Managing Pipeline-Reconfigurable FPGAs*

Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas

Carnegie Mellon University
Pittsburgh, PA 15213-3890
{cadambi,weener,seth,herman,thomas}@ece.cmu.edu

Abstract

While reconfigurable computing promises to deliver incomparable performance, it is still a marginal technology due to the high cost of developing and upgrading applications. Hardware virtualization can be used to significantly reduce both these costs. In this paper we describe the benefits of hardware virtualization, and show how it can be acheived using a combination of pipeline reconfiguration and run-time scheduling of both configuration streams and data streams. The result is PipeRench, an architecture that supports robust compilation and provides forward compatibility. Our preliminary performance analysis predicts that PipeRench will outperform commercial FPGAs and DSPs in both overall performance and in performance per mm^2.

1 Introduction

The cost of generating and maintaining software for reconfigurable computers is significantly higher than for general purpose computers. Unless there is a way to ease the process of developing applications for FPGAs and ways to allow the performance of applications to scale with improved silicon technology without redesign, reconfigurable computing will remain a marginal technology. In this paper, we propose hardware virtualization as the solution to this problem. We present an FPGA architecture that is capable of hardware virtualization for pipelined applications. The management of the virtualization is performed in hardware at run time, and is completely invisible to the application designer and the compiler.

Virtual memory systems use a small physical memory to emulate a large logical memory by moving infrequently accessed memory into slower cheaper storage media. This has numerous advantages for the process of software development. First, neither programmers nor compilers need know exactly how much physical memory is present in the system, which speeds development time. Second, different systems, with different amounts of physical memory can all run the same programs, despite different memory requirements. A small physical memory will limit the performance of the system, but if this performance is unacceptable, the user can always buy more memory. Furthermore, since the price of memory is ever decreasing, newer systems will have more memory and therefore the memory performance of legacy software will improve until these programs fit entirely into the physical memory in the system.

By analogy, an ideal virtualized FPGA would be capable of executing any hardware design, regardless of the size of that design. The execution speed would be proportional to the physical capacity of FPGA, and inversely-proportional to the size of the hardware design. Because the virtual hardware design is not constrained by the FPGA's capacity or I/O limitations, generation of a functional design from an algorithmic specification would be much easier than for a non-virtual FPGA and could be guaranteed from any legal input specification. Optimizing the virtual hardware design would result in faster execution, but would not be required to get an application running initially. Thus, hardware virtualization enables FPGA compilers to more closely resemble software compilers, where unoptimized code generation is extremely fast, and where more compiler-time can be dedicated to performance optimization when necessary. This accompanying benefit to hardware virtualization is called *robust compilation*.

A set of virtualized FPGAs could be constructed that all share the ability to emulate the same virtual hardware designs, but that differ in physical size. The members of this FPGA family with larger capacity will exhibit higher performance because they emulate more of the virtual design at any one time. Future members of this family, built in newer generations of silicon, could emulate virtual hardware designs at higher levels of performance *without redesign*, much like the way microprocessor families run binaries from previous generations without re-compilation. This benefit, which we call *forward-compatibility*, allows the expense of generating (or purchasing) virtual hardware designs to be amortized over multiple generations of silicon.

The technique of pipelined reconfiguration [10, 7], has been proposed as a technique to provide hardware virtualization for pipelined applications. Configurations for each pipeline stage are created at compile-time. During execution, the configuration for each pipeline stage is brought into the executing FPGA fabric, one stage every cycle. When the FPGA fabric is fully populated by active pipeline stages, older pipeline stages are replaced by newer pipeline stages. Figure 1 shows an example of this procedure for a five stage pipeline running on an FPGA with a capacity of two pipeline

*This research was supported in part by DARPA contract DABT63-96-C-0083.

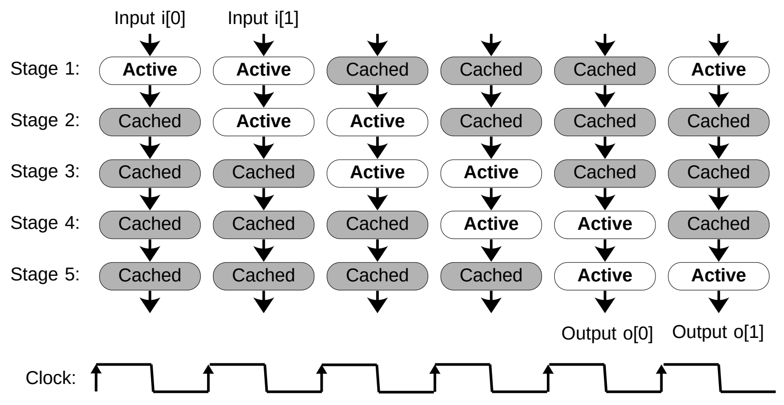


Figure 1: *Pipelined reconfiguration. An example of mapping a five stage pipeline onto a FPGA with the ability to hold two stages.*

stages. In this example, there are two results produced every five cycles. The FPGA "scrolls" through the pipelined application, and each run through the application takes five cycles and produces two results.

Future FPGAs in this family, which will utilize denser silicon to provide higher capacity, will be able to hold more virtual pipeline stages, and thereby provide higher throughput. Furthermore, a compiler for these devices does not have to know about the size (in terms of pipeline stages) of the physical hardware in order to generate a functional design.

Pipelined reconfiguration creates three significant problems for FPGA architectures. First, to perform as illustrated in Figure 1, an FPGA must be able to configure a computationally significant pipeline stage in one cycle. Second, because there may or may not be enough FPGA capacity to hold the entire pipeline, the movement of configuration data between storage and active FPGA fabric must be controlled. Third, the schedule of data memory accesses, the inputs and output to the pipeline, must be determined at run-time. This paper focuses on our solution to the latter two problems in the context of PipeRench, a co-processor we are developing at Carnegie Mellon.

The technique of pipelined reconfiguration works only on pipelined applications. However, computing workloads are becoming dominated by pipelineable algorithms in the domains of three-dimensional rendering, signal and image processing, and cryptography. Extremely fine-grained pipelining is the most important technique used by reconfigurable systems to obtain high throughput [4]. We believe that if reconfigurable systems ever become widely practical, they will be predominately applied to pipelineable applications.

1.1 Previous Work

PipeRench provides robust compilation by allowing an application to transparently exceed the logical capacity of the physical FPGA at runtime. The Virtual Wire "softwire" compiler [1] provided a degree of robustness by virtualizing the I/Os between FPGAs in a multi-FPGA logic emulation system at compile time. The challenge faced by most FPGA-based logic emulators is that the input netlist is usually too large to fit into one FPGA. The netlist must be partitioned across multiple devices and meet FPGA I/O constraints. When I/O constraints are violated, the "softwire" compiler time-multiplexes different logical I/Os on a single physical I/O. The I/O constraint violation is fixed by reducing performance. PipeRench is a single-chip FPGA computing devices, not a logic emulator. Our objective is to deal with large logical netlists, not by overflowing into other devices and dealing with I/O constraints, but by time-multiplexing the on-chip logic to emulate the desired design at a degraded level of performance.

Multiple context FPGAs [2, 3, 12], have been proposed as a way to create logically larger devices through rapid reconfiguration. These architectures do allow idle logic to be stored outside of the active FPGA fabric. These devices, however, do not meet our criteria for being virtualized FPGAs, because there is no way to create designs independently of the number of contexts. Therefore, there is no way to obtain forward-compatibility with them. Furthermore, the task of compilation for these architectures is more complex than it is for a flat, single context FPGA, because the compiler needs to place and route multiple, interdependent contexts simultaneously. PipeRench provides true hardware virtualization by allowing a design of any size to be run on the fabric.

Xilinx [11] developed and patented mechanisms to allow legacy bitstreams to be used in newer FPGAs without redesign. This type of compatibility allows the FPGA vendor to update an FPGA's architecture without making all old configuration bitstreams obsolete. Users can expect older designs to run slightly faster on new devices because in a newer process, the transistors will be faster and the interconnects will be shorter. This type of compatibility does not, however, allow exploitation of the additional numbers of transistors and interconnects present on the newer device. Using pipelined reconfiguration, a user can expect a performance increase due to both faster transistors and higher parallelism.

Pipelined reconfiguration for commercial FPGAs, such as the XC6200, has been described [7]. They do not, however, present any mechanism for control of the configuration stream and data stream with respect to the virtualization. The techniques described in this paper to control the configuration stream and data accesses apply to all devices that use pipeline reconfiguration.

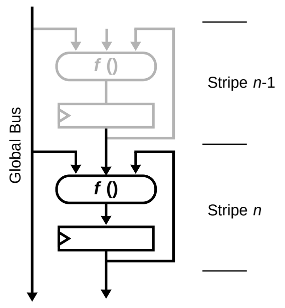


Figure 2: *Generalized stripe functionality.*

1.2 Overview

In Section 2, we discuss the architecture of PipeRench, and focus on how it allows for execution and concurrent reconfiguration. In Section 3, we discuss the requirements and design of a configuration controller, which takes care of the movement of configuration streams within the FPGA. Section 4 describes a data controller, which manages the input and output behavior of PipeRench. The performance of PipeRench for a set of FIR filters is characterized in Section 5 and compared to commercial FPGAs and DSPs.

2 PipeRench Architecture

In order to achieve high-performance and forward-compatibility, a pipeline reconfigurable device must have two architectural features. First, the architecture must support the configuration of a computationally significant pipeline stage every cycle, while concurrently executing all other other pipeline stages in the FPGA. Second, the architecture must allow different pipeline stages to be placed in different absolute locations in the physical device at different times. Only relative placement constraints are observed, so that a pipeline stage can get its inputs from the previous stage and send its outputs to the subsequent stage. No existing FPGA has these features. This section describes how these features are provided in PipeRench, and how other components of the PipeRench design connect to the active FPGA fabric in order to manage the configuration and data flows.

As described in [10], a pipeline-reconfigurable architecture requires a very high-throughput connection to the configuration memory that stores the virtual hardware design. Configuration storage in PipeRench is on-chip and connected to the FPGA fabric with a wide data bus, so that one memory read will configure one pipeline stage in the fabric. This wide configuration word is written into one of many physical blocks in the FPGA fabric. We call these blocks *stripes*, and they define the basic unit of reconfiguration in the architecture. We use the word stripe to describe both the physical structures to implement the functionality of a pipeline stage (a *physical stripe*), and the configuration word itself (a *virtual stripe*), which may or may not be resident in a physical stripe. Since a virtual stripe can be written into any physical stripe, all physical stripes must have identical functionality and interconnect.

Designing the stripe to provide adequate functionality for a wide range of applications with a limited number of configuration bits is a critical and complex task, the description of which is beyond the scope of this paper. In general, the functionality within a stripe can be described as a combinational function of three inputs: the registers within that stripe, the registers from the previous stripe, and a set of global interconnects, as shown in Figure 2. The combinational function $f()$ is defined by the configuration bits in the virtual stripe.

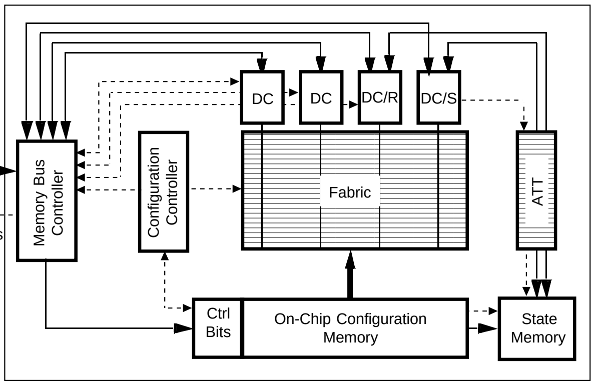


Figure 3: *Architecture Overview. Sold lines are data paths, Dashed lines are address and control paths.*

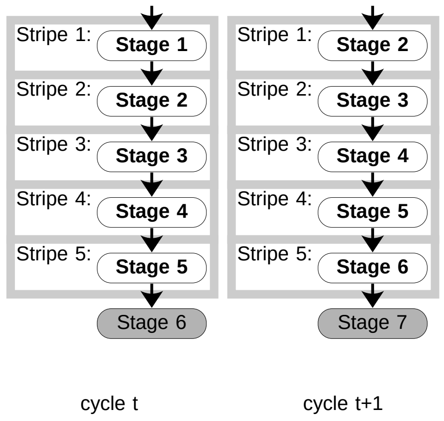
PipeRench is currently envisioned as a coprocessor in a general-purpose computer (see Figure 3). It is a memory mapped device, and has access to the same memory space as the primary processor. All the virtual stripes for all the applications that are to run on PipeRench are stored in main memory. A PipeRench“executable” consists of configuration words, which control the fabric, and data controller parameters, which determine the application’s memory read/write access pattern. The processes of loading the configuration memory and data controllers from off-chip, and configuring the fabric from the configuration memory, are the responsibilities of the configuration controller, described in Section 3.

Figure 4 illustrates two possible physical floorplans for physical stripes. In Figure 4(a), the virtual stripes move every cycle into a different physical stripe. This has two advantages: the interconnect between adjacent virtual stages is very short, and new virtual stripes are written into only one physical stripe (on the bottom). The chief disadvantage with this layout is that all the configuration data must move every cycle. This is a tremendous power sink, and it reduces performance because now the clock cycle must include the time it takes for the configuration data to move and settle.

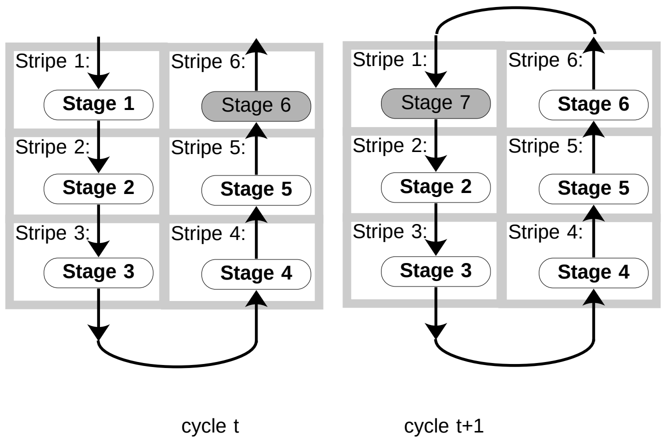
An alternative layout is illustrated in Figure 4(b), which shows the physical stripes arranged in a ring, allowing the configuration to remain stationary. The three disadvantages to this approach. First, it requires configuration data to be loaded anywhere in the fabric. Second, there is a longer worst-case interconnect between adjacent stripes (at the bottom and the top). Third, one stripe in the fabric is always configuring instead of computing resulting in a small reduction in throughput. In this example, it seems like we logically have five stripes, when in fact there are six in the fabric. At this point we believe that the disadvantages of this approach are outweighed by the power and performance advantages.

There are three types of interconnect necessary for a stripe: intra-stripe, local inter-stripe and global inter-stripe. Intra-stripe routing is used to interconnect the elements of a stripe to create the functionality of the pipeline stage.

Local inter-stripe interconnect receives inputs from the previous stripe and sends outputs to the next stripe in the



(a) Shifting Configuration



(a) Stationary Configuration

Figure 4: *Shifting and Stationary Configuration*

pipeline. Since this is a pipelined application, and each stripe contains a pipeline stage, there is no need for non-registered interconnect between non-adjacent stripes. It is essential that all local inter-stripe interconnects be registered, and that the configuration bits from one stripe cannot change anything in the path between that stripe's registers and its interconnection to the following stripe. For example, in Figure 4, the computation in stage 2 at cycle $t+1$ requires the result of the computation in stage 1 at cycle t. But in cycle $t+1$ the configuration for stage 1 is being removed from the fabric or overwritten. If a change to the configuration effects the ability of stage 2 to see stage 1's last computation, the results can not be guaranteed.

Global inter-stripe interconnect is used to get operands (and results) to (and from) any input (and output) stripes in the pipeline, to support broadcast of operands to multiple stripes, or to save and restore the state contained in a stripe's registers when it is removed or inserted from the FPGA fabric. The state in a stripe may also be initialized using the restore functionality.

At the end of each global data bus is a data controller, which handles processing of the inputs and outputs from the application. Because the sequence of data writes and reads from the fabric depends upon the number of physical stripes in the FPGA and the number or virtual stripes in the application, the data controller must do run-time scheduling of memory accesses. In order to provide the necessary memory bandwidth, the data controllers may contain memory caches to take advantage of data locality, or FIFOs to deal with "bursty" memory traffic. All the data controllers access off-chip memory through a shared memory bus control unit. This unit arbitrates access to a single memory bus. The memory bus control unit is also the path used to load the configuration memory.

Two of the data controllers have additional functionality that allow them to deal with the problem of saving and restoring a pipeline stage's state when it is removed and later returned to the FPGA fabric. The physical stripes in PipeRench are constructed to have a special path from a global bus into and out of the state registers. This path is enabled when the stripe contains state that would be lost if that stripe was removed from the fabric. The state information for each stripe is stored in an on-chip state memory. This memory has one location for each location in the configuration memory, and can therefore hold the state for any application that can fit into the configuration memory. In order to keep track of which virtual stripe is placed in each physical stripe, there is an Address Translation table (ATT in Figure 3) with one entry per physical stripe.

We have created an initial prototype of the PipeRench architecture. The stripe in our prototype consists of a set of 4-bit ALUs and registers connected with hierarchical crossbar interconnect. Based on the data gathered from this design, we believe that it is possible to build, in 50 mm^2 of 0.35 micron silicon, a PipeRench that includes 28 physical stripes, each consisting of 32 4-bit ALUs. Another 50 mm^2 of silicon could be used to create a configuration memory capable of storing 512 virtual stripes. Our simulations indicate that this chip could run at 100 MHz.

3 Configuration Management

In this section we describe how the virtual stripes of an application are mapped to the physical stripes of the hardware fabric. Since pipelined reconfigurable architectures can map an application of any size to a given physical fabric, the configuration controller must handle the time-multiplexing of the application's stripes onto the physical fabric, the scheduling of the stripes, and the management of the on-chip configuration memory. Additionally, the controller is the interface between the host, the configuration memory, the fabric, and the data controllers.

After a general description applicable to all pipelined reconfigurable architectures, we present the controller used by PipeRench. The interaction between the configuration controller and the data controller is discussed in Section 4.

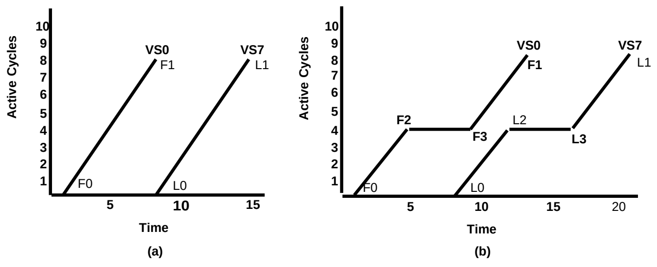


Figure 5: *Variation of the active cycles with time for (a) the non-virtualized and (b) the virtualized case. (a) shows the case for 8 virtual stripes on 8 physical stripes while (b) shows the case for 8 virtual stripes on 5 physical stripes. The two curves represent the first and the last virtual stripes (VS0 and VS7).*

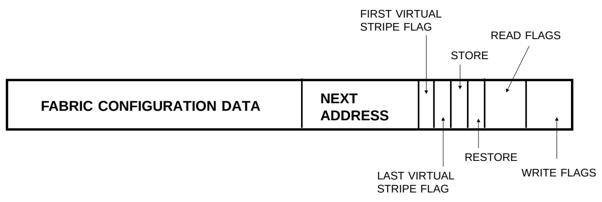


Figure 6: *The structure of a configuration word consisting of the configuration data that goes to the fabric, the next address field, and a set of flags. The flags comprise indicators for the first and the last virtual stripes, and other fields described in Section 4.*

3.1 Characteristics of a configuration controller

We break down the tasks of managing the configurations into four sub-tasks: interfacing (between the host and the fabric), mapping (the configuration words to the hardware), scheduling (time-multiplexing and managing virtualization), and managing the on-chip configuration memory.

A controller manages the interface between the host and the fabric. At the very least the interface must allow the host to initiate execution on the co-processor, and allow the co-processor to indicate that it has completed execution. In PipeRench, the host initiates a new application by specifying the main-memory address of the first configuration word of an application, the number of iterations to be performed, and the main-memory addresses for data input and output. Additionally, the host can specify the addresses of the data input and output buffers. PipeRench contains a register that indicates whether it is working or idle; this register can be periodically polled by the host.

The mapping task involves loading the virtual stripes into the on-chip configuration memory and the fabric itself. If the application fits in the fabric, the task is greatly simplified. If, however, the application is larger than the available hardware, stripes need to be swapped out during execution. Therefore, given an application, the controller must detect the case when virtualization is required and time-multiplex the application appropriately.

The controller schedules individual stripes of an application to ensure that each virtual stripe is present in the fabric long enough to process all the data: if a virtual stripe needs to be swapped out prematurely, it is reloaded later. Figure 5 shows the extent of time that the first and last virtual stripe spend in the fabric for the virtualized and the non-virtualized case. In the virtualized case, i.e., $v > p$ where v is the number of virtual stripes and p is the number of physical stripes, the number of active cycles for each stripe has a plateau of length $(v - p + 1)$ which occurs when the stripe is swapped out of the physical fabric. Each time a virtual stripe is loaded into the fabric it remains there for at most $p - 1$ active cycles. The controller thus has to swap stripes in and out at regular intervals. Points F0 and F1, and L0 and L1 in Figure 5 indicate the initial loading and completion points of the two stripes; the stripes are swapped out at the points F2 and L2, and swapped back in at F3 and L3 respectively.

Finally, the controller must use the on-chip configuration memory efficiently, since going off-chip to fetch a configuration word is time-consuming, and may lead to pipeline bubbles. If an application or multiple applications have common configuration words, these may be shared; shared configuration words need appear only once in the on-chip memory. Thus space utilization is enhanced as are the chances of fitting an application in the on-chip memory.

3.2 PipeRench's Configuration Controller

Here we present our implementation of a configuration controller for PipeRench. For the sake of simplicity, we omit discussion of pipeline stalls and present a controller that loads the entire application into the on-chip memory before beginning execution.

In PipeRench, an "executable" is composed of a series of configuration words[1] each of which includes three fields: fabric configuration bits, a next-address field, and a set of flags used by the configuration and data controllers (see Figure 6). The flags relevant to the the configuration controller are the first- and the last-virtual-stripe flags. The controller uses these to determine the iteration count and the number of stripes in the application.

The general architecture of the controller is shown in Figure 7. When the *done* line is enabled, the host can start a new application by specifying a start address and the number of iterations. The controller then lowers the *done* line until the application has completed the number of iterations specified. The (slightly simplified) algorithm in Figure 8 is used.

Mapping the configuration

Each virtual stripe in an application includes a next-address field which is used by the controller to find and then load the next stripe in the application. When the stripe is placed in the on-chip configuration memory, the next-address field is translated to an address in the on-chip memory. A record of this translation is maintained in a fully-associative on-chip Stripe Address Translation Table (SATT) (see Figure 7) [2]

A counter is used to maintain the number of virtual stripes in the application. If the number of virtual stripes is larger than the number of physical stripes in the fabric, the controller will time-multiplex the application onto the fabric.

[1]An "executable" also contains the parameters that control data accesses as discussed in the next section.

[2]The number of entries in the SATT is small compared to the size of the application: it will not be on the critical path.

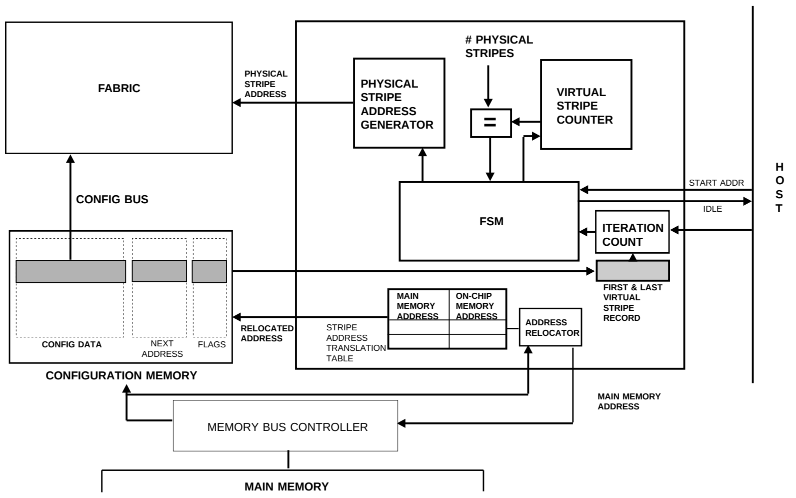


Figure 7: *The configuration controller architecture, and its interface to the host, main memory, on-chip memory, and the fabric.*

```
Given: Number of physical stripes
 while (!IDLE) {
   get starting pointer and iterations from host
   load config words from main memory
                           into on-chip memory
   load first virtual stripe
   during each configuration cycle {
     while (requested iterations not IDLE) {
       if (not last virtual stripe)
         increment physical stripe address
         load next virtual stripe
       else {
        if (# virtual stripes > # physical stripes)
         increment physical stripe address
         restart loading from first virtual stripe
       }
       decrement iterations
     }
   }
   assert IDLE signal
 }
```

Figure 8: *Algorithm for configuration management.*

Configuring physical stripes

On every cycle the controller enables a specific physical stripe to be reconfigured. PipeRench uses a counter modulo the number of physical stripes to sequentially generate physical stripe addresses. This simple method automatically ensures that if the application is too big to fit in the fabric, configured stripes are overwritten and the hardware is virtualized over the entire physical fabric.

Virtualized execution: keeping track of iterations

Once stripes are overwritten, they may need to be reloaded since all the requested iterations may not have been performed (i.e., each stripe may not have processed all the data required). In order to do this and execute an application for a certain number of iterations, we use two of the flag bits: the first-virtual-stripe flag and the last-virtual-stripe flag.

When the first virtual stripe is loaded into the fabric, the controller updates a record: it notes the address of the physical stripe where it was loaded. By monitoring this record during loading and swapping stripes, it can ascertain the number of cycles the first virtual stripe has spent in the fabric (i.e., the number of iterations it has executed). In addition to monitoring the first stripe, the controller also monitors when the last virtual stripe is swapped into the fabric.

Using the first and the last stripe, the iteration count may be managed in the following manner: when the first virtual stripe completes its required number of iterations, it does not need to be reloaded ever again. Hence the loading of the application can now stop (and a new application may be started) after loading the last virtual stripe.

3.3 Summary

In this section, we analyzed and described the four main subtasks of configuration management for pipelined reconfigurable architectures: interfacing, mapping, scheduling and memory utilization. In our implementation of the configuration controller for PipeRench, we use a next-address field to access configuration words from memory, use a counter (modulo the number of physical stripes) to generate the physical stripe addresses, and identify the first and last stripes by flags in order to keep track of iterations. This simple configuration controller can map an application with any number of virtual stripes onto a fabric with a given physical size.

4 Data Management

Managing the flow of data for virtualized applications is one of the main challenges in designing a pipelined reconfigurable architecture. Virtualization can cause disruptions in the flow of data, requiring the explicit management of execution state. The key to a successful pipelined reconfigurable architecture is to make these disruptions transparent to the application designer. This section presents our data controller architecture and shows how it manages the virtu-

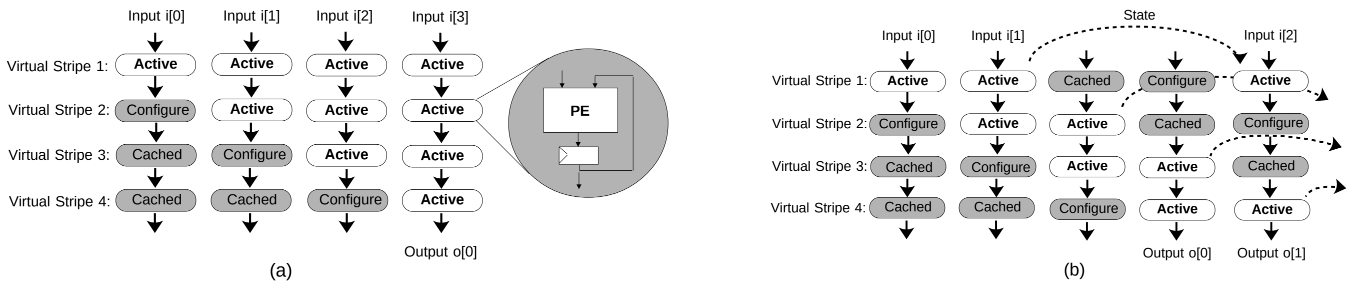


Figure 9: *Comparing input/output and state management with no virtualization and virtualization. (a)With enough hardware (no virtualization) there is no need to save state and input/output timing remain unchanged, (b) With less than enough hardware (virtualization) a stripe's state must be saved and input/output timing changed.*

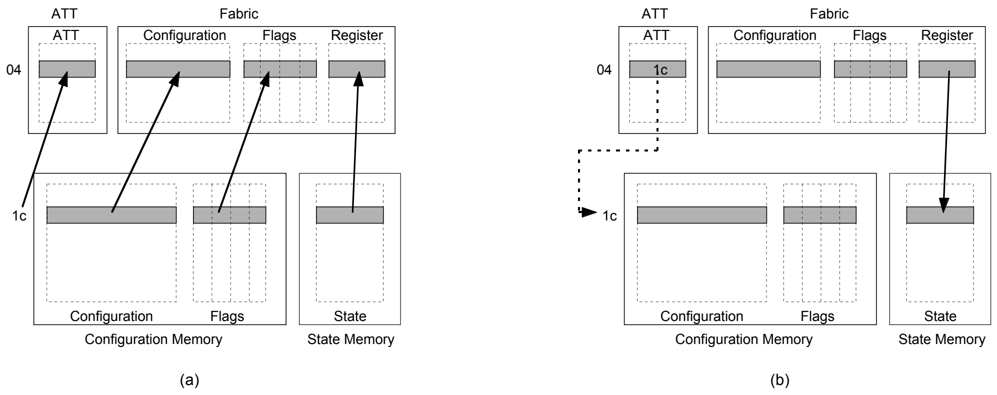


Figure 10: *Restoring (a) and storing (b) state between fabric, configuration memory, state memory and ATT.*

alization of two implementations of a convolution application.

When there is no virtualization, there is no need to store and restore state or change input/output timing. Figure 9(a) shows the execution of a simple application with no virtualization. Though PEs may contain functions of their own registered outputs, there is no need to save state because all the configurations remain in the fabric. Also, input and output are needed every cycle since the stripes that need input and output remain in the fabric.

However, when applications are virtualized, the stripe state may need to be remembered and the input/output timing changed. Figure 9(b) shows the execution of the same application, which now requires virtualization since there are only three physical stripes for the four virtual stripes. When stripes are functions of their own registered outputs, the state of that stripe must be stored while its configuration is not in a physical stripe and restored when it is returned to the fabric. Furthermore, input and output are only needed when the stripes that consume or produce data are in the fabric. In the example in Figure 9, input (output) is only needed when the first (last) stripe is in the fabric.

4.1 Data controller Architecture

The data controller architecture consists of four separate data controllers (see Figure 3). Each controller manages one global bus which is dedicated to either state storing, state restoring, data input or data output per application. When a controller is dedicated to storing or restoring state, the data controller interfaces between the fabric and the state memory. When a controller is dedicated to inputting or outputting data, the controller interfaces between the fabric and the memory bus controller. To determine which task each data controller performs, controllers contain control registers which describe functionality. The control registers specify the beginning data address, stride, and whether that bus is used for input, output, store, or restore.

Managing Stripe State

When needed, a stripe's state is kept in the state memory (see Figure 11), which is addressed differently for stores and restores. During a restore, which takes place in the configuration cycle, the state memory address is the same address as that used to access the configuration memory. As Figure 10(a) shows, when a stripe's configuration is written into the fabric, that stripe's state and flags are also written. In order to remember the address in the state memory for that stripe's state, the configuration memory address is written into the Address Translation Table (ATT). When storing state, the ATT supplies the state memory address, as shown in Figure 10(b).

Managing Data Input/Output

When managing Input/Output, configured stripes communicate with the input and output controllers through flags, and these controllers communicate via address and control logic with the memory bus controller. Each controller receives the flag bits that show the read and write data requests for its corresponding bus (Read Flags and Write Flags in Figure 6). The flag bits are part of each stripe's control word and specify if that stripe reads or writes to each of the

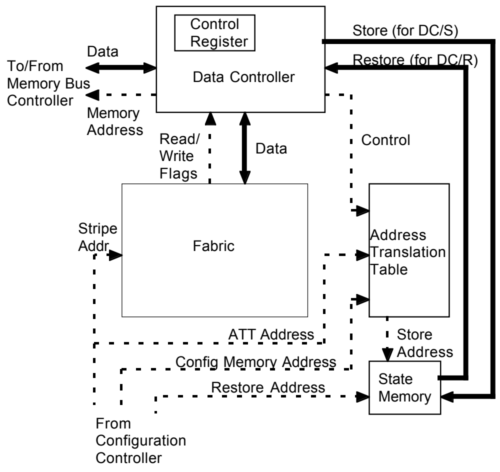


Figure 11: *Data controller architecture - Solid lines are data and dashed lines are address or control.*

four buses. The data controllers receive these flag bits from the fabric and generate the necessary address and control lines for the memory bus controller (see Figure 11). Therefore, when a stripe is configured to produce data on a bus, the controller generates the appropriate signals to write the data (likewise for a read).

The data controller is also responsible for generating the addresses for both the input and output data streams. We currently can generate addresses that are affine functions of the loop index. The starting address is supplied by the host when the application starts and the stride is specified as part of the application. When the fabric performs a read or write, the next address is in the sequence is generated by incrementing the current address by the stride. We are examining ways of generating addresses for a richer set of applications.

4.2 Example convolution data flow

To make the function of the data controllers more concrete, we now present two different implementations of the convolution shown in Figure 12. Using the terminology presented in Kung [6], we present a systolic example where the data flow is well-suited to PipeRench and easily managed by our controller architecture, and as an example of a mapping less-suited to our architecture, a semi-systolic implementation. The weights in both these examples are stationary in the stages. This leads to savings in hardware since the constant weights may be propagated through the multiplier and therefore configured into the hardware. The following examples all assume that the functionality for one tap of this convolution can be supplied by one stripe. In reality, a multiply-by-constant operation may require several stripes ($< n/2$ for an n-bit multiplicand), depending on the functionality of the stripe and the value of the constant. These techniques can be easily generalized to deal with multiple stripe taps.

Double Pipelined X convolution

Figure 13 shows a fully systolic implementation that contains a single pipelined output Y, a double pipelined input X, and stationary weight W. The X's enter the pipeline from the first stage. Every cycle a new X with a higher index is

```
for i=1 to NumberOfInputs {
  Y[i] = 0
  for j=1 to NumberOfTaps {
    Y[i] = Y[i] + X[i+j]*W[j]
  }
}
```

Figure 12: *Convolution algorithm.*

inserted. The data controller for this bus addresses the data memory from the beginning address supplied in its control registers. The data is driven on the bus and is read by the first stripe. When the first stripe asserts the corresponding read flag, the data controller increments the memory address by the contents in the stride register (in this case, 1) and readies the next piece of data on the bus. A controller for the pipelined Y output is similar, with the exception that it monitors the write flags and writes the data into memory instead.

In this example, some of the data in a stripe needs its state stored or restored. The double pipelined X contains state that needs to be stored and restored; the registered feedback is from the first register delay to the second register delay in the same stripe. The single pipelined Y value does not require storing or restoring since the stripe's functions do not contain registered feedback.

Broadcast X convolution

Figure 14 shows a semi-systolic implementation. This implementation broadcasts the X values to all stages at one time. The Y values are single pipelined through the array of stages, and the weights are stationary. Since all stages are single pipelined there is no state to store or restore, and therefore this implementation does not need buses for store or restore. However, it is difficult to implement because not all stripes expecting data will be in the fabric at the time the data is broadcast.

Two possible solutions are to either use two input buses and broadcast the data multiple times, or have the configuration controller insert stalls in the configuration stream until the bus is available. The first solution is not transparent to the application. It requires double the number of virtual stripes since one set of stripes reads from one bus and a second set reads from another bus. The second and preferable solution is transparent, but reduces throughput and requires the configuration controller to stall between configuring the last virtual stripe (the end of one iteration through the virtual stripes) and the first virtual stripe (the beginning of the next iteration through the virtual stripes). Figure 15 shows how the stalls could be used to keep the semi-systolic implementation possible.

5 Performance

In this section we compare the expected performance of our architecture against commercial FPGAs with similar processing technology and area, and against commercial DSP processors on FIR filters of varying sizes.

Based on our design of the PipeRench prototype in 0.5 micron silicon, we believe that in 50 mm^2 of 0.35 micron silicon it is possible to have 28 stripes, each with a 128-bit wide datapath. Expected cycle time for this datapath is

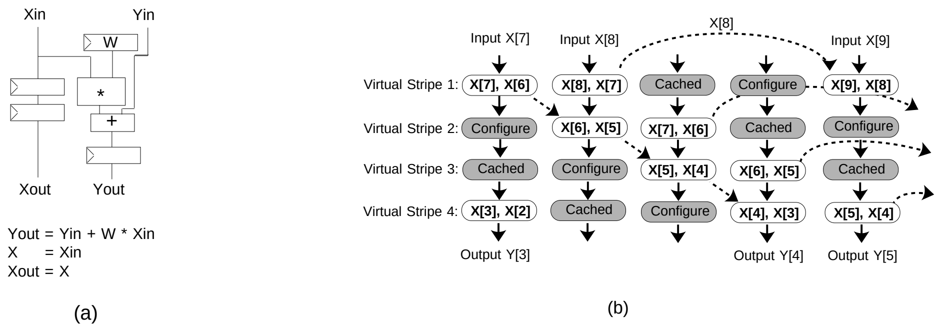


Figure 13: *Systolic Convolution. (a) Each stage's function contains a double pipelined X input, single pipelined Y output, and stationary weight W, (b) Example of the data flow for this implementation. The dashed lines indicate how Y is accumulated as time progresses. The dashed arcs indicate state store and restore.*

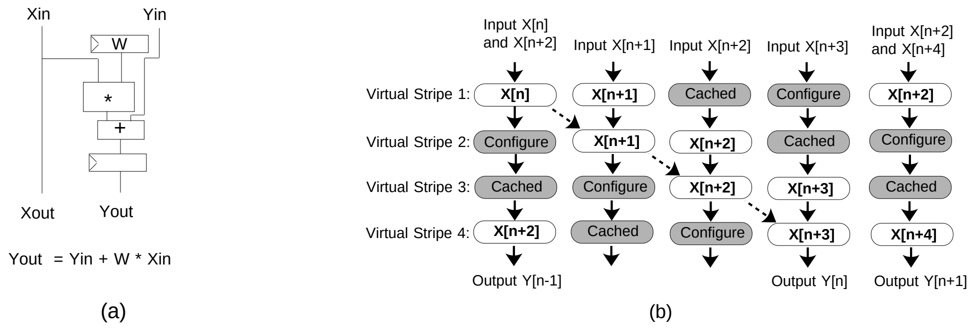


Figure 14: *Semi-Systolic convolution. (a) Each stripe's function contains a broadcast X, single pipelined Y, and stationary W, (b) Example of the data flow for the broadcast convolution. The dashed lines indicate how Y[n] is accumulated as time progresses. Notice that at some time steps multiple inputs must be supplied to the fabric.*

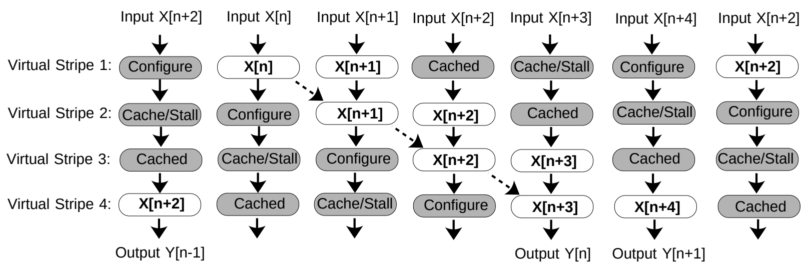


Figure 15: *Example of the data flow for the broadcast convolution. The configuration controller inserts stalls to reduce bus contention(which cause the virtual stripes to remain cached longer). The dashed lines indicate how Y[n] is accumulated as time progresses.*

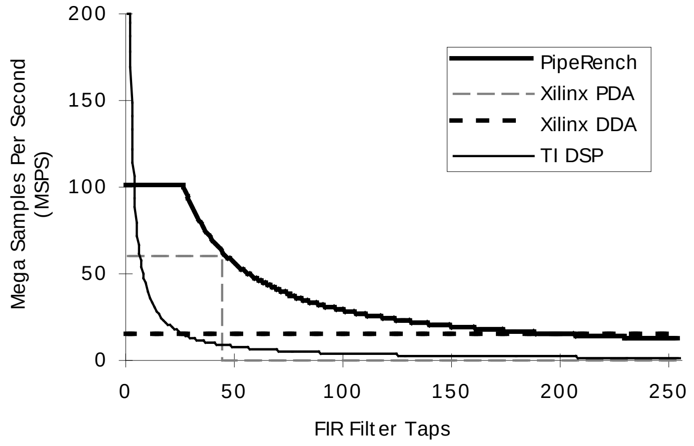


Figure 16: *Performance on 8-bit FIR filters: PipeRench, Xilinx FPGA using parallel and serial arithmetic and Texas Instruments DSP.*

100 MHz. An SRAM for configuration memory will consume another 50 mm^2 of area, and will store 256 configuration words of 1024 bits each. One 128-bit wide stripe is capable of holding one tap of a 8-bit FIR filter with 12-bit coefficients. As shown in Figure 16 this enables an FIR filter with less than 29 taps to run at the full clock rate of 100 MHz. Larger filters demonstrate a graceful degradation of performance out to around 256 taps, at which point the on-chip configuration storage is full. For larger filters, smart cache management techniques can be used to continue the degradation, albeit at a steeper rate due to the need to fetch some configuration data from off-chip.

Based on measurements of Xilinx FPGAs built in 0.35 micron technology [9], we believe that 100 mm^2 of area represents about 1750 CLBs. Given this amount of logic, and using parallel distributed arithmetic (shown as Xilinx PDA in Figure 16), it is possible to create filters that run at around 60 MHz and have up to 48 taps [8]. More than 48 taps will not fit, which effectively causes performance to fall to zero. Using double-rate distributed arithmetic (shown as Xilinx DDA), it is possible to construct extremely large filters given this amount of silicon [8]. Due to the bit serial nature of these implementations however, the maximum sampling rate of these filters is 14 MHz.

The Texas Instruments TMS320C6201 [5] is a commercial DSP which runs at 200 MHz and contains two 16- by 16-bit integer multipliers. For filters with small numbers of taps, the high clock speed of this device yields extremely high performance. This performance decays rapidly with an increasing number of taps due to the presence of only two multipliers. PipeRench exhibits higher performance than the DSP. With respect to performance, PipeRench is more like an FPGA. But with respect to performance degradation as the size of the filter grows, PipeRench is more like an instruction set processor such as this DSP.

6 Conclusions

Pipeline-reconfigurable FPGAs provides several benefits for reconfigurable computing, including forward-compatibility and more robust compilation. We believe these benefits enable the development of FPGAs that have the performance advantages for DSP applications associated with current FPGAs, and the ease and economy of development associated with microprocessors. Managing the configuration and data flows is a significant issue in the design of these devices. PipeRench's configuration controller performs run-time mapping and scheduling of configuration transfers, interfaces to the host processor, and manages the configuration storage. The data controllers provide mechanisms for storing and restoring of state, as well as access to operand data for a variety of systolic and semi-systolic pipeline implementations. A prototype of the PipeRench architecture has been fabricated and is currently being tested.

References

[1] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Napa, CA, April 1993.

[2] N. Bhat, K. Chaudhary, and E.S. Kuh. Performance-oriented fully routable dynamic architecture for a field programmable logic device. M93/42, 1993. U.C. Berkeley.

[3] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Napa, CA, April 1994.

[4] B. Von Herzen. Signal processing at 250mhz using high-performance FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 62–68, Monterey, CA, February 1997.

[5] Texas Instruments. TMS320C6201 digital signal processor, revision 2, 1997.

[6] H. T. Kung. Why systolic architectures? In *IEEE Computer*, pages 37–45, Piscataway, NJ, January 1982.

[7] W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung. Pipeline morphing and virtual pipelines. In W. Luk and P. Y. K. Cheung, editors, *Field-Programmable Logic and Applications*, London, England, September 1997. To be published.

[8] B. Newgard. Signal processing with Xilinx FPGAs, September 1996. `http://www.xilinx.com/appnotes/sd_xdsp.pdf`.

[9] J. Rose. Private communications, 1997.

[10] H. Schmit. Incremental reconfiguration for pipelined applications. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 47–55, Napa, CA, April 1997.

[11] S. Trimberger. Field programmable gate array with built-in bitstream data expansion. U.S. Patent No. 5,426,379, June 1995.

[12] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, April 1997.