Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable

Benjamin A. Levine and Herman H. Schmit
Department of Electrical and Computer Engineering
Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213
blevine@cmu.edu, herman@ece.cmu.edu

Abstract

*Hybrid architectures, which are composed of a conventional processor closely coupled with reconfigurable logic, seem to combine the advantages of both types of hardware. They present some practical difficulties, however. The interface between the processor and the reconfigurable logic is crucial to performance and is often difficult to implement well. Partitioning the application between the processor and logic is a difficult task, typically complicated by entirely different programming models, heterogeneous interfaces to external resources, and incompatible representations of applications. A separate executable must be produced and maintained for each type of hardware. A novel architecture called HASTE (**H**ybrid **A**rchitecture with a **S**ingle **T**ransformable **E**xecutable) solves many of these difficulties. HASTE allows a single executable to represent an entire application, including portions that run on a reconfigurable fabric and portions that run on a sequential processor. This executable can execute in its entirety on the processor, but for best performance portions of the application are mapped onto the fabric at run-time. The application representation is key to making this concept viable, and several different ones were examined. Some used a relatively conventional register instruction set architecture (ISA) while others used a new queue-based ISA. An ISA using a modified form of register addressing has been shown to have the best overall characteristics and should allow for the practical implementation of HASTE.*

1. Introduction

Reconfigurable computing hardware has been shown to be a flexible and high performance way to implement computationally intensive applications, especially applications which operate on streaming data and which have lots of exploitable parallelism. Examples of reconfigurable computing architectures include RaPiD [3], PipeRench [4], Imagine [5], and MATRIX [2]. The majority of practical reconfigurable computing architectures developed to date are not computationally complete, so they are either used as coprocessors for a general purpose CPU, or include a conventional processor as an integral part of the architecture. An architecture that combines a CPU and reconfigurable hardware is often referred to as a *hybrid architecture*. The reconfigurable portion of a hybrid architecture will be referred to as the *reconfigurable unit* (RU). Examples of hybrid architectures with a closely coupled CPU and RU include OneChip [6], GARP [7], and MorphoSys[1]. Hybrid architectures require that an application be partitioned into two types of code, corresponding to the two types of hardware. Highly parallel portions of the code that operate on large amounts of data are often referred to as kernels, and typically consist of an inner loop, or small set of nested loops, in the original code. These kernels are converted into a suitable configuration for the RU and executed on it, while less parallel and/or more control-intensive portions of the code are implemented on the CPU.

While they seem to provide the high performance of the reconfigurable hardware, combined with the computational completeness of the CPU, hybrid architectures do present some drawbacks. They require the generation and maintenance of two separate executables for each application: an executable for the CPU and an executable, or configuration, for the reconfigurable portion of the architecture. The task of partitioning the application is often quite difficult. Typically the reconfigurable fabric is programmed using either a specialized language or a hardware description language, while the CPU is programmed using a conventional high-level language. The need for two diverse programming flows and computational models greatly complicates the partitioning and development process. Furthermore, a substantial amount of development time must be spent to create interface hardware and software to allow the CPU and RU to communicate and remain coordinated. The interface between them must be at least somewhat specific to the particular CPU and RU being used, which limits the portability of the executables.

One view of the ideal hybrid architecture would be one that has a complete and unified computational model, such that an entire application could be programmed as a single entity, with no need for interface programming or design. It would allow for both conventional sequential execution on the processor and highly parallel spatial execution on the reconfigurable fabric, without requiring the designer to attend to the details of coordinating these different modes of execution. In order to maximize performance, there would be little or no overhead required to switch between modes of execution, so that all of an application that could benefit from execution on the RU could do so efficiently. Such an architecture would have programming semantics and tool flow closer to conventional software design than to hardware design, expanding the pool of potential users.

The goal of our research is to allow a single executable to represent both portions of the application, with the configuration for the RU generated at run time. We call this idea HASTE, Hybrid Architecture with a Single Transformable Executable.

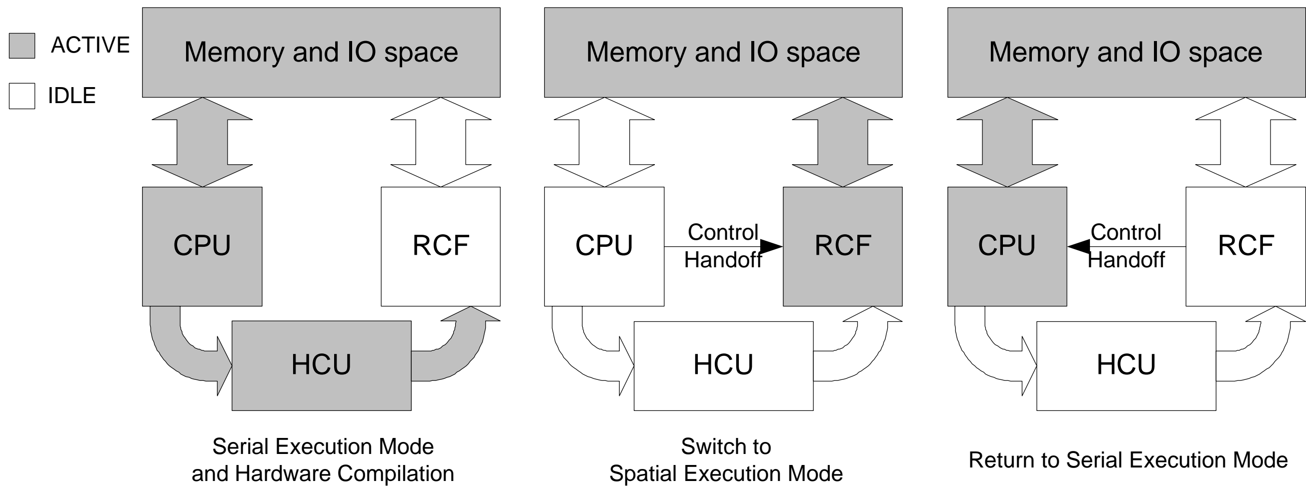

Figure 1: HASTE in operation. The HCU creates a configuration for the RCF at run-time.

HASTE architectures meet most or all of the characteristics of the ideal architecture described previously. HASTE is targeted towards embedded systems and systems-on-a-chip that must efficiently implement highly parallel applications that operate primarily on streaming data. This category includes many applications of interest, especially those in the signal, image and video processing domains, as well as encryption, channel coding, and other communication applications.

Figure 1 illustrates the HASTE concept. HASTE is made up of three main components: a general purpose processor (CPU), a reconfigurable computational fabric (RCF), and a hardware compilation unit (HCU). The RCF is an array of reconfigurable logic, reconfigurable interconnect, and storage, arranged to form a pipelined datapath; it will be discussed in Section 2. The CPU is a Von Neumann processor which operates sequentially on an instruction stream and implements control flow as well as data processing. It may be very similar to a conventional RISC processor, or a more specialized processor architecture may be used. Both the CPU and RCF share a common interface to memory and IO. Some aspects of the different CPUs considered in this research will be discussed in section 3. The HCU converts a sequential instruction stream for the CPU into a configuration for the RCF and is detailed in Section 4.

For simplicity, we will assume here that each kernel is comprised of a single loop body which iterates a fixed number of times and has a single entry point and a single exit point, although we can accommodate more complex kernels in HASTE. The beginning and end of each kernel is marked in the executable. The CPU will execute the code outside of the kernel or kernels, then execute one iteration of the loop body comprising a kernel while the HCU creates a configuration for the RCF. On subsequent loop iterations, the RCF uses this configuration to execute the loop body as a pipelined datapath. Once the kernel is complete, control passes back to the CPU. In a traditional hybrid architecture, this kernel code could be compiled into an executable to run on the processor, or synthesized to run on the reconfigurable logic, but it would not be possible for it to be implemented as a single executable that could run on both.

To make the HASTE concept work we must choose an application representation which has a valid sequential semantic and yet also contains the information necessary for the HCU to create a spatial configuration. In this paper we will investigate and compare several different application representations and determine their efficiency. The idea of an application representation in this context encapsulates several things, including the form and encoding of the sequential instruction stream, the associated CPU architecture needed to process the stream, the way in which the kernels are configured in the RCF, and the HCU algorithms used to convert the instruction stream into the RCF configuration. Although somewhat different than the traditional usage, we will refer to all parts of the application representation as the instruction set architecture (ISA).

To explore the efficiency of different ISAs, we must first describe what is meant by efficient. Efficient has several meaning in this context. First, we need a compact sequential code sequence. If we require many instructions to represent a kernel, then the kernel will run slowly on the sequential processor. If our representation requires us to make changes to the way we express the non-kernel portions of the application as well as the kernels themselves, then compact code is even more important. Secondly, we would like an ISA that allows us to create configurations that make an efficient use of the fabric. If we must have many extra hardware resources to allow the HCU to create a configuration, or if the configurations that it produces do not use the available resources efficiently, then HASTE will be of limited usefulness. The third kind of efficiency relates to how easy it is for the HCU to transform the kernel sequential code into the fabric configuration. We would like this to be a simple process so that we don't require a great deal of hardware for the HCU, and we would also like this process to complete quickly, ideally in the time it takes a single loop iteration to execute on the sequential processor. We would like the total size of the executable to be small. If we require each instruction to be large or if we require many instructions, then the total executable size will be large, which increases requirements for storage and instruction bandwidth.

This paper is structured as follows: Section 2, 3, and 4 discuss the fabric, the ISAs and CPUs, and the HCU, respectively. Section 5 shows our experimental results and Sections 6 and 7 discuss some conclusions and possible future work.

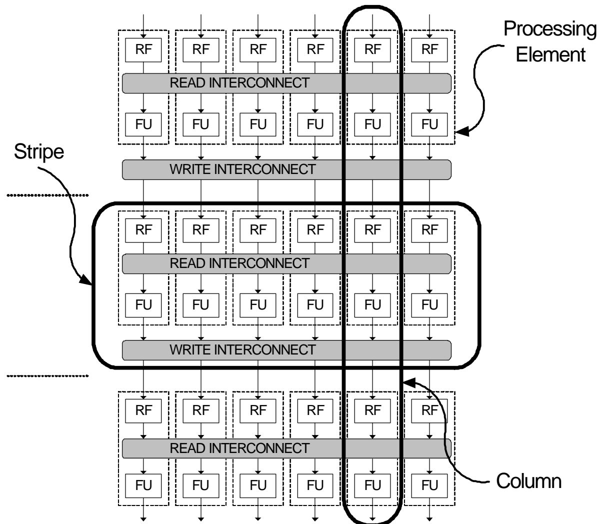

Figure 2: Three stripes of a six column wide fabric.

2. Hardware Structures and Models

2.1. Reconfigurable Fabric

The efficiency of different ISAs depends in some cases on the specific characteristics of the RCF used. Some ISAs require that the RCF have certain characteristics in order for it to be used at all. In order to enable exploration of as large a design space as possible, a generic and parameterizable model of the reconfigurable fabric will be used. This model assumes that the fabric consists of coarse grained functional units, register files, and programmable interconnect, arranged to form a pipelined datapath, as shown in Figure 2. Each pipeline stage, or *stripe*, is composed of several processing elements and two programmable interconnect networks. Each processing element contains a programmable functional unit (FU), and a register file (RF). One interconnect network allows the FUs in one stripe to read values from the RFs in the same stripe. The other interconnect network allows the FUs in one stripe to write values into the RFs in the next stripe. Our fabric model can represent a wide range of fabrics using only a few base parameters. A somewhat simplified version of the model was used for this study; the relevant parameters are listed in Table 1

Customized datapaths for each specific application kernel are implemented by programming the functional units, register files, .and interconnect. A single configuration word specifies the necessary programming for each processing element, including the portion of the interconnect networks associated with that processing element. A set of one configuration word for each processing element in a stripe provides all of the necessary information to configure that stripe. Each configuration word is composed of several fields, with each field corresponding to a particular hardware structure. The hardware model specifies the specific fields that make up the configuration word for each processing element. The number and types of fields required will vary between different ISAs. A detailed diagram of the processing element model is shown in Figure 3.

TABLE 1 : Hardware Model Parameters

Global Parameters	
Fabric width (number of columns)	W
Functional Unit Parameters	
Operand bitwidth	B
Number of operands	P
Register File Parameters	
Number of words	F
Number of read ports	P_R
Number of write ports	P_W
Interconnect Parameters	
Read span	S_R
Write span	S_W

2.2. Functional Unit

For the purposes of this paper, we will use a single functional unit type, with the number of operands that it accepts, P, equal to 2, and the bitwidth of the those operands fixed at some arbitrary value, B. In addition, we will specify that the bitwidth of all of the other components of the datapath as equal to B as well. All functional units are identical and all can execute any of the operations that we specify as being legal for the RCF. The details of the configuration of the functional unit are not relevant to our investigation of the different ISAs and so we will abstract the functional unit configuration to consist of a single field, `FUNC`. The value of `FUNC` can be determined solely from the sequential instruction itself by a simple table lookup in the HCU.

2.3. Register File

The register file in each PE is parameterized by three values: F, the number of registers; P_W, the number of write ports; and P_R, the number of read ports. Each register file entry contains a single B-bit wide word. For this study, all register files with F larger than one are pass register files. A pass register file is one in which all register values which are not overwritten in the current stripe are passed to the corresponding registers in the register file in the same column in the next stripe, where column refers to all of the processing elements at the same horizontal location in each stripe in the fabric. The simplest register file has parameters $\{F = 1, P_W = 1, P_R = 1\}$, and corresponds to a single register. If $F > 1$, then we need to specify the register for each read and write, so we will need P_R read address fields, each labeled `RA`x, where x is the corresponding read port number, and P_W write port address fields, each similarly labeled `WA`x. The write address fields also incorporate a write enable bit, which is not shown separately for clarity. This means that we will have a field `WA1` for the case when $F = 1$, and this field is solely a write enable bit.

2.4. Interconnect

The interconnect model is shown in Figure 4. There are two sets of interconnect wires, as shown in sub-figure (a). One set

connects the register files in a stripe with the functional units in the same stripe. These wires perform reads of values in the register files. Each register file read port connects to a read bus and can be read in one or more PEs. Each read bus can be read by S_R PEs, including the source PE, where S_R is the read span. Each PE can read from P_R local read buses, which are those originating in the current PE, and $(S_R \bullet P_R) - P_R$ remote read buses, which originate in other PEs in the same stripe. S_R is always odd and the PEs that are connected are always symmetrical, so $S_R = 3$ means the current PE and the PEs to the immediate right and left can read the value; $S_R = 5$ means that two PEs on either side and the current PE can all read the value, and so on. In Figure 4, we see examples of read spans equal to 1,3, and 5 in (b), (c), and (d), respectively. The functional unit operands are selected from the read buses using operand select muxes. Since we are assuming two operands per FU, there will be two operand select muxes, labeled OSM1 and OSM2 in Figure 3. Each operand select mux can chose from $(S_R \bullet P_R)$ read buses. A configuration field for each operand select mux will needed, `OS1` and `OS2`.

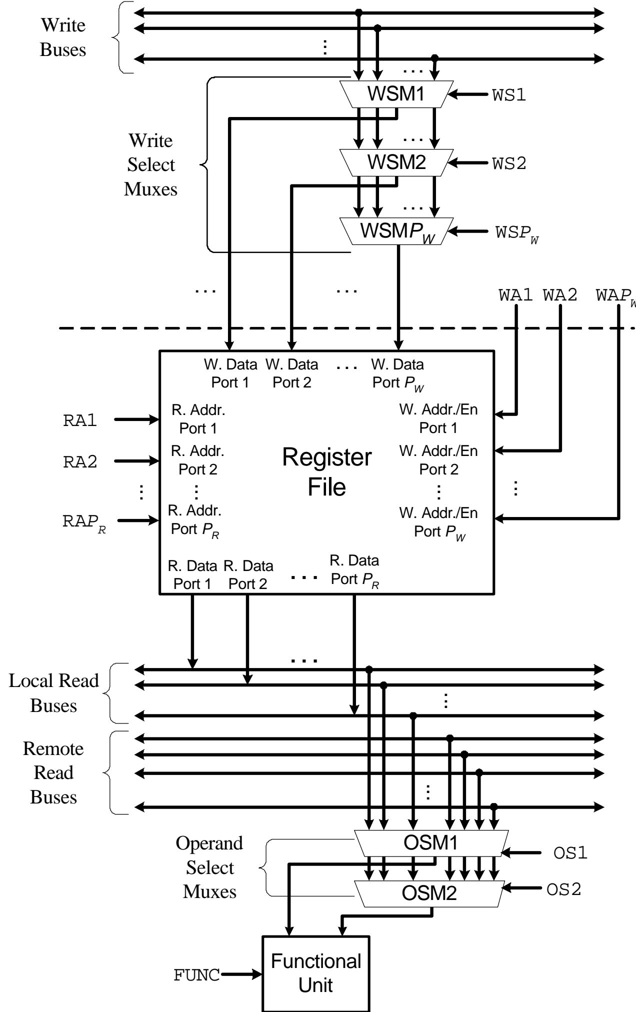

Figure 3: Processing element model.

The second set of interconnect wires connects the functional units in a stripe with the register files in the next stripe. These wires perform writes of results produced by the functional units into register files. The architecture has an associated write span, S_W, similar to the read span, which determines how many different register files each functional unit can write to. In Figure 4, we see examples of write spans equal to 1,3, and 5 in (e), (f), and (g), respectively. If the write span is larger than one, a write select mux is required for each write port. These are shown as WSM1...WSMP_W an Figure 3. Each write select mux has an associated configuration field, `WS1` through `WSx`.

3. Application Representations for HASTE

3.1. Original Kernel Representation

We will only explore how to represent the applications kernels that we wish to run on the RCF. The rest of the code will run on the processor only and is not of central concern here; only the code that must run on both processors is of interest for this paper. The kernels that we are mapping to the RCF are initially represented as a data-flow graph (DFG), a directed-acyclic graph where each node represents an instruction and each edge represents an operand. Inputs and outputs to the DFG are all memory loads and stores. Each node in the DFG is assigned a

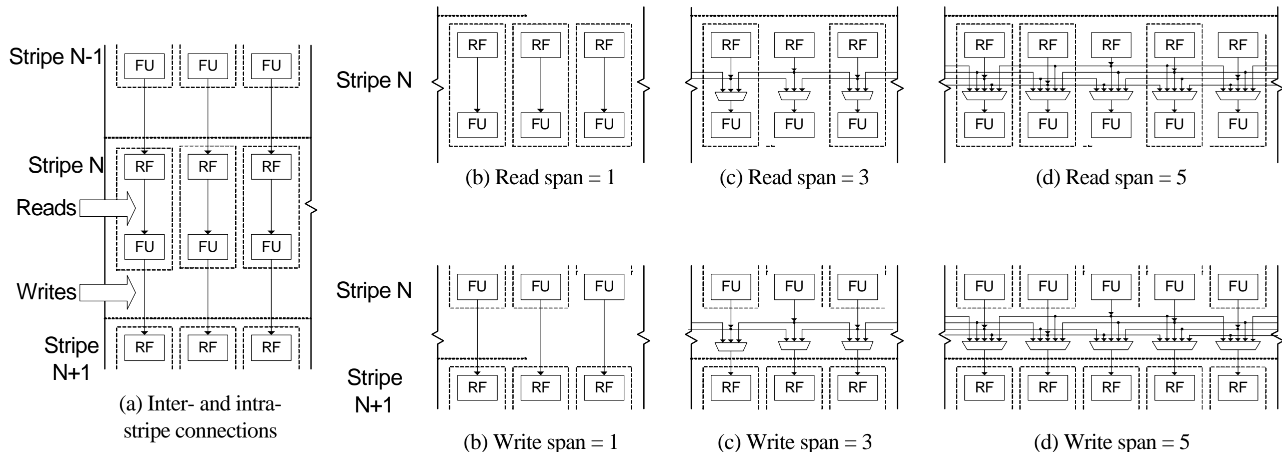

Figure 4: Interconnect model showing different read and write spans.

level, such that all input edges to the node originate in nodes in a higher level, and all output edges terminate in a node at a lower level. No control is explicitly represented in the DFG, although control can be incorporated in a DFG using techniques such as if-conversion [8]. DFGs are a standard way to represent portions of an application and are generated by most compilers.

The kernel DFGs may need to be modified to use with the different HASTE ISAs and fabrics. We need to ensure that the kernels will run and give the same results on both processors. So the executable for each kernel must have a valid sequential semantic and also must contain enough information to allow for the construction of a valid spatial semantic. A spatial semantic, in this context, means information about the placement of operations and the connections between operators to produce the same results as the sequential semantic; in effect, placement and routing of the fabric. This means that the executable must be somewhat specific to the particular fabric it will be run on.

The DFG may first need to be modified so as to create a valid sequential instruction sequence. It then may need to be modified so as to be implementable on fabrics with a specific set or range of parameters. This may require the addition of instructions that move data in the fabric and/or processor. No instructions which do actual computation will be added, changed, or removed, so the underlying computation being performed will not change. The position of operands in the DFG may also be changed, changing the instruction ordering in the corresponding instructions stream.

3.2. Queue ISA

The easiest kind of ISA for the HCU to convert is a queue ISA, which is an ISA targeting a queue machine. More details of queue machine operation can be found in [11], but the basics will be repeated here. A queue machine is analogous to the more familiar stack machine, such as the Java Virtual Machine [10], except that it uses an operand queue rather than an operand stack. Given some arbitrary stream of instructions, it may be possible to reorder them so that they can run on a queue processor, but this is not true of most instruction sequences. Fortunately there is a correspondence between certain properties of a DFG and correct ordering of instructions for a queue machine. Any data flow graph that is both level and planar will produce an instruction sequence that will run correctly on a queue machine [12]. A level graph is one that can be drawn with the nodes in rows such that every edge goes from a node in one row to a node in the next row and no edge skips a row. An example of a non-level graph and a level graph are shown in Figure 5(a) and Figure 5(b), respectively. A planar graph is one that can be drawn without any edges crossing. An example of a graph that is both level and planar is shown in Figure 5(c).

By adding new nodes to the data flow graph, we can produce a level-planar graph from any arbitrary DFG, so queue machines can be used to implement any instruction sequence, if certain queue manipulation instructions are allowed. These instructions are `SWAP`, which replaces edges that cross, and `PASS`, which allows instructions to cross levels. In the queue CPU, a `PASS` instruction removes an operation from the head of the queue and places it on the tail of the queue, and a `SWAP` instruction removes two instructions from the head of the queue, reverses their order and places them on the tail of the queue. The process of making a graph level and planar is shown in Figure 5. The level planar property of the DFGs limits the range of fabrics that can be used.

Since the operands for the queue ISA are determined solely by their position in the operand queue, the instruction format is very compact. Only a single byte is needed to specify the instruction. Constants are supplied in succeeding bytes if needed. The instruction encoding for all ISAs are shown in Figure 6 and the queue instruction format is shown in Figure 6(a). It consists of a single 8 bit field labeled `OP`.

3.3. Register File ISA

A traditional register ISA seems to be a logical choice. This

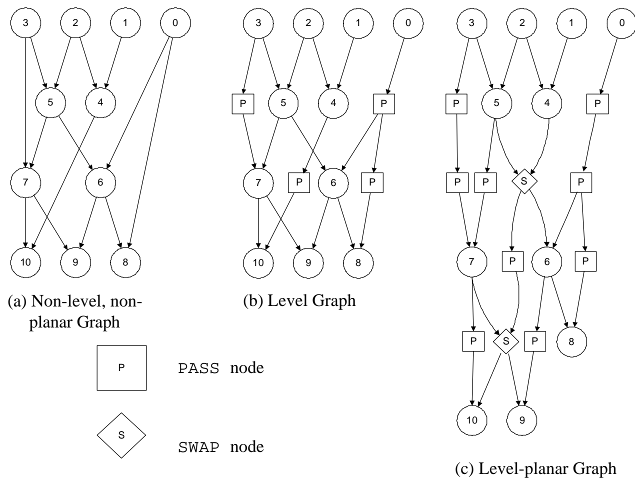

Figure 5: Queue ISA DFG transformation.

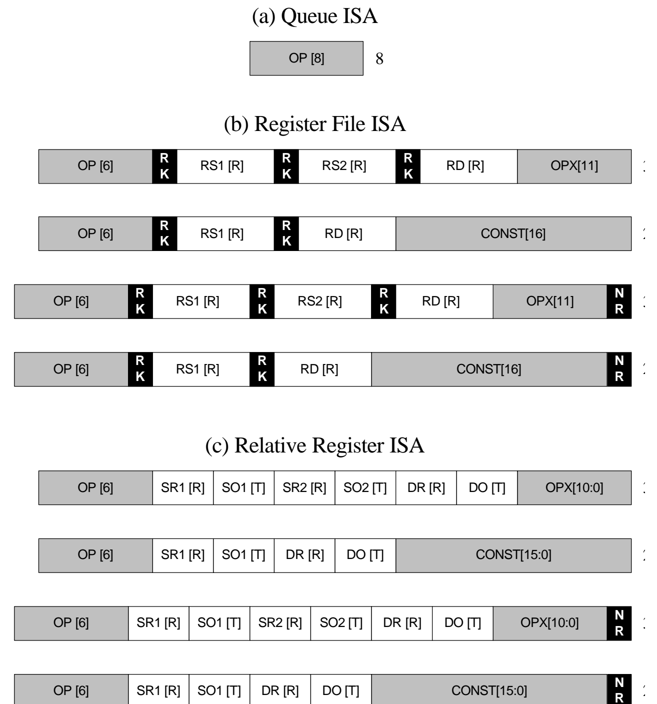

Figure 6: Instruction formats for all ISAs.

makes the construction of the CPU easy, since a conventional or nearly conventional processor architecture can be used. However, extracting parallelism and constructing a spatial configuration from code produced by a standard compiler is an extremely difficult, if not impossible, task. It is necessary to produce executables in this register ISA that allow for the creation of a configuration that can be implemented on the fabric that they will be run on. Thus the compiler must know the base parameters of the fabric and the algorithms used by the HCU, and modify the DFG accordingly. It does this by producing a spatial configuration as part of the compilation pass. Then register assignments are made and the sequential code can be written out. One restriction on this ISA type is that the number of architected registers in the CPU limits the number of live variables that we can have in fabric. In order to support large kernels, a large number of registers are needed. This in turn requires more bits to represent the source and destination registers and may require a large instruction size, thus increasing code size.

Another problem is that the task of the HCU is very complicated. Since operands are identified only by register numbers, the HCU must keep track of the location of these values and route the fabric accordingly. There is no correspondence between the architected register file number used in the CPU and the register location in the fabric. To completely specify the location of a register in the fabric, we must know the column of the register file and the number of the register file entry. Note that there will typically be more registers in a single row of the fabric than there are registers in the CPU. Values stored in a single register in the CPU may be stored multiple places in the fabric. This greatly increases the amount of hardware required and may require longer than one loop iteration to convert the executable.

We must make a few modifications to a typical register ISA to make it work with HASTE. First of all, we need to give information to the HCU as to the location of each instruction. For our fabric, this means the row and column location. If we assume that instructions are issued from left to right and from the top row to the bottom, then determining the location is quite easy if each row is filled with instructions. If each row is not full, then we need some way to indicate when to start placing instructions on the next row. We can either include a bit in each instruction that is set to indicate the end of a row or add an instruction that indicates the end of a row. This would be ignored (treated as a No-op) by the sequential processor. We will compare both options and use whichever one impacts efficiency the least. Another modification is necessary if we have a fabric with limited interconnect. In the fabric, if an operation needs operands that are in columns too far apart to be read in any single PE, than one or both operands must be moved. A move instruction can be implemented to read operands from one column and write to another. The HCU interprets a move instruction as a routing directive that reads a value from a register in one column and writes it to a register in a column some number of columns to the left or right. The move instruction would be treated as a No-op by the sequential processor, since the value being moved would still be referred to by the same register number and be in the same location in the CPU's register file. Both moves and new row indicators only need to be added to portions of the executable to be run on the RCF.

Our register file ISA instruction encoding is based on the MIPS32 ISA [9]. Our instruction encoding is shown in Figure 6(b). There are two relevant instruction formats, one with three registers specified and one with two registers and a constant. We modify only the size of the register fields, requiring them to be some width R, where R is the number of bits needed to address all registers in our register file. Each register field has an additional *register kill* bit, labeled RK. The register kill bit indicates to the HCU that the current value of a specific register will not be used again and it can stop tracking the location of that value in the fabric. We may also want to add a single bit, labeled NR, to indicate a new row. If the new row bit is not used, new rows are indicated by a new row instruction, which is ignored by the CPU.

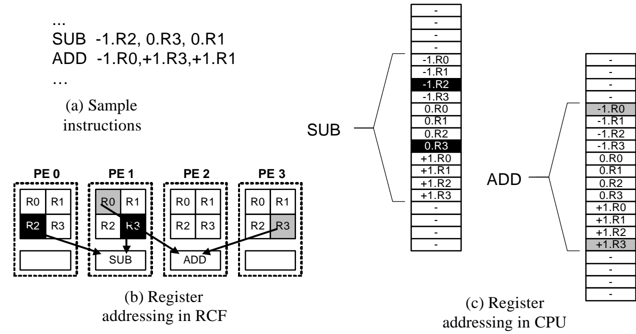

Figure 7: Relative register addressing.

3.4. Relative Register ISA

Determining the routing for the PEs is difficult for a register file ISA because the only available information about operands is the CPU register file number. Another problem is that we are limited in the size of the application by the size of the architected register file. Rather than require the HCU to determine which register file and register entry to read and write operands to and from, it would be nice if this information was present in the executable. An efficient way to do this is shown in Figure 7. In Figure 7(a) are two example instructions. Each register is addressed by a register number and an offset. The meaning of this in the RCF is straightforward. The register number refers to a register in the register file of each PE in the fabric. There are F of these registers, using our fabric model parameters. The offset refers to the column location of the register file in the fabric, relative to the current column. So in our example, the SUB instruction's first operand is located in register 2 at offset -1, meaning the column one to the left of where the instruction is being implemented, and the second operand is in register 3 at offset 0, meaning the current PE. The total number of columns we can address should be the same as the read span. Figure 7(b) shows the locations of operands for the example instructions.

The meaning of type of register addressing in the CPU is less clear. One way to implement this ISA would require the CPU to use a register file with a sliding window. The row location of each instruction would need to be tracked in the CPU and the register file window indexed accordingly. The size of the window is dependent on the read and write spans and the size of each register file in the fabric. Since the movement of the register file window is predictable, register values that aren't needed can be cached,

allowing for a very large register file and thus allowing for very wide fabrics. Some kernels will require modification of their DFG to allow for operands to be read from locations outside the register window; this can be done by using move instructions, as discussed for the register file ISA. These move instructions will be needed for all portions of the application, not just the kernels. We will also need new row indicators just as for the register file ISA and these will also be needed for the whole application, since we need to properly index the register window. The register addressing in the CPU is shown in Figure 7(c). Each register is indicated by a register number field of size R, with R large enough to represent F registers, and an offset field of size T, with T large enough to represent the larger of S_W and S_R. Except for the register file, we can use a mostly standard RISC CPU for this ISA.

4. HCU Operation

The hardware compilation unit accepts a stream of sequential instructions and produces a set of configuration words for the reconfigurable fabric. It must do this stripe-by-stripe, in the order that the instructions are presented, and must do so as quickly as possible, ideally within a single iteration of the loop. Different ISAs will require different HCU algorithms. Since we would like the architecture to be scalable to very large fabrics, HCU algorithms which require access to an entire row at once are not feasible. The HCU algorithm must also be feasible to implement in a reasonable amount of hardware. Describing the HCU hardware in detail would be beyond the scope of this paper. However, the different algorithms can be described such that comparisons between their complexity can be made and the relative difficulty of implementing them in hardware can be estimated. We will use the HCU model shown in Figure 8 for all HCU algorithms. This model consists of two main portions, an *issue unit*, which receives the sequential instruction stream from the CPU, and does some amount of processing on the instructions, and a chain of *configuration stations*, one for each column of the fabric, where the final configuration word for each PE is determined.

The HCU reads an instruction each clock cycle into the issue unit. A lookup table is maintained in the HCU to convert the operation field or fields of the instruction into a suitable value for the `FUNC` field of the configuration word. The lookup table may also contain other information about the instruction needed by the HCU algorithm, such as the number of operands. The issue unit passes the decoded instruction, and possibly other data, into the first configuration station. Each clock cycle instructions advance from one configuration station to the next. When a row is complete, the configuration for the stripe is passed into the fabric. For some ISAs, additional processing is required in the configuration stations to complete the generation of the configuration. Configuration stations may be connected together, if needed to generate the configuration word, but only a limited number of configuration stations may be so connected, so that we can maintain some limitation on the amount of interconnect required. Only the C configuration stations nearest the issue unit can be connected together in an arbitrary manner; all others must simply pass a predetermined set of data in one direction, to the adjacent configuration station which is located farther from the issue unit. C is equal to the larger of the read span and the write span for register file ISA and relative register ISA, and is equal to zero for the queue ISA. The simplest HCU algorithm is for the queue ISA and it will be discussed next.

4.1. HCU Algorithm for Queue ISA

A queue ISA requires only a simple fabric architecture. It needs only a single register in the register file $\{F = 1, P_W = 1, P_R = 1\}$, writes all results to the same column, and reads from some range of adjacent columns $\{S_W = 1, S_R >= 3\}$. This architecture doesn't need register file addresses or write select muxes, so it has only four fields, `OP`, `OS1`, `OS2`, and `WA1`.

Listing 1 shows pseudocode for the queue ISA HCU algorithm. In the pseudocode listings in this paper, `courier` font is used to indicate fields and signals present in the hardware, normal text indicates variables in the pseudocode with no direct equivalent in hardware or functions implemented as hardware components, and *italic* text indicates pseudocode. Portions of the code represent the functionality of the issue unit and portions the functionality of the configuration stations. Since we are representing hardware, the code represents concurrent processes, as they would be expressed in a hardware description language. However, we are using higher level constructs to describe the algorithms here so that the length of the listings is much less than would be needed for an HDL.

The issue unit needs to maintain two queues of column

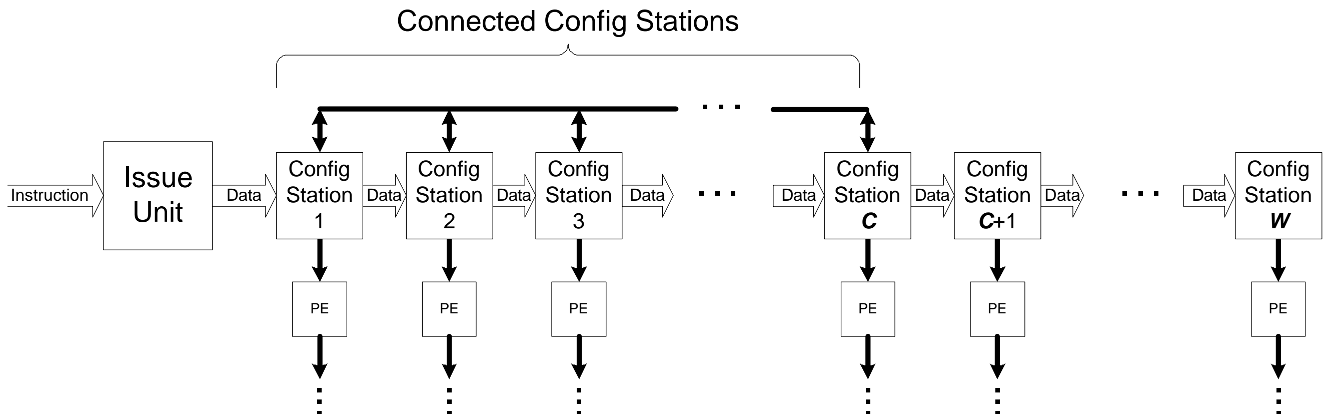

Figure 8: HCU hardware model.

numbers to track the locations of operands. One, called `this`, tracks the column location of operands which will be used in the current row and the other, `next`, maintains the column location of operands which were produced in the current row and will be used in the next row. The queue ISA instruction has just one 8-bit field, `OP`. The lookup table in the HCU returns the number of inputs and outputs for each `OP`, as well as the value for the `FUNC` field of the config word. Each output of the current instruction is represented by an entry in the `next` queue. Since each instruction starts at position 0, at the left of the fabric, a 0 is inserted into the queue. As each instruction is shifted onto the config stations, the location of operands for the next row in the `next` queue are incremented, to reflect that the position of each instruction is also increasing by one column. If the current instruction does have an output, the `WA1` field is set, enabling a write to the register. The `WA1` field from the previous row is maintained in each config station to enable use of the proper value in the current configuration word. This is necessary because register write values for the current PE are actually associated with the previous PE in the same column (see Figure 3).

The location of each input for the current instruction is retrieved from the `this` queue. These locations are shifted onto the configuration stations as column numbers `COL1` and `COL2`. The config station logic decodes these column numbers into mux settings for the `OS1` and `OS2` fields using the known column location of the config station. If there are no more operands in the `this` queue, the HCU starts a new row, and all configuration words in the config stations are shifted down into the fabric. The contents of the `next` queue are copied into the `this` queue, since they will be used as inputs in the new row.

Listing 1: Queue ISA

```
Issue Unit:
Inputs: OP
Outputs: FUNC_C, COL1, COL2, WA1_C
for each instruction:
    [FUNC_C, numInputs, numOutputs] <= lookup(OP);
    for i = 1 to numOutputs {enqueue 0 in next}
    for all elements in next {increment element};
    if (numInputs != 0 and this is empty) {
        this = next;
        clear next;
        start new row;
    }
    if (numInputs > 0) COL1 <= dequeue value from this;
    if (numInputs = 2) COL2 <= dequeue value from this;
    if (numOutputs = 1) WA1_C <= 1 else WA1_C <= 0;

Configuration Station:
Inputs: FUNC_C, COL1, COL2, WA1
Outputs: FUNC, OS1, OS2, WA1
    OS1 <= decodeMux(COL1, station position);
    OS2 <= decodeMux(COL2, station position);
    FUNC <= FUNC_C;
    WA1 <= prevWA1;

    on new row: prevWA1 <= WA1_C;
```

4.2. Register File and Relative Register HCU Algorithms

The HCU algorithms for the register file ISA is shown in Listing 2. Space does not permit detailing its operation. The HCU code for the relative register ISA is very similar to Listing 2 and will not be shown here. The relative register ISA algorithm is much simpler to implement than the register file ISA, because register locations can be determined directly from the instruction, without needing to maintain a table of equivalent register locations. Both register ISA algorithms are more complex than the queue ISA algorithms and much of the processing for both occurs in the configuration stations, and so must be duplicated across the width of the fabric. This means that these algorithms require many times more hardware than is needed for the queue ISA, although the hardware is fairly simple for the relative register ISA. Both register ISA algorithms have another drawback; most of the configuration is generated only when a new row is issued, and several clock cycles are required for the generation of the configuration. Thus it will take longer than one loop iteration on the CPU for the HCU to complete the RCF configuration.

Listing 2: Register file ISA

```
Issue Unit:
Inputs: OP, OPX, RS1, RS2, RD, RK1, RK2
Outputs: FUNC, RS1_C, RS2_C, KR1, KR2, RD_C
for each instruction:
    if (OP = NEWROW or newrow bit set), start new row;
    else:
    [FUNC, numInputs, numOutputs] <=
        lookup(OP, OPX, CONST);
    if (numOutputs > 0) RD_C <= RD;
    if (numInputs > 0), RS1_C <= RS1, KR1 <= RK1;
    if (numInputs > 1), RS2_C <= RS2,KR2 <= RK2;

Configuration Station:
Inputs: FUNC, RS1_C, RS2_C, KR1, KR2, RD_C
Outputs: FUNC, OS1, OS2, WA1, RA1...RAx
on new row:
    RS1Locs <= queryLoc(RS1_C,K1);
    RS2Locs <= queryLoc(RS2_C,RK2);
    RS1Loc <= nearest(RS1Locs);
    RS2Loc <= nearest(RS2Locs);
    bus1 = requestRead(RS1Loc, RS1_C,KR1);
    bus2 = requestRead(RS2Loc, RS2_C,KR1);
    OS1 = decodeMux(bus1, RS1Loc);
    OS2 = decodeMux(bus2, RS2Loc);

    for each read request(regNum, regKill) {
        reg <= location found in register table;
        enqueue [reg, regNum, regKill] on readList;
    }
    remove duplicates from readList;
    for each entry in readList;
        portNumber <= next open read port;
        RA[portNumber] <= reg;
        if (regKill = 1), remove regNum from register table;
        return portNumber for read to requestor(s);
    WA1 <= next open entry in register file;
```

5. Experimental Results

There are five metrics that we will use to evaluate the different ISAs. The first is code length, the total number of sequential instructions in the executable. Code length is important since it controls the time required to execute the kernel on the sequential processor and also the time required to convert the application from the sequential form into a configuration for the RCF. The number of instructions for each kernel is initially the same, but as we have seen, the different ISAs may require the addition of new instructions, not in the original kernel, that will increase the number of instructions. These added instructions include the move and new row operations for the register ISAs and the pass and swap operations for the queue ISA. The code length is the number of instructions in the original kernels plus the number of added instructions of all types. The second metric, hardware utilization, measures the percentage of the fabric doing actual computation. The width of the fabric times the number of stripes determines the total number of PEs needed for the application. Only those PEs holding operations that are part of the original kernel are considered to be doing useful work, so the hardware utilization is defined as the number of instructions in the original kernel divided by the number of PEs needed. The third metric is the code size, which refers to the number of bytes needed to represent the application. This depends on two factors; the size of each instruction and the number of instructions (code length). Since we are using the same basic instructions, the instruction size is mostly dependent on the bits needed to encode operand sources and result destinations, as can be seen in Figure 6. Both register ISAs may also use new row bits if these are found to be create more compact code than new row instructions. The fourth metric is the HCU algorithm complexity. We have only a qualitative measure for this, which was discussed in the previous section. The fifth metric is the number of pipeline stages needed for the application, which effects the latency of the applications.

In order to explore the efficiency of the ISAs, a variety of benchmarks were mapped to reconfigurable fabrics using each ISA. A program called AppMapper was created, which implements the necessary transformations to create valid DFGs for specific fabrics and ISAs. The benchmarks and their characteristics are listed in Table 2. The benchmarks are kernels from various application domains, including signal processing, image processing, and cryptography, and they vary widely in size and structure.

TABLE 2 : Benchmark Applications

Name	Description	Number of levels	Number of instructions
dct	dct transform	9	114
fft	fast Fourier transform	7	88
fir	10 tap FIR filter	25	94
haar	Haar transform	7	62
idea	IDEA encryption	168	311
n-queens	discrete math problem	12	28
over	video “over” operator	10	41
perm	matrix permutation	16	29
quantize	JPEG quantize	27	122
rc6	RC6 encryption	25	78
	Total:	**306**	**967**

For the queue ISA case, the DFG for each kernel was made level planar as described previously. Several different heuristics were used for each step of the process and the best overall results were kept. Since each queue ISA instruction is a single byte, the total code size is equal to the number of instructions. The fabric width for the queue ISA was determined by the maximum graph width, which is minimized during the levelling process. The write span for the queue ISA is always one, and the read span does not effect the efficiency of the encoding, so the only mapping performed was to a fabric with write span = 1 and an arbitrarily wide read interconnect. For the register file and relative register ISA, the efficiency of the encoding is dependent on the interconnect of the fabric being mapped to and the width of the fabric. The register file ISA always uses a write span of one, but the read span was varied from three to complete interconnect. For the relative register ISA equal read and write spans were used, again varied from three to complete interconnect. For both ISAs, the width was increased to the point where increasing the width no longer affected the mapping. Every possible width and read span combination was mapped to. In Table 3, the best results for each ISA are shown, with the best results determined by counting the code length as most important, followed by hardware utilization, code size, and number of stripes as least important. The register file ISA shows a distinct trade-off between short code length and high hardware utilization and so two sets of results are shown. The other ISAs had high correlation between short code length and high hardware utilization so only one set of results is shown.

TABLE 3 : ISA Mapping Results

	Queue ISA				Register File ISA - Best Code Length				Register File ISA - Best HW Util %				Relative Register ISA			
	Code Length	HW Util %	Code Size	# of Stripes	Code Length	HW Util %	Code Size	# of Stripes	Code Length	HW Util %	Code Size	# of Stripes	Code Length	HW Util %	Code Size	# of Stripes
dct	446	16%	446	31	114	100%	570	19	114	100%	570	19	114	100%	570	19
fft	688	9%	688	36	88	100%	440	11	88	100%	440	11	88	100%	440	11
fir	128	40%	128	26	118	78%	590	40	94	73%	470	32	109	73%	436	32
haar	107	26%	107	10	62	97%	310	16	62	97%	310	16	62	97%	310	16
idea	1214	21%	1214	207	495	44%	2475	176	357	37%	1785	168	311	46%	1555	168
nqueens	53	29%	53	14	57	54%	228	13	28	54%	140	13	32	54%	128	13
over	55	47%	55	11	41	93%	205	11	41	93%	205	11	41	93%	164	11
perm	107	13%	107	16	29	66%	145	22	29	66%	145	22	29	48%	145	20
quantize	522	15%	522	58	138	85%	690	36	124	70%	620	29	131	85%	524	36
rc6	240	17%	240	28	94	72%	470	27	84	41%	420	27	80	72%	400	27
Total	**3560**	**18%**	**3560**	**437**	**1236**	**65%**	**6123**	**371**	**1021**	**56%**	**5105**	**348**	**997**	**66%**	**4672**	**353**

6. Conclusions

All three of the ISAs studied here seem feasible for use with HASTE architectures. The queue architecture requires only very simple HCU hardware. However, the high instruction count overhead required to make DFGs level planar makes the queue ISA representation very inefficient in all metrics except for total code size. The instruction count is more than three times as long as for the original kernels and nearly 50% more stripes are required than there were original levels. The hardware utilization is less than 20% for the entire benchmark set and never more than 50% for any single benchmark. The code size is small since each instruction is only a single byte. Using the queue ISA requires the use of a queue processor, requiring the design of an entirely new kind of processor architecture. Using the queue processor requires that all portions of the application be transformed into level planar DFG form, incurring high overhead for the entire application.

The register file ISA allows for a range of mappings that can allow a significant trade-off between hardware utilization and code length for many applications. When optimized for either metric, the results were almost as good as the relative register ISA, the best for both metrics. The code size was typically larger than for the relative register ISA. An instruction encoding almost identical to a MIPS processor can be used for this ISA, with the addition only of new row bits and register kill bits. These could easily be stripped from each instruction to be run on the CPU and so a nearly standard MIPS core could be used. The HCU hardware for the register file ISA is quite complicated, however, and requires extra configuration latency.

The relative register ISA achieves high efficiency on all metrics except code size and has the best hardware utilization and shortest code length of all the ISAs. It also requires relatively simple HCU hardware, although not as simple as the queue ISA. It does require some modifications to a standard MIPS core, but only to the register file. Based on these results, the relative register ISA appears to be the best candidate for the HASTE architecture.

7. Future Work

Now that the characteristics of the ISAs are known, it is possible to proceed with the complete specification of the HASTE ISA and construction of a HASTE simulator. Simulation of entire applications of interest will provide more information on the performance of these architectures and help motivate future research. Design of a complete HASTE micro-architecture using a hardware description language would be desirable to determine the actual hardware requirements and determine specific power and performance measures in particular process technologies. This would allow for more direct comparison between HASTE architectures and other computing hardware. The parameterizable fabric model is well suited for use in an automated layout generator, which would allow the exploration of a wide range of different fabric types as part of any detailed exploration of hardware performance. There are also numerous architectural possibilities to be explored, including the use of smaller (perhaps 4- or 8-bit) functional units and the fabric and the automatic composition of these to form wider units as needed, assuming some indication of required bitwidth in the instruction encoding.

Acknowledgements

This work was made possible by IBM Corporation and the Semiconductor Research Corporation, who support Mr. Levine's research with an IBM/SRC Graduate Fellowship.

References

[1] Singh, H.; Ming-Hau Lee; Guangming Lu; Kurdahi, F.J.; Bagherzadeh, N.; and Chaves Filho, E.M., "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," IEEE Transactions on Computers, Vol. 49, No. 5, May 2000, pp. 465-481.

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

[3] Cronquist, D.C.; Fisher, C.; Figueroa, M.; Franklin, P.; and Ebeling, C, "Architecture design of reconfigurable pipelined datapaths," in *Proceedings of the Conference on Advanced Research in VLSI*, March 1999, pp. 23-40.

[4] Schmit, H.; Whelihan, D.; Tsai, A.; Moe, M.; Levine, B.; and Taylor, R.R., "Piperench: a virtualized programmable datapath in 0.18 micron technology," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2002, pp. 63-66.

[5] Kapasi, U.J.; Dally, W.J.; Rixner, S.; Owens, J.D.; and Khailany, B., "The imagine stream processor," in Proceedings of the IEEE International Conference on Computer Design, 2002, pp. 282- 288.

[6] Wittig, R.D. and Chow, P., "OneChip: an FPGA processor with reconfigurable logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 126-135.

[7] Callahan, T.J.; Hauser, J.R.; and Wawrzynek, J., "The Garp architecture and C compiler," IEEE Computer, Vol. 33, No. 4, Apr 2000, pp. 62-69.

[8] J. Allen, K. Kennedy, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th Annual Symposium on Principles of Programming Languages*, January 1983.

[9] MIPS Technologies, *MIPS32™ Architecture for Programmers Volume II: The MIPS32™ Instruction Set*, http://www.mips.com/publications/processor_architecture.html

[10] Lindholm, Tim and Yellin, Frank, *The Java Virtual Machine Specification,* 2nd. edition, Java Series, Addison-Wesley, USA, 1999

[11] Schmit, H.; Levine, B.; Ylvisaker, B, "Queue Machines: Hardware Compilation in Hardware," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, April 2002.

[12] L. Heath, S. Pemmaraju, and A.Trenk, "Stack and Queue Layouts of Directed Acyclic Graphs: Part {I}", in SIAM Journal on Computing, vol. 28, no. 4, 1991, pp 1510-1539.