physically constrained hardware component. Hardware virtualization works by time-multiplexing physical resources. An ideal hardware virtualization technique would execute a virtual application of size v, on a physical hardware of size p ($p < v$), with a performance degradation of p/v. We will consider virtualization in the two unbounded dimensions of the fabric in Figure 6. First we will deal with the virtualization of rows, then the virtualization of columns. The virtualization techniques can be applied independently to allow the hardware fabric in Figure 6 to be emulated on a physical fabric with bounds in both directions.

Row virtualization is performed in a manner identical to pipeline reconfiguration in PipeRench [1]. The basic idea is that the configuration information for a pipeline stage, or stripe, is stored in a separate memory. By reading these configuration words from the configuration memory, and storing it in the physical fabric, a stage of the pipeline can be reconfigured every cycle. Pipeline reconfiguration works by configuring one stage ahead of the data in a pipeline. After p

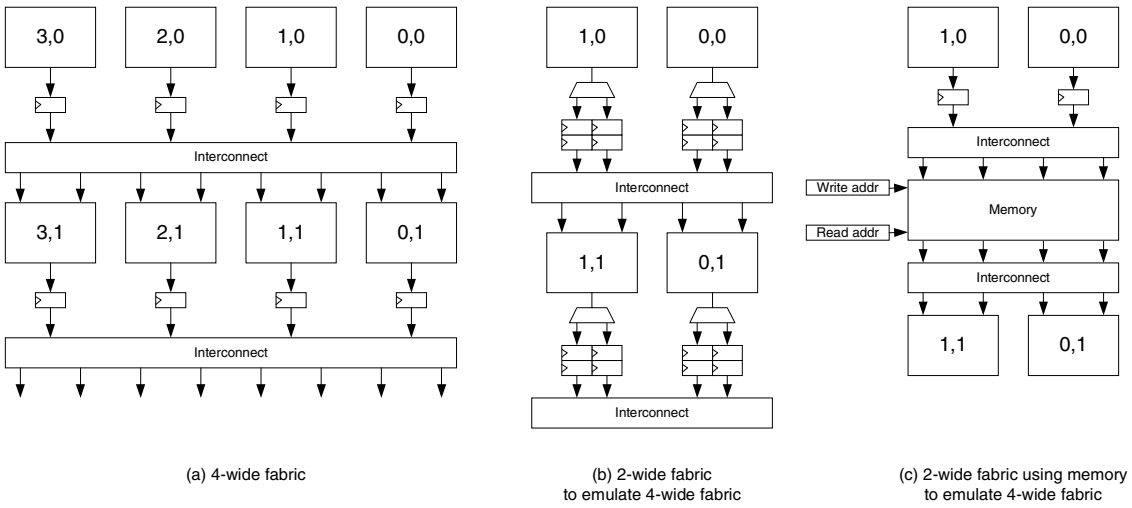


Fig. 11. Width virtualization options for general queue machines: (a) virtual fabric with width of 4, (b) 2-wide fabric without memory, (c) 2-wide fabric using memory.

cycles, when the physical fabric with p stages is fully configured, the first stage is reconfigured to be virtual stage p+1. Reconfiguration proceeds, so that for a virtual pipeline consisting of v stages, on cycle t, virtual stage t (mod v) is configured on physical stage t (mod p). Using this approach, nearly linear performance degradation is possible. In PipeRench, a pipeline stage takes one cycle to reconfigure. Concurrently with that reconfiguration, p-1 stages are operating on data. Therefore the overhead of virtualization is a single physical stage and the area necessary for storage of the virtual design.

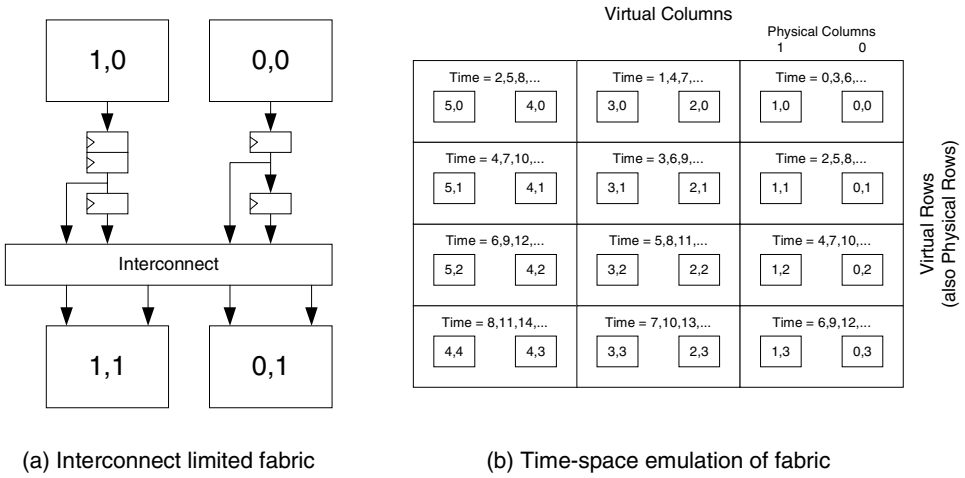
Virtualization within a row is facilitated by the queue machine model because all of the operators on a single row are independent. Figure 11 illustrates a fabric with a width of four functional units and two ways to build a fabric with a width of two functional units that can emulate that 4-wide fabric. In this case each cycle of the four-wide fabric is emulated in two cycles of the two-wide fabric. In Figure 11(b) an additional level of registers is required between rows so that one row may perform the computation of one "virtual cycle" before passing its results to the next row. Because the general queue machine model requires complete inter-row interconnect the fabric in Figure 11(b) has complete interconnect between the entire row of eight potential operands from the previous row. This clearly does not scale well because the interconnect limits the width of a row that can be emulated on a given fabric. In addition, there must be adequate registers to support the complete width of the widest row. A better approach to building a emulation fabric for a wide general queue machine is shown in Figure 11(c). This implementation uses a memory to emulate the complete interconnect of the inter-row interconnect. Since the graph is planar, all operands for a particular row can be written into a contiguous section of memory and read back from that memory. The use of the memory also allows for greater storage density of the results from the previous row, and makes it possible to support a very wide virtual row.

If the fabric is interconnect limited, as described in Section IV, it is possible to construct a much more scalable real fabric without use of a memory between each row. Because every column in this fabric need only access one column to the left or right, a smaller fabric needs to be able access only the left most operand from the "next" micro-cycle and one operand to the right in the previous micro-cycle, as illustrated in Figure 12(a). This design allows virtual emulation to proceed in a diagonal manner through the virtual design, as illustrated in Figure 12(b).

VI. Related work

Queue machines have been invented and re-invented a number of times. Sometimes they have been viewed as the curious dual of the popular stack machine [8]. Other researchers have noticed the fact that in a queue machine the operands and instructions are aligned with each other, which makes it possible to build an efficient superscalar or data flow machine [6][7][12]. To our knowledge, no other researcher has explored the dynamic compilation of software to hardware using a queue machine model.

Conversion of binaries from one form to another is a topic of intense recent research. The Intel Pentium Pro and all subsequent Intel 32-bit processors have converted the external binary to a sequence of finer-grained RISC-like instructions for execution on the internal core [11]. Transmeta also performs conversion of Intel 32-bit code to an internal representation [10]. In a broader sense, many microarchitectural innovations are essentially embedding some compiler optimization in hardware. For example, a trace cache [13] reorders instructions and stores the decoded version of that code sequence for efficient fetch and decode in subsequent iterations. Another example is dynamic vectorization [14], which



(a) Interconnect limited fabric (b) Time-space emulation of fabric

Fig. 12. Width virtualization options for interconnect-limited fabric: (a) block diagram (b) time-space emulation.

looks for looping behavior in instruction traces, and reorganizes the execution in order to improve performance. The idea of dynamic hardware compilation is a radical extrapolation of such techniques for applications kernels that exhibit very large levels of parallelism.

VII. Conclusions

The logistical problems of development, delivery, maintenance and support of reconfigurable computing applications seriously hinders their entry into the general-purpose market. This paper proposes to solve these problems with a unified representation of complete applications and dynamic on-chip compilation of those applications into hardware. This is very challenging to do using a typical RISC instruction set architecture, because use of a register file tends to obfuscate the structure of the data flow graph. Stack machines have no register file, but they move independent instructions away from each other in the instruction sequence and can only express data flow graphs that are trees. Queue machines are more complete, as any data flow graph can be converted to a queue machine representation. This conversion does increase the number of instructions in the application representation, but these instructions may be significantly smaller than a RISC instructions with explicit operands. Queue machines also have properties that allow the structure of the data flow graph to be easily extracted, in a single sequential execution of the code. A queue machine application can be easily mapped to an appropriate hardware fabric. In case the fabric generated by the queue machine code is too large for the physical fabric present on the chip, we presented a number of techniques to virtualize the hardware.

Acknowledgments

This work was partially funded by DARPA ITO/TTO under contract DABT63-96-C-0083. Herman Schmit is partially supported by an NSF CAREER grant. Benjamin Levine is supported by an IBM/SRC graduate fellowship.

References

[1] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler" in *Computer*, pp. 70-77, April, 2000.

[2] D. Cronquist, P. Franklin, S. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines, (FCCM)*, pp. 116-127, 1998.

[3] L. S. Heath, S. V. Pemmaraju, A. N. Trenk, "Stack and Queue Layouts of Directed Acyclic Graphs: Part I," SIAM J. Comput. Vol 23, No. 4, pp. 1510-1539.

[4] PACT Corporation, "Parallel and Sequential XPP Processing Models," http://www.pactcorp.com/.

[5] M. Jünger, S. Leipert, and P. Mutzel, "Level Planarity Testing in Linear Time," Technical Report, Zentrum für Angewandte Informatik Köln, Lehrstuhl Jünger, URL: http://www.zaik.uni-koeln.de/~paper, 1999.

[6] B. R. Preiss and V. C. Hamacher. "Data Flow on a Queue Machine," in *Proc. 12th Int. Symp. on Computer Architecture*, pages 342-351, Boston, MA, August 1985.

[7] S. Okamoto, "Design of a Superscalar Processor Based on Queue Machine Computation Model", IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 151-154, 1999.

[8] W. A. Wulf, "Evaluation of the WM Architecture," *Proc. 19th Int. Symp. on Computer Architecture*, pages 382-390, 1992.

[9] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Second Edition, Addison-Wesley, 1999.

[10] T.R. Halfhill, "Transmeta Breaks x86 Low-Power Barrier," in Microprocessor Report, Vol. 14, Archive 2, pp. 1,9-18, Feb. 2000.

[11] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, Vol. 9, Issue 2, Feb 1995.

[12] M. M. Fernandes, J. Llosa, and N. Topham, *Using Queues for Register File Organization in VLIW Architectures,* Technical Report ECS-CSG 29-97, Dept of Computer Science, University of Edinburgh, 1997.

[13] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in *Proc. of the 29th Annual Intl Symp. on Microarchitecture,* November 1996.

[14] S. Vajapeyam, P. J. Joseph, T. Mitra, "Dynamic Vectorization: A Mechanism for Exploiting Far-Flung ILP in Ordinary Programs," in *International Symposium on Computer Architecture,* pp. 16-27, 1999.