
Journal of VLSI Signal Processing Systems 24, 129–146, 2000.

Pipeline Reconfigurable FPGAs*

HERMAN H. SCHMIT, SRIHARI CADAMBI AND MATTHEW MOE
Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA

SETH C. GOLDSTEIN
Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

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 achieved using the technique of pipeline reconfiguration. The result is PipeRench, an architecture that supports robust compilation and provides forward compatibility. Our preliminary performance analysis on PipeRench predicts that it will outperform commercial FPGAs and DSPs in both overall performance and in performance normalized for silicon area over a broad range of problem sizes.

1. Introduction

Components in a signal processing system are typically implemented in one of two ways: (1) custom hardware or (2) software running on a processor. The advantage of implementation in hardware is that it can exploit the correct amount of parallelism in order to meet performance constraints while minimizing either the power or per-unit cost of the system. The chief problem with hardware implementations of signal processing components is the time and money consumed by the many steps of the design process and the high non-recoverable costs of fabrication. As a result of these high costs, hardware solutions are only feasible in systems that are either cost-insensitive, where the high development cost is tolerated, or systems that are produced in very high volume, where the development cost is absorbed by the lower per-unit cost of a hardware implementation.

Field-programmable Gate Arrays (FPGAs) have enabled the creation of hardware designs in standard, high-volume parts, thereby amortizing the cost of mask sets and significantly reducing time-to-market for hardware solutions. However, engineering costs and design time for FPGA-based solutions still remain significantly higher than software-based solutions. Designers must frequently iterate the design process in order to meet system performance requirements while simultaneously minimizing the required size of the FPGA. Each iteration of this process takes hours or days to complete.

Another way to reduce the effective costs of hardware design would be to frequently re-use hardware components in multiple systems. However, hardware designs are difficult to port to different process technologies. Furthermore, it is inefficient or impossible to re-use a component in a system that requires significantly more or less performance than the original component, because the parallelism exploited by a component is fixed by the original designer.

In this paper, we describe a technique, called *hardware virtualization*, that solves the problem of hardware re-use. We present techniques to virtualize pipelined applications using existing FPGA architectures. Based on the shortcomings of these techniques, we present a new method of hardware reconfiguration, called *pipeline reconfiguration*, that enables efficient hardware virtualization for pipelined applications.

*This work supported by DARPA, under contract DABT63-96-C-0083.

1.1. Hardware Virtualization

Hardware virtualization frees a designer to create a hardware design that exploits a very large amount of parallelism but also consumes a great deal of silicon area. This large hardware design can be emulated on a much smaller amount of physical hardware at a reduced level of performance. The emulation of the large design (or *virtual hardware design*) is accomplished by time-multiplexing programmable hardware.

The closest analog to the ideal of virtual hardware is virtual memory in processor systems. In virtual memory, a small physical memory is used 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 simply buys 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.

Similarly, 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, 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 initially implement or prototype the application. Thus, hardware virtualization enables FPGA compilers to more closely resemble software compilers, where unoptimized code generation is extremely fast, and where more compilation time can be dedicated to performance optimization when necessary. This accompanying benefit to hardware virtualization is called *robust compilation*.

A family 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 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*, increases the return on investment in FPGA applications. In other words, the expense of generating (or purchasing) virtual hardware designs can be amortized for many systems with different performance and cost requirements, over multiple generations of silicon.

1.2. Pipeline Reconfiguration

This paper focusses on the virtualization of hardware applications that can be formulated as a pipeline. A pipeline is a systolic array [1] where all data flow goes in one direction and there is no feedback. Figure 1 illustrates two stages of a FIR filter application transformed in such a way to meet these requirements. Transforming algorithms into pipelines is a well-understood problem. Some of the techniques for transforming algorithms into pipelined implementations are presented in [2–4]. Fortunately, a large percentage of computationally challenging applications can be implemented as pipelines, including many in the domains of three-dimensional rendering, signal and image processing,

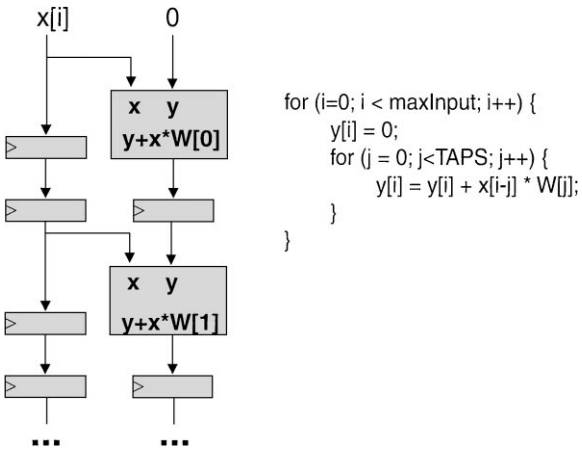


Figure 1. A pipelined FIR filter. All wires propogate in a single forward direction. One additional benefit to this design is that all multiplications have one constant operand, allowing further hardware optimization.

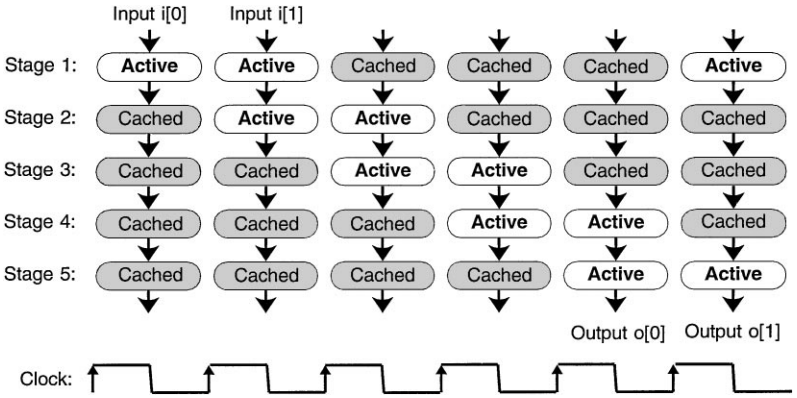


Figure 2. Pipeline reconfiguration. An example of mapping a five stage pipeline onto a FPGA with the ability to hold two stages.

and cryptography. Furthermore, extremely fine-grained pipelining is the most important technique used by reconfigurable systems to obtain high throughput [5]. If reconfigurable systems become widely used, they will be predominately applied to pipelineable applications.

Pipeline reconfiguration is a new way to use the reconfigurability of FPGAs to virtualize pipelined applications. In pipeline reconfiguration, the configuration bits corresponding to each pipeline stage are 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 2 shows an example of pipeline reconfiguration for a five stage pipeline running on an FPGA with a capacity of two active pipeline 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. Therefore the throughput of this implementation is two-fifths of the throughput possible without virtualization. The input and output behavior of this implementation is modified from the non-virtualized implementation. Input and output from the virtualized pipeline occurs in two-cycle bursts that repeat every five cycles. Ideally, virtualized FPGA should accomodate this burstiness without requiring the involvement of the pipeline designer.

In Section 2, we first present ways to virtualize pipelines using traditional FPGA reconfiguration techniques. We quantify the latency and throughput of these techniques based on system parameters such as FPGA capacity and reconfiguration time. Then we compare these techniques to pipeline reconfiguration. We show that reconfiguration time is the most important factor in the performance of all these systems. We also show that pipeline reconfigurable devices avoid many of the other problems with traditional reconfiguration, including pipeline fill and empty penalties and memory capacity problems.

In the remainder of the paper, we address a number of architectural challenges for pipeline reconfiguration FPGAs. Each section addresses one of the three significant problems for these architectures. The first problem is reconfiguration time. For maximum performance, a pipeline reconfigurable FPGA should be able to configure a computationally significant pipeline stage in one cycle. Section 3 describes the PipeRench architecture, which is designed to minimize the impact of reconfiguration time on performance. The second problem is how to control the pipeline reconfiguration at run-time in order to accurately virtualize hardware. Section 4 presents the PipeRench configuration controller, which controls the movement of configuration data between storage and active FPGA fabric. The third problem, as illustrated in Fig. 2, is that the schedule of inputs and outputs to the pipeline is dependent on the virtualization and must be determined at run-time. Section 5 presents the PipeRench data controller, which performs these functions. Section 6 presents the estimated performance of PipeRench for a set of pipelined FIR filters and compare to both commercial FPGAs and DSPs, and Section 7 presents some comparisons of pipeline reconfiguration to related work in FPGA and computer architecture.

2. Pipeline Virtualization

In this section, we evaluate methods to virtualize pipelined applications on standard FPGAs using conventional reconfiguration, and we compare it to pipeline reconfiguration in terms of throughput and latency.

2.1. Component-Level Reconfiguration

Popular commercial FPGAs such as the Xilinx 4000 family [6] and the Altera FLEX family [7] have exclusive operational and configuration modes. There is no mechanism to allow simultaneous operation and configuration or even partial modification of a configuration that is already loaded. The atomic unit of reconfiguration is the whole chip, therefore the chip is only capable of *component-level reconfiguration*. Configuration data itself is fed into these FPGAs through a small number of I/O pins. Configuration times can therefore be thousands or hundreds of thousands of times longer than the operating cycle time of a design. As we will show, this long configuration time hurts throughput, latency and memory requirement for pipelined application. *Dynamic partial reconfiguration* is a variation on component-level reconfiguration that allows the configuration memory to be written simultaneously with the operation of the chip. It was present in the Xilinx 6200 family [8]. This mode needs to be very carefully used, as it does not prevent the reconfiguration from interfering with the computation on the device.

Another type of configuration mechanism is the *multiple-context configuration*, as discussed in [9–11]. This mechanism is similar to that in a standard FPGA, except that instead of having one configuration stored in the FPGA, n complete configurations are loaded into the FPGA. A global selection bus determines which one of the n configuration should be used during the current cycle. Logical reconfiguration of the entire FPGA can be accomplished in a time comparable to the execution cycle time of the design, but the atomic unit of reconfiguration remains the whole chip.

While multiple-context configuration solves configuration speed problem, it does have limitations. First, the process of switching contexts moves a large amount of configuration data in a short period of time. Context-switching is therefore a power-intensive operation. Furthermore, the amount of "virtual" hardware emulated by a multiple-context FPGA is limited to n times the physical hardware in that FPGA. Reconfiguration beyond n contexts must take place on a low-speed, narrow configuration bus. Finally, as we shall demonstrate, component-level reconfiguration has significant disadvantages for virtualization of pipeline designs.

2.1.1. The Application. The application we will examine is a very deeply pipelined application, such as a high-order FIR filter implemented as shown in Fig. 1. Assume that this application has S identically-sized pipeline stages. Further assume that there are D bytes of data flowing between each stage of the filter every cycle, and between the filter input and output. This last assumption rarely holds in real pipelined applications. In most pipelines, the intermediate data between any two stages is much greater than D. This assumption greatly simplifies the following analysis, however.

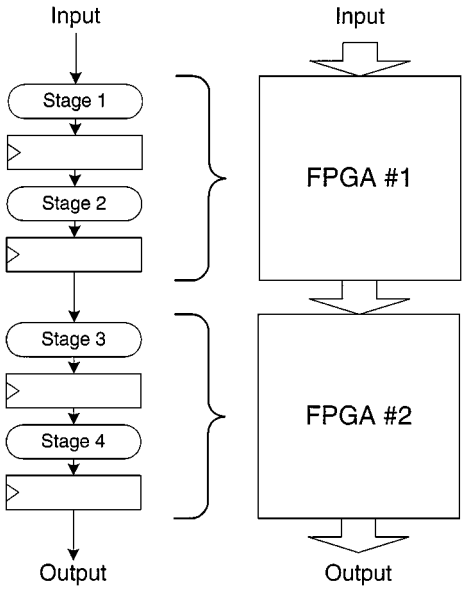


Figure 3. Example pipeline application: Four stages implemented on two FPGAs. $S = 4$ and $N = 2$.

To implement this application, we have FPGAs with a fixed logic capacity. Assume that in order to statically implement the whole filter we would require N of these FPGAs, as illustrated in Fig. 3. If the clock cycle time of the FPGA, as determined by the most complex pipeline stage is T, then the throughput of the static, N-FPGA implementation of this filter is D/T bytes per second. To simplify the analysis, we will not consider the time that it takes to configure this application initially, or to swap between different applications.

2.1.2. Virtualization. We will use component-level reconfiguration to implement this filter in one FPGA of similar capacity. The theoretical maximum throughput of the 1-FPGA implementation of this filter using Run-Time Reconfiguration (RTR) is $D/(NT)$ simply due to the reduction in computing hardware. We will examine the implementation of this filter using component-level reconfiguration in terms of its performance characteristics and memory requirements.

Using component-level reconfiguration, the N different FPGA configurations from the N-FPGA design sequentially configure a single FPGA. This level of reconfiguration has also been called *Global RTR* [12]. The configuration controller loads one configuration, and allows the FPGA to perform operations on X words of data. It takes S/N cycles to get the first result from this configuration, and $X-1$ cycles to get the remaining results. Therefore, the time required to complete these computations, in seconds, is:

$$T(X-1+S/N) \tag{1}$$

After this computation is complete, the system controller reconfigures the FPGA with the next configuration in the sequence as illustrated in Fig. 4. If it takes C cycles to reconfigure the FPGA, then the throughput of this implementation can be described using the formula:

$$\frac{DX}{NT(X-1+S/N+C)} \tag{2}$$

$$= \frac{D}{T\left(N+\frac{S-N}{X}+\frac{NC}{X}\right)} \tag{3}$$

Throughput falls short of the ideal due to the pipeline penalty and a reconfiguration penalty. The pipeline

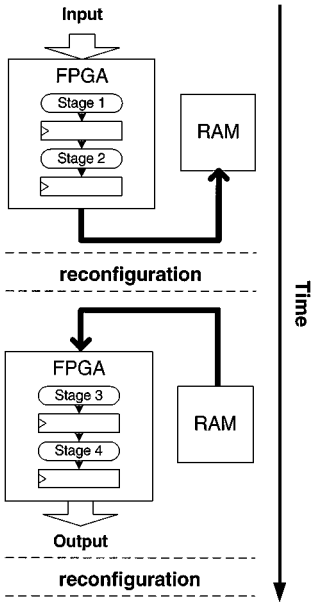


Figure 4. Component-level reconfiguration: Virtualization of pipelined application through reconfiguration of one FPGA with RAM to store intermediate results.

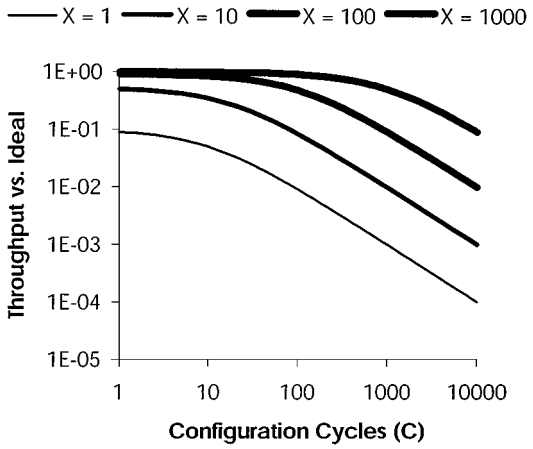


Figure 5. Throughput versus configuration time: Component-level configuration for various values of X. $S = 100$ and $N = 10$. This is a log-log plot.

penalty, which is expressed in the $(S-N)/X$ term, is the penalty suffered for having to repeatedly fill and empty the pipeline between reconfigurations. The reconfiguration penalty, which is expressed in the $(NC)/X$ term, is caused by the non-zero reconfiguration time of the device. The relationship of C, X and throughput is shown in Fig. 5. In this graph, $S = 100$ and $N = 10$. The ideal performance of this implementation is $D/(10T)$. The value on the y-axis indicates how actual throughput compares to this ideal. For the moment, assume X is small. (Note: when $X = 10$ the pipeline is just filled, and then emptied.) When C is large, as in the case of standard FPGAs, the throughput is unacceptably low. When C is small, as is the case with the multiple-context FPGAs, the pipeline penalty limits throughput.

Increasing X will increase the throughput of the implementation regardless of C, but by increasing X the latency of the implementation is also increased. The latency for this implementation is:

$$NT\left(X+\frac{S}{N}-1+C\right) \tag{4}$$

The second problem with increasing X is that it is necessary to have enough memory to store all the data output from one block during reconfiguration so that it can be used as the input to the next block of the pipeline. The required amount of memory is DX.

In addition, assuming input data arrives as a rate not greater than the throughput rate, any virtualized implementation will require a buffer to store inputs while the

Table 1. Commercial FPGA configuration times (clock frequency: 33 MHz).

Part	Config. time	C
XC4028EX [6]	8.35 ms	275,000
XC6216 [8]	92 μs	3036

lower stages of the pipeline are being executed. This buffer would also need to have a capacity of DX bytes.

If X is very large, it will be difficult to meet these memory requirements on the same chip as the FPGA. If this is the case, the time required to access off-chip memory may increase T, degrading performance of the whole system.

Table 1 shows typical values of C for two available Xilinx components using the fastest configuration mode available for that component. These results were computed assuming a modest operating frequency of 33 MHz. Obviously, reconfiguration time is going to play a critical role in determining throughput, latency and memory requirements for applications which use these components.

Multiple-context FPGAs have a C value of one cycle or less. While this effectively eliminates the configuration penalty, it does not reduce the effect of the pipeline penalty. In addition, multiple-context FPGAs only have a low C if the number of contexts held in the device is greater than N for the particular application. If a multiple-context FPGAs can have inactive configurations modified while simultaneously executing another configuration, then it would be possible to extend the virtualization. But this would require X to be large enough to hid the reconfiguration time of the inactive configuration.

2.2. Pipeline Reconfiguration

Pipeline reconfiguration is a restricted form of local RTR [12], in which the pipeline is separated into S components, each corresponding to one pipeline stage. The FPGA can hold P of these pipeline stages, and the reconfiguration happens in an incremental manner. In order to normalize the capacity to the previous analysis, $P = S/N$. During each stage of the computation, we add one additional stage to the configuration, and remove (or overwrite) a stage if necessary to keep the amount of configuration within the capacity of the FPGA. Figure 2 illustrates this procedure. Reconfiguration in this manner can be visualized as the scrolling of a window through the computation.

2.2.1. Virtualization. As with component-level reconfiguration, we will assume that the time it takes to execute a pipeline stage is T in seconds, and the number of FPGAs required to hold the whole application is N. The number of execution cycles required to reconfigure the entire FPGA is C. Therefore, the time required to substitute a new pipeline stage into the configuration is, ideally, TC/P. For one complete sweep through the application, S stages must be configured, requiring $TCS/P = TCN$ seconds. Execution of the entire pipeline will take S cycles for the first element of data, and $P-1$ cycles to process the remaining data in the pipeline. Therefore, the throughput of this implementation is equal to:

$$\frac{DP}{T(S+P-1+CN)} \tag{5}$$

$$= \frac{D}{T\left(N+1-\frac{1}{P}+(N^2C)/S\right)} \tag{6}$$

The best-case latency of this implementation is:

$$T(S+CN) \tag{7}$$

Figure 6 shows the relationship of throughput to the configuration cycles, C. For comparison, two curves for component-level reconfiguration with $X = 100$ and $X = 10$ are shown. Figure 7 shows the plots of latency for the same three implementations. These graphs show that when C is small, the pipeline reconfigured implementation exhibits both high throughput and low

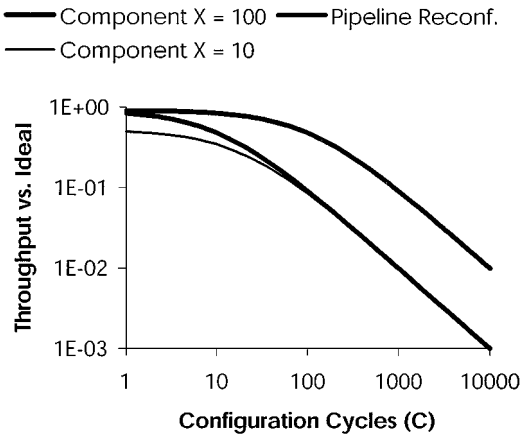


Figure 6. Throughput versus configuration time: Pipeline reconfiguration compared to component-level configuration. $S = 100$ and $N = 10$.

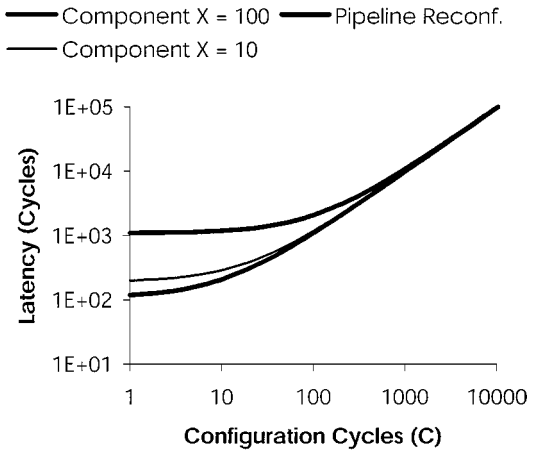


Figure 7. Latency versus configuration time: Pipeline reconfiguration compared to component-level configuration. $S = 100$ and $N = 10$.

latency. When C is large, the throughput exhibited by the pipelined reconfigured design exhibits behavior very similar to the component-level reconfigured implementation with $X = S/N$.

These graphs again demonstrate the importance of configuration cycles, C, in the throughput and latency equations. As C approaches zero, the throughput and latency of the pipeline reconfigured FPGA approach their respective theoretical optima. Component-level reconfigured implementations can only trade throughput for latency, and can therefore never optimize both quantities simultaneously.

Another advantage of pipeline stage reconfiguration is that all intermediate results remain stored in the appropriate pipeline stage. There is no need for supplemental storage. The front-end storage to buffer arriving inputs must still be present, but it needs only store DS/N bytes, as opposed to DX bytes.

The most important characteristic of incremental pipeline reconfiguration is that the presence of more hardware transparently results in higher throughput.

Pipeline reconfiguration requires the ability to modify only a portion of the FPGA at a time. Therefore it is only possible using dynamically reconfigurable FPGAs, such as the Xilinx 6200 [8]. Using the Xilinx 6200 to virtualize pipelines was described in [13]. The primary problem with using pipeline reconfiguration on an on-line reconfigurable FPGA like the XC6200 series is that the relatively low bandwidth of the configuration bus may make the effective value of C quite large. This limitation could be fixed by incorporating an on-chip configuration cache and widening the connection between the memory and the FPGA fabric.

For these reasons, we have designed an FPGA architecture specifically for pipeline reconfiguration, which we call PipeRench. PipeRench is capable of configuring a stage of the pipeline concurrently with the execution of the rest of the pipeline. Because of this concurrency, C effectively equals zero (even though the entire device still requires TP to be configured), and the performance approaches the theoretical maximum. The architecture and operation of PipeRench will be described in the following section.

3. 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 pipeline stages in the FPGA, i.e. $C = 0$. 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 should need to be 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.

In order to configure a pipeline stage every cycle, 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

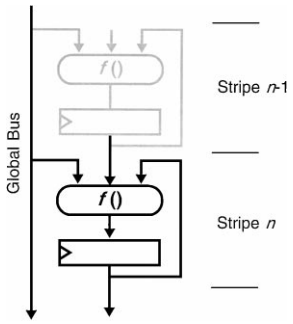


Figure 8. Generalized stripe functionality.

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 Fig. 8. The combinational function $f()$ is defined by the configuration bits in the virtual stripe.

Note the feedback path from the register in a stripe back into the combinational function $f()$. If this path is used by an application, the register bits that are fed back contain state information that must be maintained by the device. We call this information *stripe state*.

PipeRench is currently envisioned as a coprocessor in a general-purpose computer (see Fig. 9). 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 4.

Figure 10 illustrates two possible layouts for physical stripes. In Fig. 10(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 Fig. 10(b), which shows the physical stripes arranged in a ring, allowing the configuration to remain stationary. There are two 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). But because only one stripe needs to be reconfigured, it is possible to configure that one stripe while simultaneously executing the application in the remaining stripes. In Fig. 10(b), five stripes are computing, despite the

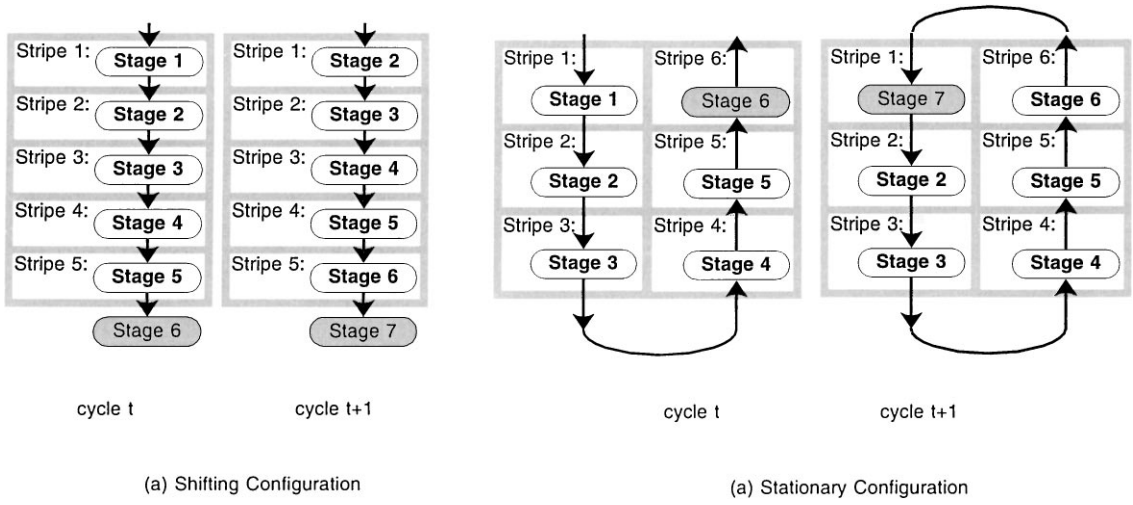


Figure 9. Shifting and stationary configuration.

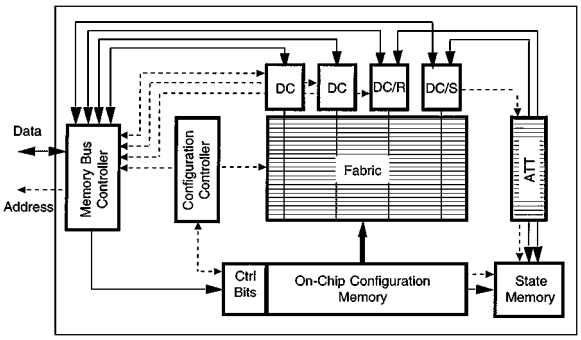


Figure 10. Architecture overview. Sold lines are data paths, dashed lines are address and control paths.

fact that there are six physical stripes in the fabric. 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 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 Fig. 10, 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 to any input stripe, get results from any output stripe, and to save and restore the stripe state when it is removed or inserted from the FPGA fabric. The stripe state 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 the "bursty" memory traffic that is caused by virtualizing the application. 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 stripe 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 registers on that stripe. 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 Fig. 9) with one entry per physical stripe.

4. 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.

We assume the interface between the FPGA and the CPU host resembles a typical slave co-processor, like a floating-point unit. 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 5.

4.1. 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 fabric), scheduling (time-multiplexing and managing virtualization), and managing the on-chip configuration memory.

The controller manages the interface between the CPU host and the fabric. At the very least the interface must allow the host to initiate execution of a particular configuration, and allow the FPGA co-processor to indicate that it has completed execution. If the configuration information is stored in main memory, it is possible to specify the application by giving 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. The co-processor could signal the completion of the application through an interrupt or a status register that is polled by the CPU.

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 11 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 Fig. 11 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 stalls. 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.

4.2. 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.

The CPU initiates the execution of an application on PipeRench by loading a set of control registers with the starting address of the executable in main memory, and the number of iterations to perform using this executable.

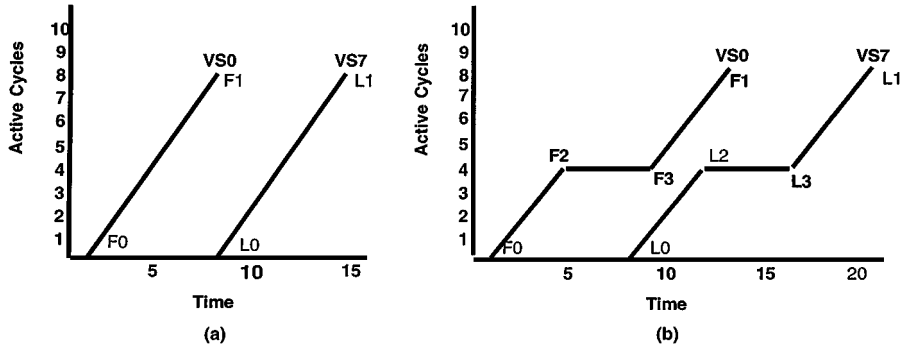


Figure 11. Active cycles example: Variation of the active cycles with time for (a) the non-virtualized and (b) the virtualized case. (a) Shows the case for 8 virtual stripes on 8 physical stripes while (b) shows the case for 8 virtual stripes on 5 physical stripes. The two curves represent the first and the last virtual stripes (VS0 and VS7).

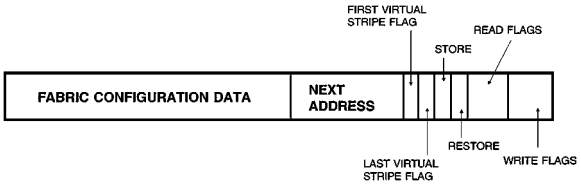


Figure 12. Configuration word. The structure of a configuration word consisting of the configuration data that goes to the fabric, the next address field, and a set of flags. The flags comprise indicators for the first and the last virtual stripes, and other fields described in Section 5.

In PipeRench, an "executable" is composed of a series of configuration words 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 Fig. 12). The flags relevant to 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 Fig. 13. When the *IDLE* line is asserted, the host can start a new application by specifying a start address and the number of iterations. The controller then deasserts the *IDLE* line until the application has completed the number of iterations specified.

4.2.1. 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), shown in Fig. 13. Fortunately, the number of entries in the SATT is small compared to the size of the application, therefore it will not be on the critical path.

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.

4.2.2. 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.

4.2.3. Tracking the 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

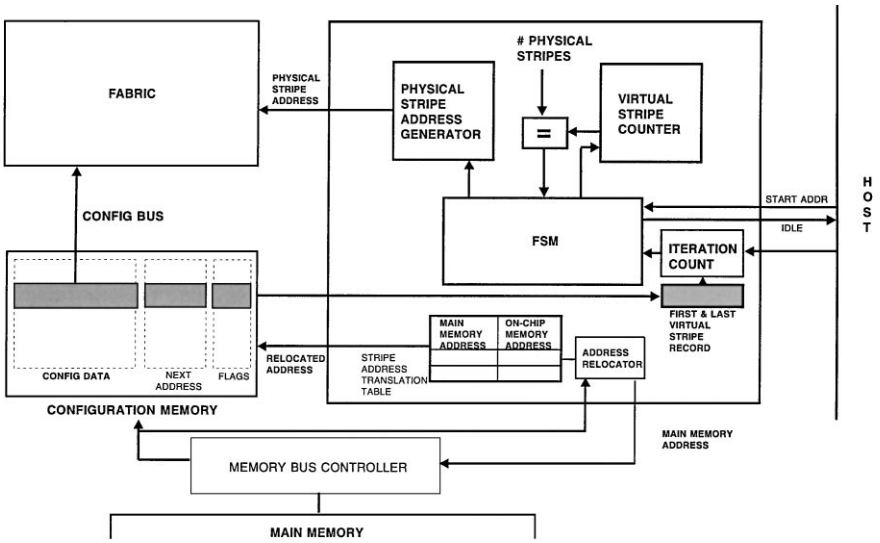


Figure 13. Configuration controller architecture: The configuration controller, and its interface to the host, main memory, on-chip memory, and the fabric.

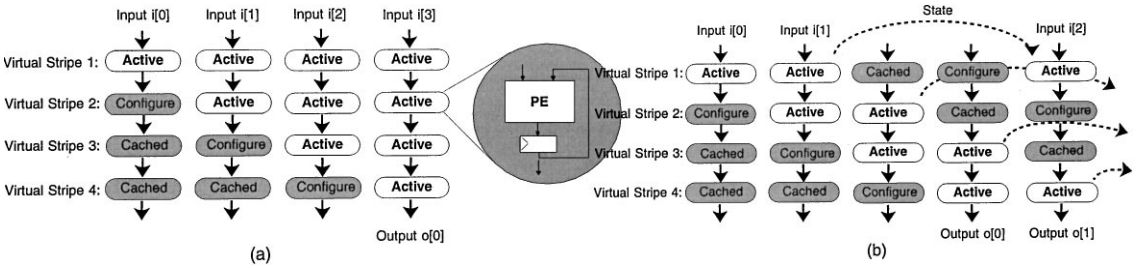


Figure 14. Virtualization I/O: Comparing input/output and state management with no virtualization and virtualization. (a) With enough hardware (no virtualization) there is no need to save state and input/output timing remain unchanged, (b) with less than enough hardware (virtualization) a stripe's state must be saved and input/output timing changed.

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 records the cycle 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.

4.3. Summary

In this section, we analyzed and described the four main sub-tasks 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.

5. Data Management

Managing the flow of data for virtualized pipelines 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 design goal in PipeRench's to make these disruptions transparent to the designer. This section presents our data controller architecture and shows how it manages the virtualization of a convolution kernel.

When there is no virtualization, there is no need to store and restore state or change input/output timing. Figure 14(a) shows the execution of a simple pipeline 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, inputs and outputs are needed every cycle since the stripes that need input and output remain in the fabric.

However, when such pipelines are virtualized, the stripe state may need to be remembered and the input/output timing changed. Figure 14(b) shows the execution of the same pipeline, 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 Fig. 14, input (output) is only needed when the first (last) stripe is in the fabric.

5.1. Data Controller Architecture

The data controller architecture consists of four separate data controllers (see Fig. 9). Each controller manages one global bus that is dedicated to either state storing, state restoring, data input or data output per application. When dedicated to storing or restoring state, the data controller interfaces between the fabric and the state memory. When a controller is dedicated to

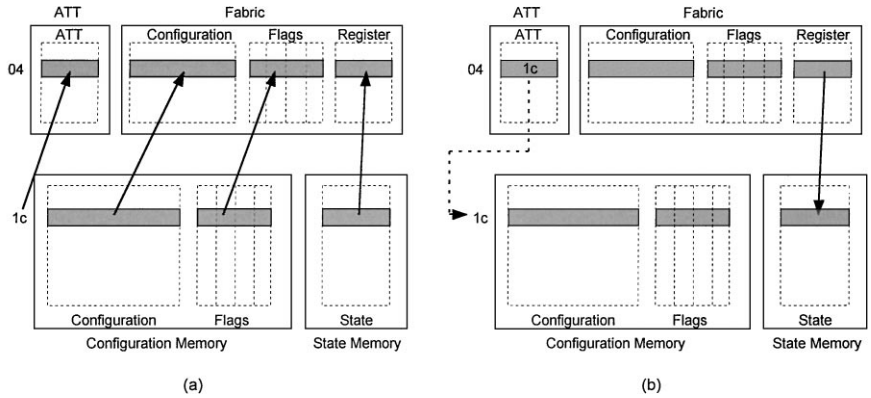


Figure 15. Store/Restore. Restoring (a) and storing (b) state between fabric, configuration memory, state memory and ATT.

data IO, 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.

5.1.1. Managing Stripe State. When needed, a stripe's state is kept in the state memory (see Fig. 16), 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 Fig. 15(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 Fig. 15(b).

5.1.2. Managing Data IO. 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 Fig. 12). The flag bits are part of each stripe's control word and specify if that stripe reads or writes to each of the 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 Fig. 16). Therefore, when a stripe is configured to produce data on a bus, the controller

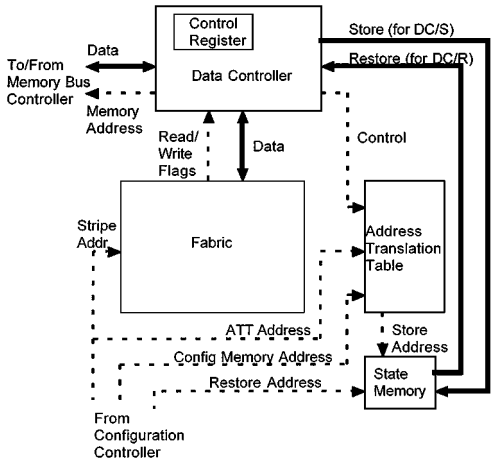


Figure 16. Data controller architecture. Solid lines are data and dashed lines are address or control.

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.

5.2. Example: Convolution Data Flow

To make the function of the data controllers more concrete, we now illustrate how the application presented

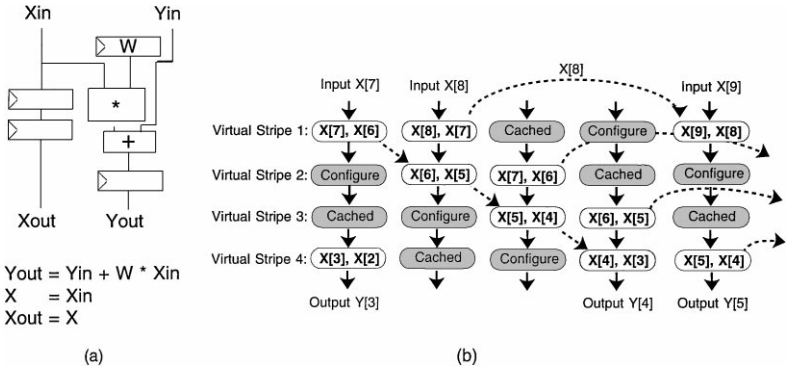


Figure 17. Systolic convolution. (a) Each stage's function contains a double pipelined X input, single pipelined Y output, and stationary weight W, (b) example of the data flow for this implementation. The dashed lines indicate how Y is accumulated as time progresses. The dashed arcs indicate state store and restore.

in Fig. 1 is virtualized on PipeRench. In the terminology presented in Kung [1], this is a strictly systolic implementation with the *X* input stream doubly pipelined. In [14], we also present a semi-systolic implementation of this same application, and illustrate the significant difficulties caused by an operand broad-cast over multiple stripes. For this reason, PipeRench current only supports strictly systolic implementations.

Figure 17 shows a fully systolic implementation of the application, which contains a single pipelined output Y, a double pipelined input X, and stationary weight W. In this example, we will assume that the functionality for one tap of this convolution can be supplied by one stripe. The X's enter the pipeline from the first stage. Every cycle a new X with a higher index is 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.

5.3. Summary

Data management should be transparent to applications no matter how many physical stripes are present and virtual stripes are needed. Our data controller architecture handles this transparency with communication between the stripes in the fabric and the data controllers. Through several flags in the control word of each stripe, the data controllers can tell what is needed by the fabric and the status of execution.

6. 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 100 MHz. An SRAM for configuration memory will consume another 50 mm^2 of area, and will store 256 configuration words of 768 bits each. One 128-bit wide stripe is capable of holding one tap of a 8-bit FIR filter with 12-bit coefficients. The total area for this chip would be 100 mm^2.

As shown in Fig. 18 the virtualization 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.

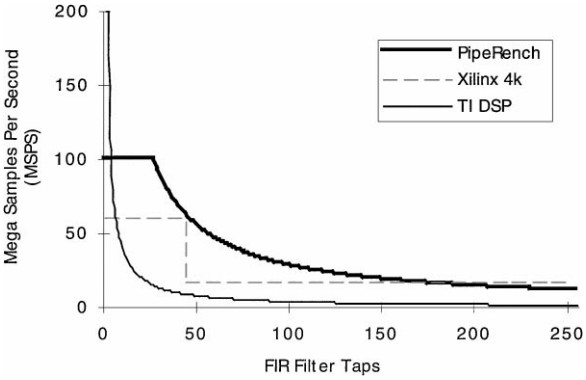


Figure 18. Performance on 8-bit FIR filters. PipeRench, Xilinx FPGA using parallel and serial arithmetic and Texas instruments DSP.

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 [15], 100 mm^2 of area is equivalent to about 1750 CLBs. Given this amount of logic, and using parallel distributed arithmetic, it is possible to create filters that run at around 60 MHz and have up to 48 taps [16]. More than 48 taps will not fit. No widely known techniques can increase the throughput of this implementation. To implement larger filters, it is necessary to transform the algorithm to use a more efficient arithmetic. Using double-rate distributed arithmetic, it is possible to construct filters with up to 260 taps given the same amount of silicon [16]. Due to the serial nature of these implementations however, the maximum sampling rate of these filters is 14 MHz. There is a larger space of filters that are unfulfilled by this solution. The discontinuities in this graph make it difficult to compute the cost/performance of the device. In addition, the discontinuities represent a significant redesign effort. The two types of arithmetic used in this case require complete new run through the synthesis and physical design tools.

The Texas Instruments TMS320C6201 [17] is a commercial DSP which runs at 200 MHz and contains two 16- by 16-bit integer multipliers. For filters with less than four taps, the high clock speed of this device yields the highest possible performance. This performance decays rapidly with an increasing number of taps due to the presence of only two multipliers. PipeRench exhibits a very similar curve to this DSP, only the capacity of the device is significantly higher. PipeRench can hold about 29 taps until the hardware is time-multiplexed. Due to the word-oriented functional units in PipeRench, the maximum clock rate is significantly higher than the FPGA. And like the DSP, the degradation in performance has no large discontinuities, and requires no re-design to adapt to less hardware.

7. Related 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 [18] 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 [9–11, 19], 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, and allow the active fabric to change very rapidly. They can be used to virtualize hardware, but because they cannot be incrementally reconfigured they suffer from the pipeline fill and empty penalty. 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 simplifies the compilation process by allowing the compiler to create a pipeline of unbounded length. The only real design constraint is making the individual pipeline stages fit into PipeRench stripes.

A form of pipeline reconfiguration for commercial FPGAs, such as the XC6200, has been described [13]. The XC6200 cannot be reconfigured at the same rate as the data flow, and it is therefore necessary to segment the pipeline. PipeRench has the configuration

bandwidth necessary to support pipeline reconfiguration, and includes mechanism for control of the configuration stream and data stream with respect to the virtualization.

Other devices are capable of partial, run-time reconfiguration, such as GARP [20], NAPA [21], and Chimeara [22], and could potentially operate using pipeline reconfiguration. Exploration of the interface between FPGAs and CPUs has been investigated in [20–24].

PipeRench also addresses many of the problems faced by other computer architectures. The most-insightful comparisons are to MMX, VLIW, and vector machines.

The mismatch between application data size and native operating data size has been addressed by extending the ISAs of microprocessors to allow a wide data path to be split into multiple parallel data paths, as in Intel's MMX [25]. Obtaining SIMD parallelism to utilize the parallel data paths is non-trivial, and works only for very regular computations where the cost of data alignment does not overwhelm the gain in parallelism. PipeRench has a rich interconnect to provide for alignment and allows PEs to have different configurations so that parallelism need not be strictly SIMD.

VLIW architectures are designed to exploit dataflow parallelism that can be determined at compile time [26]. VLIWs have extremely high instruction bandwidth demands. A single PipeRench stripe is similar to a VLIW processor using many small, simple functional units. In PipeRench, however, a stripe remains configured for a number of cycles, and the same computation is performed on a larger data set, thereby amortizing the instructions over more data.

The instruction bandwidth issue has been addressed by vector machines such as the T0 [27] and IRAM [28]. In many ways, PipeRench is similar to vector machines with an unbounded vector size and with VLIW functional units. The problem with classical vector architectures is the vector register file is a physical or logical bottleneck that limits scalability. Allocating additional functional units in a vector processor requires additional ports on the vector register file. The physical bottleneck of the register file can be ameliorated by providing direct forwarding paths to allow chained operations to bypass the register file, as in the Cray-1 [29]. PipeRench eliminates these constraints by eliminating the vector register file. All connections in PipeRench are local, and the chaining is explicit. Therefore, the number of functional units can grow without increasing the complexity of the issue and control hardware.

8. Conclusions

Pipeline-reconfigurable FPGAs provide the high-performance associated with FPGAs for DSP applications. In addition, they provide the forward-compatibility and robust compilation that is associated with more traditional processors. 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.

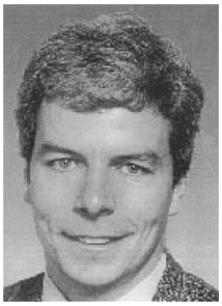
Acknowledgments

We would like to acknowledge the efforts of former members of the PipeRench research group: Jeffry Weener, Kevin Jaget, Matthew Myers and Ronald Laufer. We thank the anonymous reviewers for their helpful feedback.

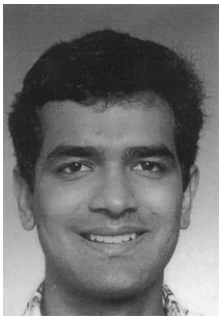
References

1. H.T. Kung, "Why Systolic Architectures?" *IEEE Computer*, Piscataway, NJ, 1982, pp. 37–45.
2. R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Structures*, MIT Press, 1996.
3. D.I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Transactions on Computers*, vol. C-31, no. 11, 1982, pp. 1121–1126.
4. D.I. Moldovan, "On the Design of Algorithms for VLSI Systolics Arrays," *Proceedings of the IEEE*, vol. 71, no. 1, 1983, pp. 113–120.
5. B. Von Herzen, "Signal Processing at 250 mhz Using High-Performance FPGAs," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1997, pp. 62–68.
6. Xilinx, *XC4000 Series Field Programmable Gate Arrays*, revision 1.04, Sept. 1996.
7. Altera, Data book, 1998.
8. Xilinx, *XC6200 Field Programmable Gate Arrays*, revision 1.7, Oct. 1996.
9. A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (Eds.), Napa, CA, April 1994, pp. 31–39.

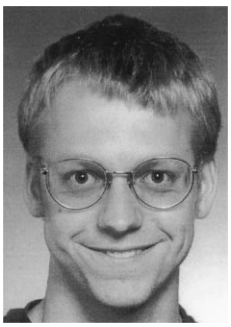
10. S. Scalera and J.R. Vazquez, "Design and Implementation of a Context Switching FPGA," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K.L. Pocek (Eds.), Napa, CA, April 1998, pp. 78–85.
11. S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K.L. Pocek (Eds.), Napa, CA, April 1997, pp. 22–28.
12. M.J. Wirthlin and B.L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1996, pp. 122–128.
13. W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung, "Pipeline Morphing and Virtual Pipelines," *Field-Programmable Logic and Applications*, P.Y.K. Cheung, W. Luk, and M. Glesner (Eds.), London, England, Sept. 1997, pp. 111–120.
14. S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas, "Managing Pipeline-Reconfigurable FPGAs," *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998, pp. 55–64.
15. J. Rose, Private communications, 1997.
16. S. Knapp, Using Programmable Logic to Accelerate DSP Functions, 1995. http://www.xilinx.com/appnotes/dspintro.pdf.
17. Texas Instruments, TMS320C6201 digital signal processor, revision 2, 1997.
18. J. Babb, R. Tessier, and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (Eds.), Napa, CA, April 1993, pp. 142–151.
19. N. Bhat, K. Chaudhary, and E.S. Kuh, Performance-oriented fully routable dynamic architecture for a field programmable logic device. M93/42, U.C. Berkeley, 1993.
20. J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor With a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 24–33.
21. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998, pp. 28–37.
22. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 87–96.
23. R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *MICRO-27*, November 1994, pp. 172–180.
24. R. Witting and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.
25. A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, 1997, pp. 24–38.
26. R.P. Colwell, R.P. Nix, J.J. O'Donenell, D.B. Papworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of ASPLOS-II*, March 1987, pp. 180–192.
27. J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan, "Spert-II: A Vector Microprocessor System," *IEEE Computer*, vol. 29, no. 3, March 1996, pp. 79–86.
28. C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer*, 1997, pp. 75–78.
29. R.M. Russell, "The CRAY-1 Processor System," *Communications of the ACM*, vol. 21, no. 1, 1978, pp. 63–72.



Herman H. Schmit is an assistant professor of Electrical and Computer Engineering at Carnegie Mellon University. He received his Masters and Ph.D. from Carnegie Mellon in 1992 and 1995, respectively. His research interests include reconfigurable computing, low power digital design and IP-based design.
herman@ece.cmu.edu

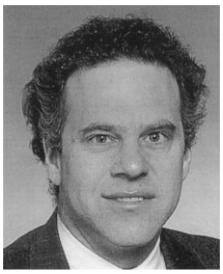


Srihari Cadambi is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at Carnegie Mellon University. He received his Masters from the University of Massachusetts, Amherst, in 1996. His research focus is compiler optimizations for reconfigurable architectures.
cadambi@ece.cmu.edu



Matthew Moe is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at Carnegie Mellon University. He received his B.S. and M.S. degrees from Carnegie Mellon in 1996

and 1998, respectively. His research focus is on design optimization and scalability of reconfigurable computing architectures.
moe@ece.cmu.edu



Seth Copen Goldstein is an assistant Professor in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. from the University of California at Berkeley in 1997. Before attending UC-Berkeley he was the founder and CEO of Complete Computer Corporation which developed Complete C, a cross platform object-oriented toolkit. His current research interests focus on architectures and compilers for reconfigurable computing systems.
seth@cs.cmu.edu