Incremental Reconfiguration for Pipelined Applications

Herman Schmit
Dept. of ECE, Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper examines the implementation of pipelined applications using run-time reconfiguration. Throughput and latency of pipelined applications can be significantly improved when reconfiguration is performed at the level of individual pipeline stages, as opposed to configuration of the entire FPGA. If reconfiguration and execution can be performed simultaneously, the performance of a pipelined application approaches its theoretical maximum. This paper proposes a new FPGA configuration mechanism, called striping, that supports pipeline stage reconfiguration and simultaneous configuration and execution. Additionally, the use of the pipeline stage as the atomic unit of reconfiguration introduces a design abstraction that enables the development families of upwardly-compatible FPGAs and virtual hardware design.

1.0 Introduction

Run-time reconfiguration (RTR) has the potential to greatly increase the performance and applicability of FPGA-based computing [3]. There are two significant problems inhibiting the deployment of applications based on RTR. First, design methodologies for partially reconfigured applications, or local RTR, are completely *ad-hoc*. Second, existing FPGAs lack reconfiguration mechanisms that adequately support local RTR.

Many of the computationally intensive tasks typically targeted by FPGA accelerators have certain regular data flow patterns that allow the creation of pipelined implementations. Historically, the creation of a pipelined implementation has served one purpose: increasing parallelism and thereby increasing throughput or decreasing power.

This paper proposes the pipeline stage as the basic unit of reconfiguration. This approach significantly reduces the latency and increases the throughput of the implementation, while minimizing the amount of required external storage. Additionally, the use of the pipeline stage as the atomic unit of reconfiguration introduces a design abstraction that enables the development of upwardly-compatible FPGA families. Upwardly-compatible FPGAs will provide users with the security of knowing that their existing applications will still function on future generations of the FPGA family, at a higher level of performance.

A new FPGA architecture is being developed which explicitly supports incremental pipeline reconfiguration. This FPGA, called a striped FPGA, differs from available FPGAs in two ways. First, this FPGA is reconfigured at a granularity that corresponds to the chosen basic unit of reconfiguration, the pipeline stage. An on-chip configuration cache allows pipeline stages to be loaded into the FPGA cells at a very high speed. Second, the a striped FPGA has interconnect especially suited for the implementation of pipelined applications.

The next section describes the configuration mechanisms used by existing FPGAs and the details of striped reconfiguration. Section 3.0 describes two strategies for implementing pipelined computations using RTR, and also describes how well the different configuration mechanisms perform using these implementation strategies. The final sections of this paper describe configuration control of a striped FPGA, the proposed cell architecture of our striped FPGA, and some targeted applications for this architecture.

2.0 FPGA Configuration Mechanisms

In this section, we describe the three existing mechanisms that are currently used for configuring FPGAs. We then propose a fourth mechanism, called *striped configuration*, which combines the advantages of the three existing mechanisms and is especially suited to incremental reconfiguration for pipelined applications.

2.1 Standard FPGA Configuration

Popular commercial FPGAs such as the Xilinx 4000 family and the Altera FLEX family 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. 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 in Section 3.0, this long configuration time hurts throughput, latency and memory requirement for pipelined application.

2.2 On-line Configuration

On-line configurable FPGAs have the configuration storage of the FPGA accessible and writable during execution mode. The most recent of this class of FPGAs is the Xilinx 6200 family [7], which maps the entire configuration storage to a region of address space that can be written by a connected microprocessor. Reconfiguration can take place concurrently with execution, and partial reconfiguration is possible. The atomic unit of reconfiguration in on-line reconfigurable chips such as the Xilinx 6200 is fairly small. Any one functional block can be independently reconfigured at any one time.

The reconfiguration speed of these chips is still limited by a relatively narrow interface to the configuration memory. The Xilinx 6200 has an 32-bit configuration data bus. All new configurations must be loaded from off-chip storage, under the control of other hardware such as a microprocessor. When used to implement pipelined applications, these FPGAs require significant data storage, or require extensive routing resources, as will be described in Section 3.0.

2.3 Multiple-context Configuration

A third type of configuration mechanism is the multiple-context configuration, as proposed in [2] and [5]. 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 this mechanism solves configuration speed problem, it does have limitations. First, the unused configuration information must be stored in many small, area-inefficient memories distributed all over the chip. The size of a multiple-context cell is therefore significantly larger than the size of a standard FPGA cell. 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. And, as we shall show in Section 3.0, FPGAs that use multiple contexts may require significant data storage to implement pipelined applications.

2.4 Striped Configuration

Striped configuration combines many of the advantages of on-line configuration and multiple-context configuration, and has some additional qualities that are particularly suited towards implementation of pipelined applications.

The most important idea of striped configuration is that the atomic unit of reconfiguration of the FPGA has been chosen so that it matches a sensible level of reconfiguration within the application it is performing. A pipelined application can be easily decomposed into pipeline stages. The striped FPGA is separated into a set of stripes, and in an ideal scenario, each of the application's stages would fit into one of the FPGA's stripes.

An FPGA stripe may be configured in one configuration cycle from the data stored in a wide, on-chip configuration cache, as illustrated in Figure 1. This provides high-speed reconfiguration, and allows the unused configuration information to be stored in a single large on-chip RAM, with much higher densities possible than in a multiple-context FPGA. Modification of the configuration cache can take place concurrently with execution, so there is no hard limit to the amount of virtual hardware that can be emulated.

The second basic idea of striped configuration is that the configuration information within one stripe can move to a neighboring stripe within the FPGA. This feature is particularly useful for pipelined applications. In addition,

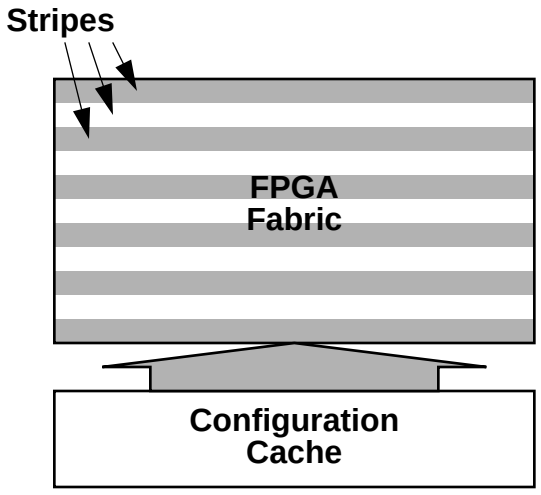

Figure 1. Stripe Configured FPGA

this feature avoids the problem of requiring the wide configuration bus to run over the entire FPGA fabric, consuming valuable interconnect resources and incurring extra delay. Instead, the configuration bus comes from the configuration cache and loads the lowest stripe in the FPGA fabric. Configuration information from that stripe can shift upwards into other stripes in the fabric.

An FPGA using striped configuration is well suited to implementing a variety of pipelined applications. In the next section we will quantify the performance advantages of striped configuration compared to the other three configuration mechanisms. In Section 4.0 we will discuss other advantage of striped configuration, the most significant of which is upward compatibility. An application that has been designed for a striped FPGA can be run on future versions of a compatible FPGA at significantly higher levels of performance. Finally, in Section 5.0 we describe the functionality of the FPGA fabric we plan to implement in a striped FPGA, and in Section 6.0 we describe some example pipelined applications.

3.0 Implementing Pipelined Applications using RTR

The application we will examine is a very deeply pipelined application, such as a high-order FIR filter. Assume that this application has *S* identically-sized pipeline stages. Further assume that there are *K* bytes of data flowing between each stage of the filter every cycle, and between the filter input and output. This assumption, that there is a constant amount of data flowing between each stage in each cycle rarely holds in real pipelined applications. The assumption is made to ease the following analysis. None of the following methods rely on constant data flow between stages.

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 Figure 2. If the execution of a pipeline stage takes *T*, then the throughput of the static, N-FPGA implementation of this filter is K/T bytes per second.

We will use RTR to implement this filter in one FPGA of similar capacity. The theoretical maximum throughput of the 1-FPGA implementation of this filter using RTR is $K/(NT)$ simply due to the reduction in computing hardware. We will examine the implement this filter using component-level reconfiguration and partial reconfiguration, and examine the performance characteristics and memory requirements of each implementation technique.

3.1 Component-level Reconfiguration

Using component-level reconfiguration, the *N* different FPGA configurations from the *N*-FPGA design are used to configure a single FPGA. This level of reconfiguration has also been called Global RTR in [3]. The configuration controller loads one configuration, and allows the FPGA to perform operations on X pieces 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:

$$T(X-1+S/N) \qquad \text{(EQ 1)}$$

After this computation is complete, the system controller reconfigures the FPGA with the next configuration in the sequence as illustrated in Figure 3. If it takes *C*

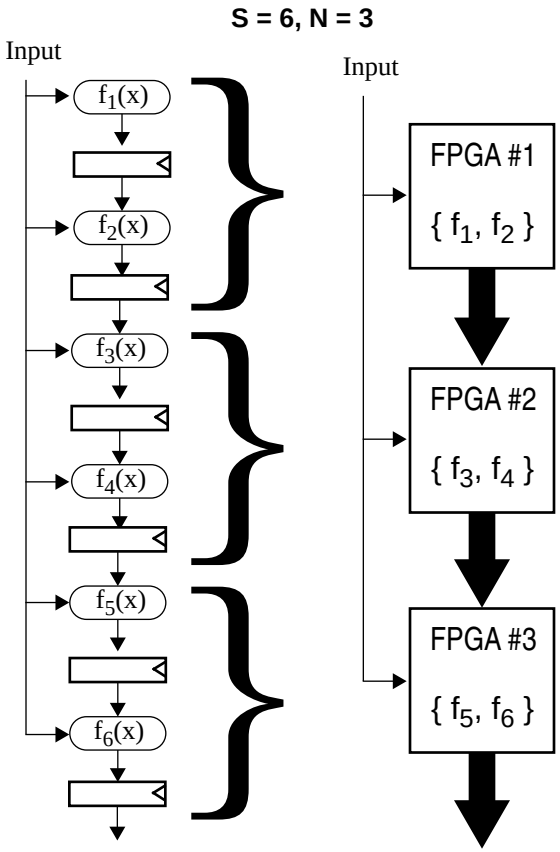

Figure 2. Example Pipelined Application

cycles to reconfigure the FPGA, then the throughput of this implementation can be described using the formula:

$$Throughput = \frac{KX}{NT\left(X + \frac{S}{N} - 1 + C\right)} = \frac{K}{T\left(N + \frac{S-N}{X} + \frac{NC}{X}\right)} \quad \text{(EQ 2)}$$

Throughput falls short of the ideal based on two factor. The first factor is the penalty suffered for having to repeatedly fill and empty the pipeline between reconfigurations. This factor is reflected in the $(S-N)/X$ term in (EQ 2). The second factor is the penalty for reconfiguration, which is reflected in the $(NC)/X$ term. Both of these terms can be reduced by increasing X. The relationship of *C* and *X* on throughput is shown in Figure 4 for a pipelined application. In this graph, *S* = 100 and *N* = 10. The ideal performance of this implementation is $K/(10T)$. The value on the *y*-axis indicates how actual throughput compares to this ideal. When C is small, this graph shows the throughput degradation due to the pipeline penalty. When C is large, the pipeline penalty becomes relatively unimportant because the throughput degradation due to configuration time is so large.

It would seem that increasing *X* is the best way to increase the throughput of implementation. The first problem with arbitrarily increasing *X* is that the latency of the implementation increases. The best-case latency for this implementation is:

$$Latency = NT\left(X + \frac{S}{N} - 1 + C\right) \quad \text{(EQ 3)}$$

Worst-case latency for this implementation can be as high as two times the best-case latency, because new input data might arrive just after the first reconfiguration has been replaced by the second reconfiguration, in which case that data will have to wait in a buffer for this entire sequence of configurations to conclude before its computation begins.

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:

$$Memory = KX \quad \text{(EQ 4)}$$

If *X* is very large, it will be difficult to fit the entire memory 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.

Any type of FPGA can be reconfigured using component-level reconfiguration. It is the only mode possible for standard FPGAs, as well as multiple-context FPGAs. Standard FPGAs have a very large value of *C*. On-line reconfigurable FPGAs can be reconfigured on a component-level basis by simply halting execution until the entire FPGA fabric has been reconfigured.

Table 1 shows typical values of *C* for two available Xilinx components using the fastest configuration mode

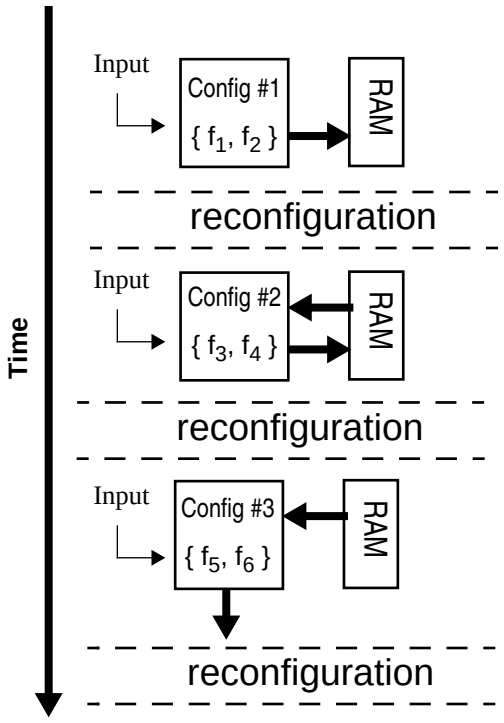

Figure 3. Component-level Reconfiguration

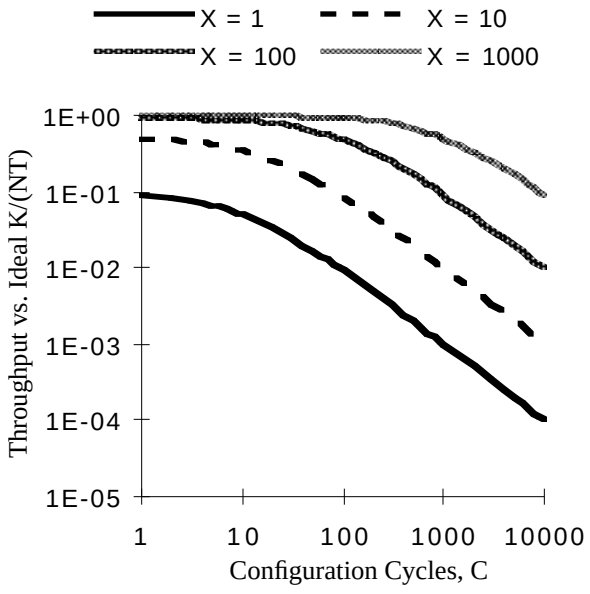

Figure 4. Throughput vs. C and X

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, or latency and memory requirements for applications which use these components.

Part	*Config. Time*	*C*	Reference
XC4028EX	8.35 ms	275,000	[8]
XC6216	92 µs	3036	[7]

Table 1. Commercial FPGA Configuration Times

Multiple-context FPGAs have a *C* value 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 can only be effective if $N \le Contexts$.

3.2 Incremental Pipeline Reconfiguration

Incremental pipeline reconfiguration is a restricted form of local RTR [3], in which the pipeline is separated into *S* components, each corresponding to one pipeline stage. The FPGA can hold S/N of these pipeline stages, and the reconfiguration happens in an incremental manner. During each stage of the computation, we add one additional stage to the configuration, and remove a stage if necessary to keep the amount of configuration within the capacity of the FPGA. Figure 5 illustrates this procedure. Reconfiguration in this manner can be visualized as the scrolling of a window through the computation.

Assume that the time it takes to execute a pipeline stage is *T*, and the number of execution cycles required to reconfigure this entire FPGA is *C*. Therefore, the time required to substitute a new pipeline stage into the configuration is, ideally, $TC(N/S)$. Because we have to configure S pipeline stages, the total reconfiguration time for one sweep through the application will be *TCN*. Execution of the entire pipeline will take *S* cycles for the first element of data, and $(S/N) - 1$ cycles to get the remaining data out of

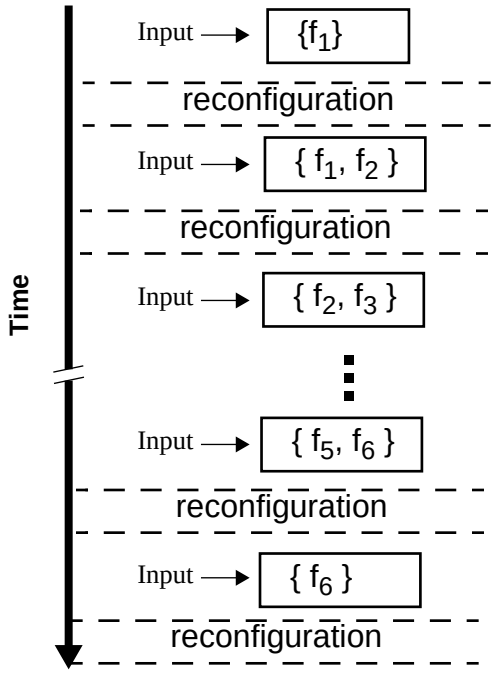

Figure 5. Pipeline Stage Reconfiguration

the configuration. Therefore, the best-case latency and throughput of this implementation can be described with the following expressions:

$$Latency = T\left(S + \frac{S}{N} - 1 + CN\right) \quad \text{(EQ 5)}$$

$$Throughput = \frac{K\frac{S}{N}}{T\left(S + \frac{S}{N} - 1 + CN\right)} = \frac{K}{T\left(N + 1 - \frac{N}{S} + \frac{N^2}{S}C\right)} \quad \text{(EQ 6)}$$

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 best-case latency for the same three implementations. These graphs show that when *C* is small, the incrementally reconfigured implementation exhibits both high throughput and low latency. When *C* is large, the throughput exhibited by the incrementally reconfigured design exhibits behavior very

similar to the component-level reconfigured implementation with $X \approx S/N$.

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 incrementally reconfigured FPGA approach their respective theoretical maximums. 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.

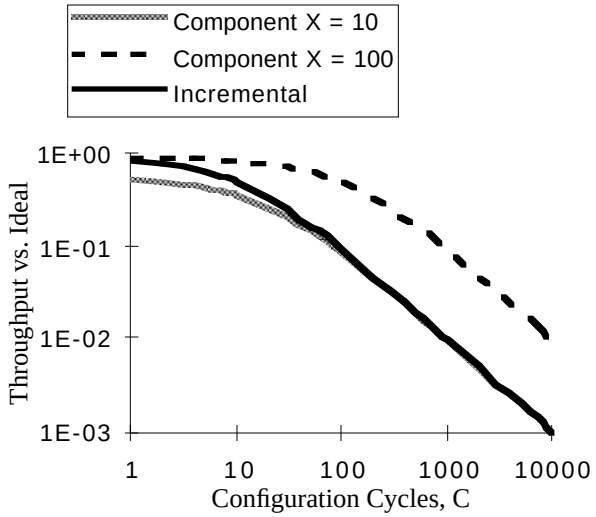

Figure 6. Throughput of Incremental and Component-level Reconfiguration

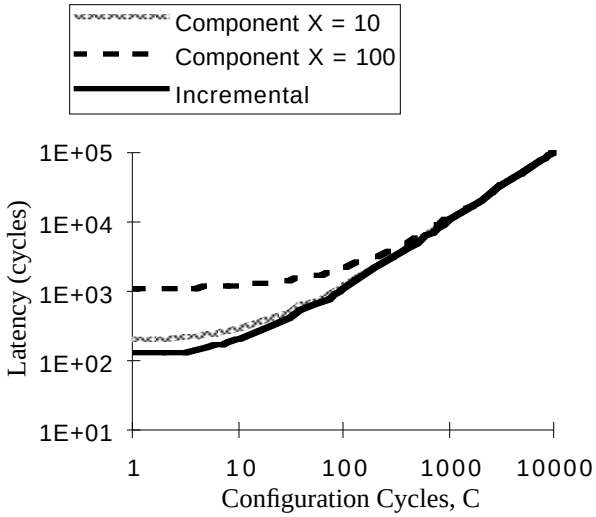

Figure 7. Latency of Incremental and Component-level Reconfiguration

The most important characteristic of incremental pipeline reconfiguration is that the presence of more hardware transparently results in higher throughput. Figure 6 shows how throughput increases as the amount of real hardware in the FPGA approaches the amount of virtual hardware in the pipelined application (as *N* approaches one.) The non-linearity of this graph is due to the fact that each run through the pipelined application, the FPGA has to be completely emptied before it can begin another sweep through the application. As a result, when the amount of real hardware is just slightly less than the amount of virtual hardware (*N* is slightly greater than one), throughput is half of the maximum. When the amount of real hardware equals or exceeds the amount of virtual hardware, the pipeline no longer has to be repeatedly emptied and filled, resulting in a large performance boost. If an application could begin execution at the top of pipeline again while the bottom of the pipeline was emptying, this graph would be linear. This scenario is possible whenever the bottom and top of the pipeline do not share any global resources, like data busses or memories.

Only on-line reconfigurable FPGAs and striped FPGAs are capable of incremental pipeline reconfiguration. There are two problems with using pipeline reconfiguration on an on-line reconfigurable FPGA like the XC6200 series. First, 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. Second, there is a problem caused by interconnecting the substituted pipeline stages in on-line reconfigurable FPGAs. At the beginning of execution, the pipeline stages are placed in previously empty locations in the FPGA. After the FPGA

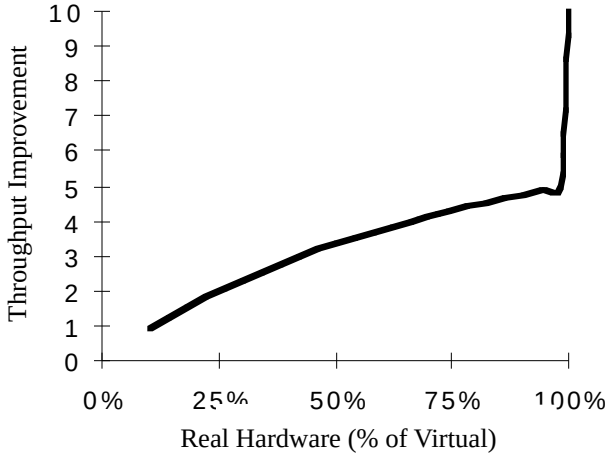

Figure 8. Throughput of Incremental and Component-level Reconfiguration

fills up, the oldest pipeline stage is replaced with the next stage. Sometimes the location of the old stripe is far from where the newest stripe would prefer to be, as happens when stage f_5 replaces stage f_1 in Figure 9a. In order to connect stage f_4 to stage f_5 in this example, the FPGA would have to have significant global interconnect resources. Also, the delay of this data transfer across the whole chip might significantly increase *T*.

The striped FPGA solves these problems by integrating an on-chip configuration cache and by shifting configuration data within the FPGA fabric. As illustrated in Figure 9b, when a new pipeline stage is placed in the fabric, it can always be placed adjacent to the stage previous to it in the fabric. Old pipeline stages are removed by shifting them off the top of the FPGA fabric.

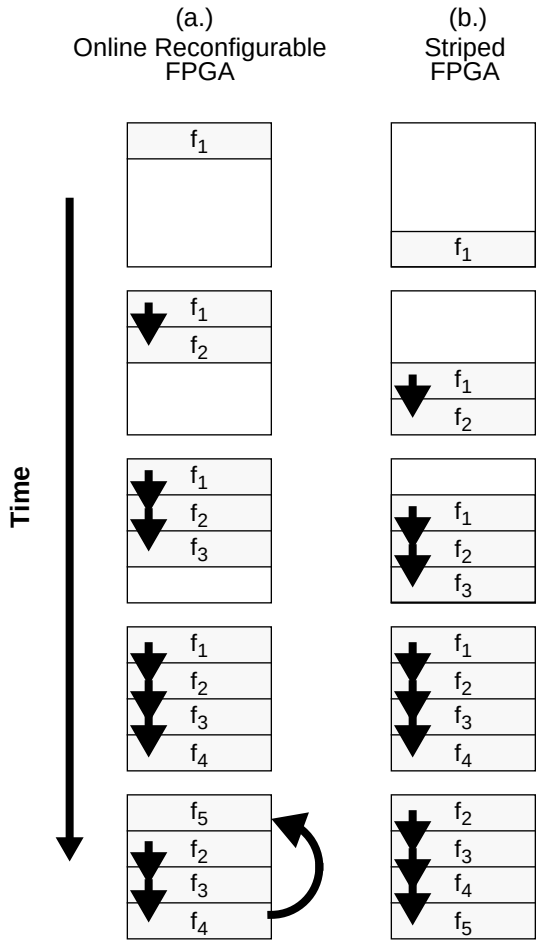

Figure 9. Pipeline Stage Placement and Routing

4.0 Striped Configuration Control and Design

A significant advantage of striped configuration is that the increased size of the atomic unit of reconfiguration eases problems of configuration control and algorithm design for RTR. While the primary contents of the configuration cache are configuration bits for the FPGA fabric, it will also contain control information, such as the address in the cache of the next stripe to load, whether and when this stripe can be shifted off the top of the FPGA fabric, and whether this stripe can read or write the global data buses that route external data to all the stripes. The current control mechanism envisioned for a striped FPGA coprocessor is illustrated in Figure 10.

The first advantage of this control methodology is that it reduces the burden of reconfiguration control which will usually be placed upon the microprocessor. In a striped FPGA with this control mechanism, the microprocessor can input the first address of a pipelined application, which will initiate its execution, and then do perform some other task relevant to the current application.

The design methodology for striped FPGAs allows any pipeline to be broken up into a set of stripes which can be run on a compatible striped FPGA. The control mechanism can load the application into an FPGA with any number of physical stripes. Throughput will increase as the number of stripes increases, allow us to create applications which run on a variety of FPGAs at different performance levels. We could therefore have some FPGAs have more on-chip storage and fewer physical stripes, or vice versa. A design constructed for one chip could run on any of the other chips, albeit at a different performance level. In addition, older pipeline designs will still run on FPGAs built with newer, denser technology, at higher levels of perfor-

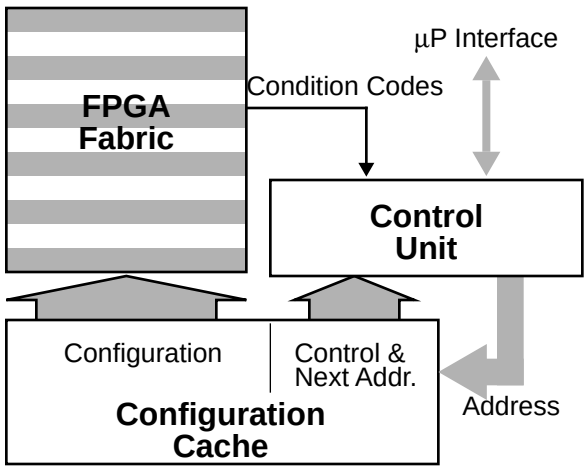

Figure 10. Stripe Control

mance. This upward-compatibility will allow the users to preserve their significant investment in the design of applications for FPGAs.

5.0 Proposed FPGA fabric

There are three significant constraints that the designer must deal with when designing a striped FPGA. First, the width of the stripe must match the width of the memory that stores the configuration information. Second, a stripe should be able to implement a non-trivial, pipelineable, portion of the application using a limited budget of configuration bits. Third, the interconnect between stripes must be very regular, because the configuration information for any logical stripe can be relocated to any physical stripe. In this section we will give brief summaries of our approach to these design constraints.

For the configuration cache, we could use an SRAM that has 768 configuration bits and is 6.8mm wide using 0.5µm technology. Since our applications are frequently based on small data elements, we have developed a cell that can function as a 4-bit ALU, a 4-bit MUX, or one of nine possible 4-input gates. A 4-bit register can be optionally connected to the output of this cell, and the carries for 4-bit ALUs can be cascaded to form bigger adders or subtracters. Our estimate for the width of these cells is 200 microns (including 50 microns for vertical interconnect) and it requires seven bits of configuration. If we make the width of a stripe match the width of the memory, we would be able to fit 32 of these cells in each stripe. The number of bits from the SRAM in 200 microns is 24, therefore there are 17 bits remaining for the configuration of the interconnect for each cell.

The interconnect architecture for two stripes is illustrated in Figure 11. Horizontal, or intra-stripe interconnect is used to connect between cells on the same stripe and the stripe immediately above. The vertical interconnect is global, and is primarily used to get data from the I/Os to some or all of the stripes in the fabric. Because the interconnect is either global or very local, it is possible to locate a stripe anywhere within the fabric, as long as it is adjacent to its pipeline predecessor and successor. The idea of relocatable hardware was discussed in [6]. The interconnect for this chip directly supports this concept.

The height of each cell is estimated at 300 microns, including about 150 microns for horizontal interconnect. If a 10mm by 6.8mm die is half allocated to on-chip configuration memory and half allocated to FPGA cells, we would have approximately 256 stripes of configuration in the cache, and 16 physical stripes in the fabric.

6.0 Example Applications

Our design effort is presently focussed on two specific applications, encryption using the IDEA cipher[4], and algorithms for SAR image recognition and identification. Both of these applications can be significantly accelerated by implementing a deep pipeline.

The IDEA cipher consists of eight rounds of the computation illustrated in Figure 12. The three operations in this computation are 16-bit addition, XORing, and multiplication modulus $2^{16}-1$. Obviously there is a great deal of parallelism in this computation, through both parallel execution and pipelining. The *Z* operands in this computations are portions of the key. By creating a different implementation for each key, we can propagate these constants and significantly reduce the hardware and delay of the multiplication operation. A key-specific configuration could be created before the communication, and sent over the public key encrypted channel. Alternately, there may be quick procedural methods of creating the multiplication blocks using the key as a parameter.

One of the most computationally intensive tasks in ATR target recognition algorithms is the shape sum computation [1]. This computation typically takes as input a 64 by 64 image with eight bits per pixel, and a 32 by 32 image template with one bit per pixel. In order to compute the shape sum, we place the template at some location within the input image and use it to mask the image, so that whenever a pixel in the template is zero, the corresponding pixel in the image is set to zero. The shape sum

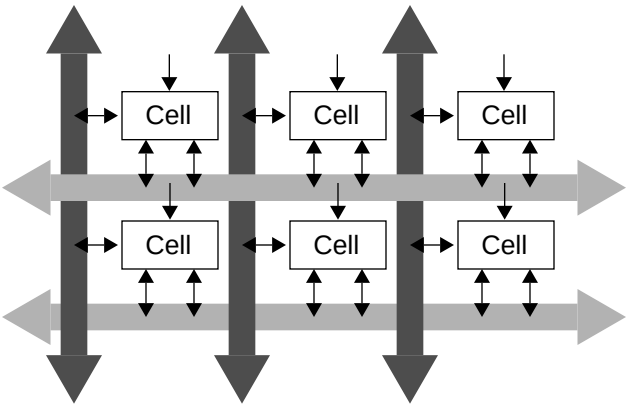

Figure 11. Basic Interconnect Architecture

is the average value of the pixels in the 8-bit image whose corresponding bit in the template is one.

The templates are sparse; only about 10% of the pixels are on. But for each image the shape sum must be computed for thousands of templates, and for all the possible displacements of the template in the input image. Because this computation must be performed at every displacement of the template, it is possible for us to create a pipelined implementation of this computation for each template. We will broadcast 128 bits of image data (16 pixels) over the entire fabric at each cycle. Such an implementation would have at least 64 stages. More stages may be required to do the division by a constant in order to arrive at an average pixel value. To further increase throughput, it is possible to create pipelines that simultaneously compute the shape sums for sets of similar templates.

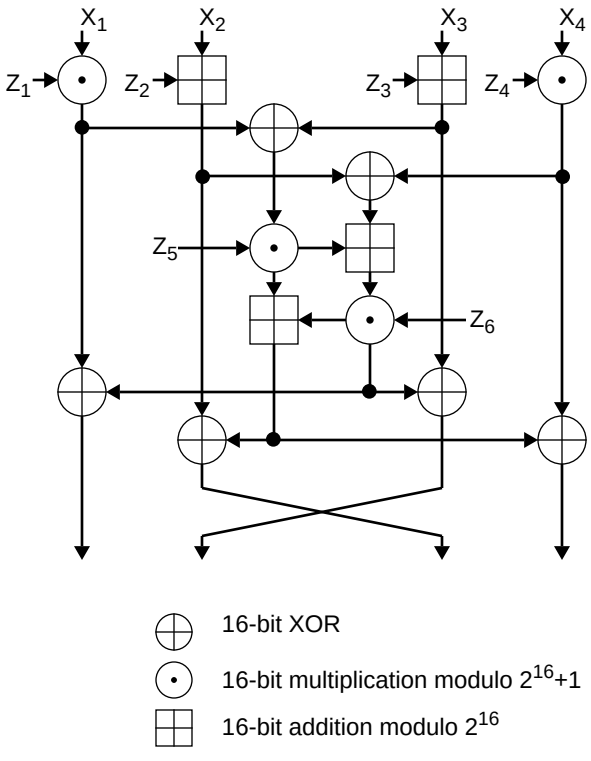

Figure 12. One round of the IDEA Cipher

7.0 Conclusions

The performance of pipelined applications on RTR based systems is highly dependent on the type of reconfiguration used. Reconfiguring at the granularity of pipeline stages can increase the throughput of an implementation, without significantly increasing the latency or required storage. If simultaneous reconfiguration and execution are possible, the performance is increased to the theoretical maximum of an RTR system. Striped FPGAs can reconfiguration at the pipeline level while simultaneously executing that pipeline. In addition, the use of the pipeline stage as the atomic unit of reconfiguration introduces a design abstraction that enables the development of CAD tools for RTR, and makes possible the creation of families or upwardly-compatible FPGAs.

8.0 Acknowledgments

The author wishes to thank DARPA (under contract DABT63-96-C-0083) for funding this research.

9.0 References

[1] B. Bray, Private communication, November, 1996.

[2] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," in *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31-39, April, 1994.

[3] B. L. Hutchings and M. J. Withlin, "Implementation Approaches for Reconfigurable Logic Applications," in *Field-Programmable Logic and Applications, FPL '95*, pp. 419-428, Oxford, 1995.

[4] B. Schneier, *Applied Cryptography*, Wiley, New York, 1996.

[5] S. Trimberger, D. Carberry, A. Johnson, J. Wong, "A Time-Multiplexed FPGA," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, April, 1997.

[6] M. J. Withlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, April, 1995.

[7] Xilinx, *XC6200 Field Programmable Gate Arrays*, Version 1.7, October, 1996.

[8] Xilinx, *XC4000 Series Field Programmable Gate Arrays*, Version 1.04, September 18, 1996.