Asynchronous PipeRench: Architecture and Performance Estimations

Hiroto Kagotani
Okayama University
Okayama City, Japan
kagotani@cne.okayama-u.ac.jp

Herman Schmit
Dept. of ECE
Carnegie Mellon University
Pittsburgh, PA 15213 USA
herman@ece.cmu.edu

Abstract - *PipeRench is a configurable architecture that has the unique ability to virtualize an application using dynamic reconfiguration. This paper investigates the potential benefits and costs of implementing this architecture using an asynchronous methodology. Since clock distribution and gating are relatively easy in the synchronous PipeRench, we focus on the benefit due to decreased timing pessimism in an asynchronous implementation. Two architectures for fully asynchronous implementation are considered. PE-based asynchronous implementation yields approximately 80% improvement in performance per stripe. This implementation, however, requires significant increases in configuration storage and wire count. A few particular features of the architecture, such as the crossbar interconnect structure within the stripe, are primarily responsible for this growth in configuration bits and wires. These features, however, are the primary aspects of the PipeRench architecture that make it a good compilation target.*

I. Introduction

Asynchronous logic has experienced a recent revival of interest in research and academia. The benefits of asynchronous logic, as discussed in the literature and summarized in [1], include:

- **Reduction in design effort:** Building a low-skew clock distribution network for a large ASIC or microprocessor is one of the most challenging aspects of the design. An asynchronous device has no global clock, and therefore eliminates this design effort.
- **Reduction in power dissipation for clocks:** In complex ASICs and microprocessors, the clock consumes a significant amount of the total power consumed by the device. The elimination of a global clock can reduce this power dissipation.
- **Reduction of timing pessimism:** Synchronous designs require the clock period to be greater than or equal to the worst case critical path, even though that path may be rarely enabled. As a result, for most cycles, the clock is slower than it needs to be. Asynchronous designs can, ideally, move from one computation to the next at the time exactly required by that first calculation.

This paper investigates the asynchronous implementation of the PipeRench architecture [2]. PipeRench is an architecture supporting virtualization of data paths for reconfigurable computing. In its current implementation, PipeRench [3] is implemented using standard synchronous methodology.

There are two independent clocks on the current implementation of PipeRench: one for the interface and one for the fabric (or data path). The interface clock connects to about one thousand registers located in a relatively small area on chip. It is not a challenge to distribute this clock. In addition, it does not consume much power. The fabric clock connects to 16K registers distributed over the entire chip. It is relatively easy to design a low-power, low-skew distribution network for the fabric clock, however. The fabric clock loads are regularly distributed around the fabric. A buffered H-tree distributes the clock symmetrically to all the loads. All wires in the H-tree network are shielded to avoid any noise problems. The fabric clock is gated by the control and interface logic so that it transitions only when computation needs to be performed. While it is not exploited in the current implementation, it is possible to use configuration information to determine the exact set of registers that are being used and that require clocking, which minimizes clock power dissipation even further. Because of its regularity and predictability, the first two benefits of asynchronous logic do not apply to the PipeRench architecture. Our examination of asynchronous PipeRench is motivated, therefore, by the pursuit of increased performance through the reduction of timing pessimism.

II. Related work

There are a number of papers in the literature on asynchronous, programmable logic. Montage [4] presented an asynchronous FPGA for prototyping of asynchronous logic. STACC [5] is another asynchronous FPGA architecture. The STACC paper presents average-case performance as a prime motivation for the architecture, but does not quantify its benefits or costs. To our knowledge, this is the first paper to discuss the interaction of hardware virtualization and average time performance.

III. PipeRench Architecture Summary

Figure 1 illustrates the PipeRench architecture as described in [3]. PipeRench is not a general piece of programmable hardware. Rather, it supports pipelined datapaths. Each pipeline stage consists of logic and registered storage. Each connection from one stripe to the subsequent stripe passes through a register. There are no connections to prior stripes, or combinational paths to subsequent stripes. Stripes consist of a number of processing elements, or PEs, which perform parallel operations in each pipeline stage. In the current implementation of PipeRench, there are 16 PEs per stripe, and each PE contains an 8-bit ALU, a pass register file, and interconnect resources to communicate with other PEs. The PEs can be cascaded to construct wider functional units.

PipeRench's unique feature is its ability to automatically virtualize hardware, which is enabled by the above constraints on the application. Figure 2 illustrates the process of virtualizing a six stage (or stripe) application on four physical stripes. PipeRench provides a high degree of compile-ability. Compile-ability is enabled first by the virtual hardware model, which frees the compiler from having to fulfill a size constraint. In addition, PipeRench has intra-stripe interconnect consisting of sixteen eight-bit busses that allow any PE to communicate with any other PE on the same or subsequent stripe.

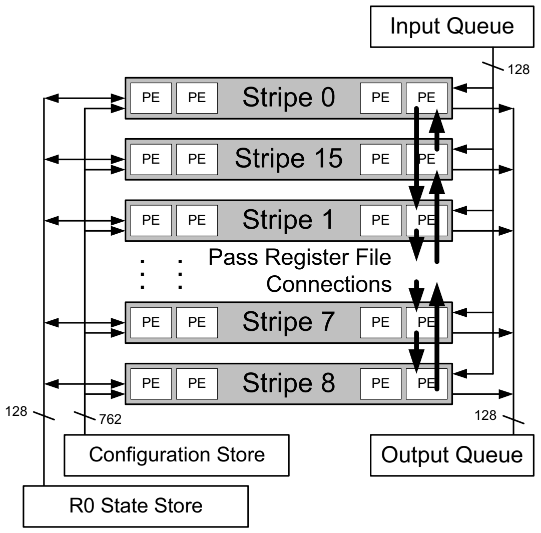


FIGURE 1. PipeRench Architecture: with sixteen stripes consisting of sixteen PEs each.

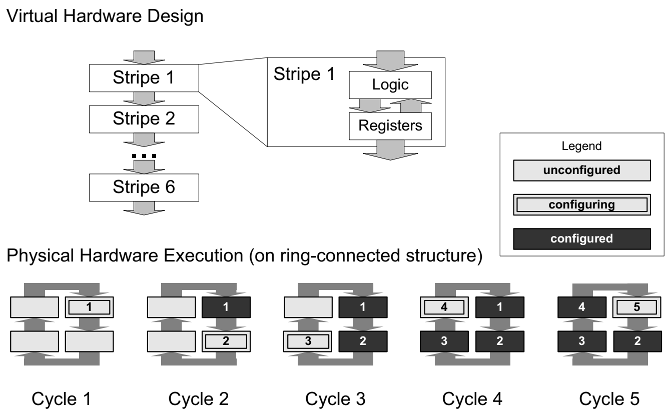


FIGURE 2. PipeRench Hardware Virtualization: This example illustrates the first five cycles of a fabric with four physical stripes executing an application consisting of six virtual stripes.

A. Hardware Virtualization and Average-case Performance

The frequency of the fabric clock is determined by the critical path within the virtual hardware design. The compiler accepts a cycle time constraint and timing information, and attempts to create a design that works at the specified frequency. A longer clock period gives the compiler more flexibility in placing and routing operations in the fabric, which usually results in higher hardware utilization.

It is highly unlikely that the compiler will be able to use the entire clock cycle in every stripe in the virtual design. Only a subset of the resources in the design will be on the critical path. In the current implementation, the fabric clock period is greater than or equal to the length of the longest path in any of the virtual stripes. However, in a virtualized hardware device like PipeRench, a stripe with a critical path may not even be currently executing in the fabric. To understand what this means, consider a virtual hardware design consisting of four stripes. As illustrated in Figure 3, three of the stripes of this design have a critical path of 1 ns. The remaining stripe has a critical path of 2 ns. Figure 3 shows a timeline of execution of this virtual design on a fabric with 3 physical stripes. Only two of the four cycles in the steady state have the stripe with the critical path of 2ns configured into the fabric. Therefore, on half of the cycles in this steady-state, synchronous PipeRench is being clocked at half the frequency that assures accurate execution. A performance gain of 33% could be achieved in this case if we could somehow clock the fabric at minimum time required by all the virtual stripes currently running in the fabric.

Figure 4 shows the throughput achieved by a virtual design as a function of the number of physical stripes. In this case, the virtual design is 100 stripes, and one of those stripes has a critical path twice the length of the other 99 stripes. The straight line shows the throughput as a function of the number of physical stripes if the design is run at the critical path of the slowest virtual stripe. The curved line shows the throughput if the same design is run while the clock is adjusted every cycle to the longest path in the stripes that are currently configured into the design.

It is also true that the critical path in a design is not always activated, even when it is configured into the physical fabric. To understand what we mean, consider the critical path of a ripple carry adder. This path goes from the carry-in of the least-significant bit to the carry-out of the most-significant bit. Yet this path is only activated if the carry is propagated across all the bits of the adder. If this is a 32-bit adder, and the inputs are randomly distributed, the critical path will only be activated once in every 2^{32} cycles.

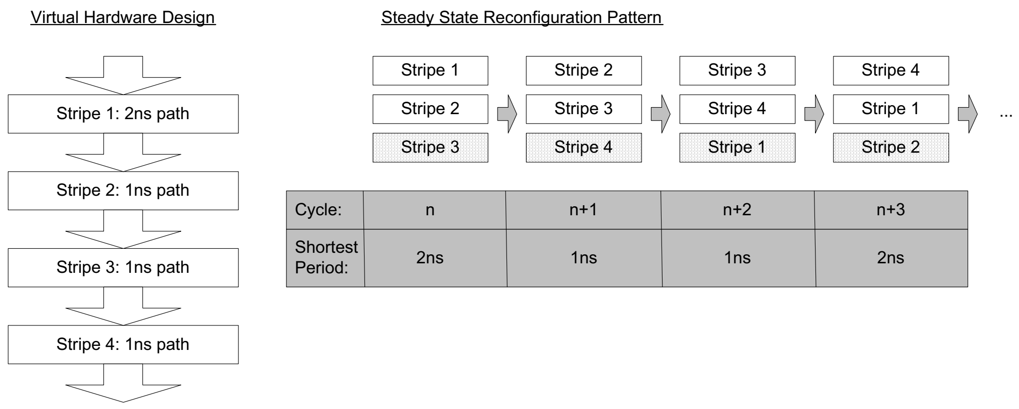


FIGURE 3. Hardware Virtualization: Using a design with variable periods for each stripe

In summary, synchronous PipeRench currently suffers due to two kinds of pessimistic estimations of performance: 1) a portion of the design that is on the critical path may not even be configured in the fabric and 2) the critical path may not be activated based on the current inputs. In this paper, we will examine the reduction of this pessimism in three different ways. First, we will investigate a technique that allows the compiler to specify the period of each stripe, and an architecture that uses this information to vary the clock rate. This is still a synchronous implementation technique, but it attempts to reduce the first kind of timing pessimism. Second, we will investigate fully asynchronous implementations of PipeRench, which reduce both kinds of timing pessimism. We consider two possibilities: one where the completion status of an entire stripe is computed, and one where the completion status is computed individually for each of the sixteen PEs within one stripe. We use two example applications to investigate the performance benefits of each technique. The first application is an automatically compiled version of the IDEA encryption algorithm, and the second is a manually optimized version of that same algorithm. This allows us to explore the interaction between the compiler and the asynchronous architecture.

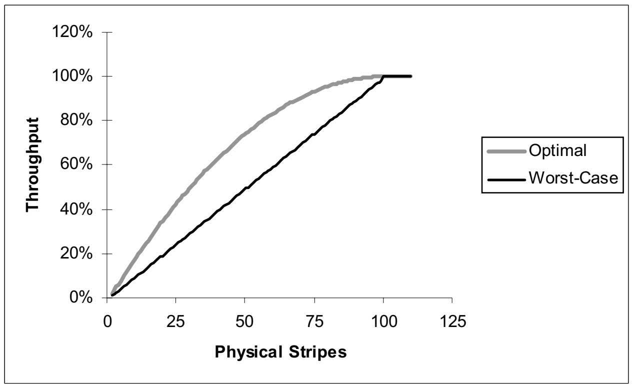


FIGURE 4. Throughput comparison of optimal and worst-case clocking. The y-axis shows throughput as a percentage of throughput achievable if there is no hardware virtualization.

IV. Variable Clock Approach

The first architectural experiment we have conducted consists of using the existing PipeRench synchronous model, with estimates of timing for each of the components (interconnect, functional unit and registers). First, we determine the longest timing path in the whole virtual hardware design. This would be the fastest that we could run the existing hardware with certainty that we have no timing problems. Second, we determine the longest timing path *within each stripe.* The results are shown in Figure 5(a). This is a plot of the automatically compiled version of IDEA encryption [6] using the DIL compiler [7] with no timing constraints specified. Because no timing constraints are specified or enforced, the DIL compiler focusses on increasing the utilization of the PEs in the fabric. As a result, the lengths of the critical paths for each stripe are widely distributed. The units in Figure 5 are abstract and based on best-effort architectural estimates of delay. We have not yet calibrated these numbers with the physical chip.

Next, we take this timing data, and with a simple timing simulator, determine how long it would take to execute the application using:

- A constant clock at a period equal to the longest critical path in the entire virtual hardware design, and
- A variable clock that has a period equal to the longest critical path in the portion of the virtual hardware design that is currently configured in the hardware.

The results of this experiment are plotted in Figure 6 as throughput versus physical stripes. In this paper, because of the lack of hard numbers for delay, all throughput numbers are relative and unitless. The difference in throughput between the fixed and variable clocks is most pronounced when the number of physical stripes is small. This design has 111 virtual stripes. Therefore, when the number of physical stripes is equal to or greater than 111, the throughput for both solutions is equivalent, because it is certain the stripe with the critical path is configured in the fabric. In fact, looking at the delay histogram in Figure 5(a), one can see that the greatest span of stripes without a long delay (over 40 units) is only about 47 stripes (from stripe 15 to 62). Therefore, when there are more than 47 physical stripes, the chip must run with a period of 40 units or greater.

From an implementation perspective, a truly variable clock is difficult to implement. PLLs can generate variable clock frequencies but these devices cannot change the clock period on a cycle-by-cycle basis. PLLs take time to stabilize at a new frequency. One way to possibly implement this variable clock would be to use a single high-speed clock. Each stripe would express its critical path as a multiple of that very short clock period. Hardware would compute the maximum timing path currently in the hardware. A counter, running on the high-speed clock, would count up to the maximum timing path to determine the time for the next clock pulse. This design is illustrated in Figure 7.

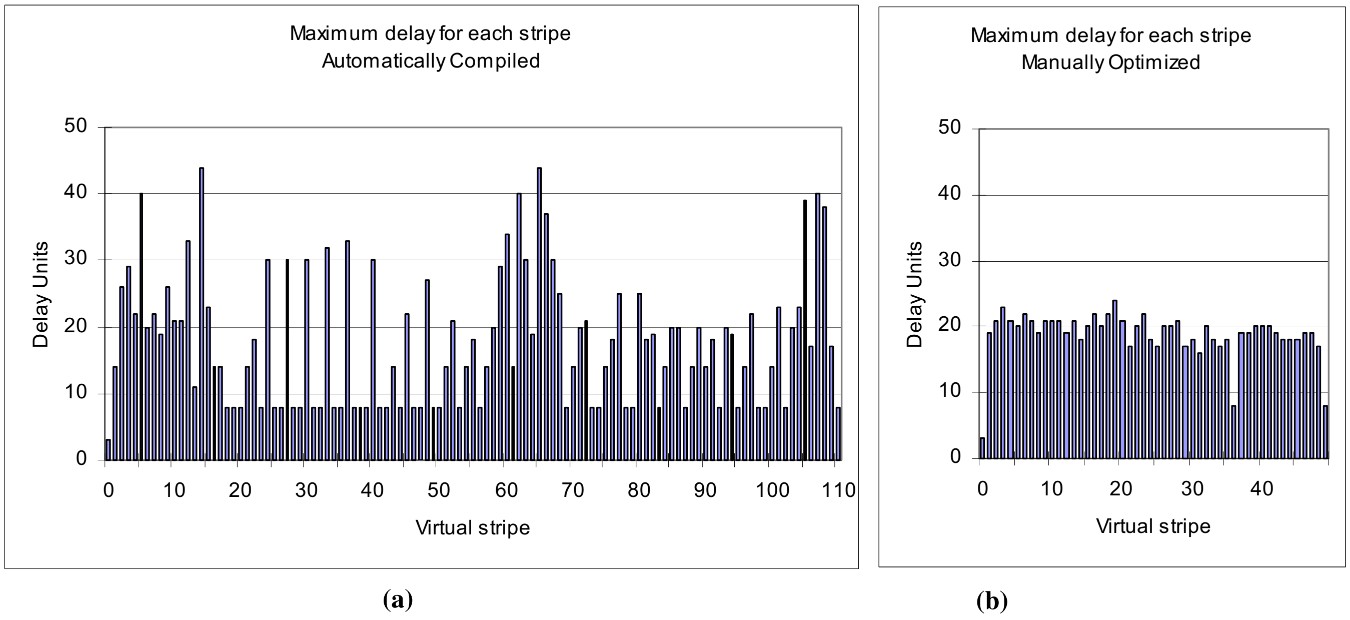


FIGURE 5. Delay distribution in compiler-generated (a) and manually optimized (b) IDEA encryption.

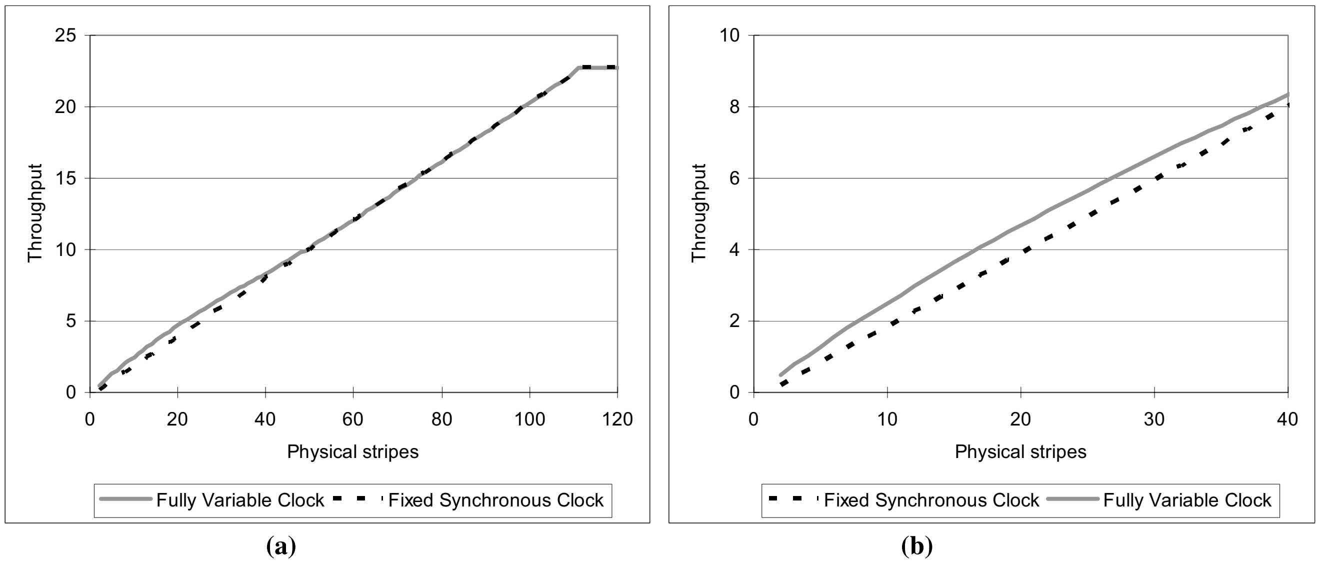


FIGURE 6. Throughput versus Physical Stripes using Automatically Compiler IDEA Encryption and Variable Clock: (a) full range, (b) zoomed in for less than 40 physical stripes.

Of course, this solution means that somewhere on the chip there must be a very high frequency clock, and the critical path of a stripe can only expressed in multiples of that clock period. This high frequency clock should run at some multiple N of the frequency of the fixed clock required for the virtual hardware design. Other stripes would have their critical path expressed as a number less than or equal to N. If N is large, it allows very fine timing control, but also requires a very high speed clock and a larger counter that is capable of running at that frequency. Smaller values for N are easier to implement, but causes the timing of more stripes to be overestimated. Experimentally, we determined that with N=8 we can achieve 85% of the benefit of a fully variable clock. A larger N improves performance slowly, and makes implementation much more difficult. Clock multiples less than eight dramatically reduce the gain from variable clocking in this example.

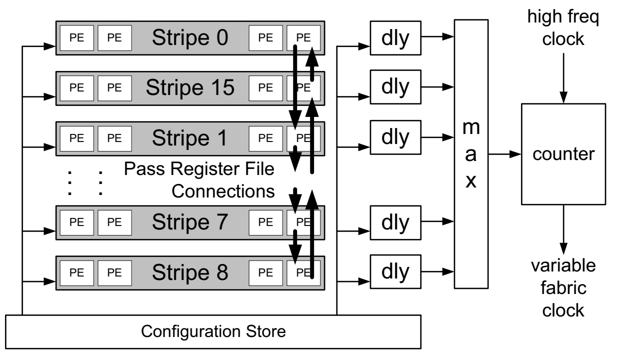
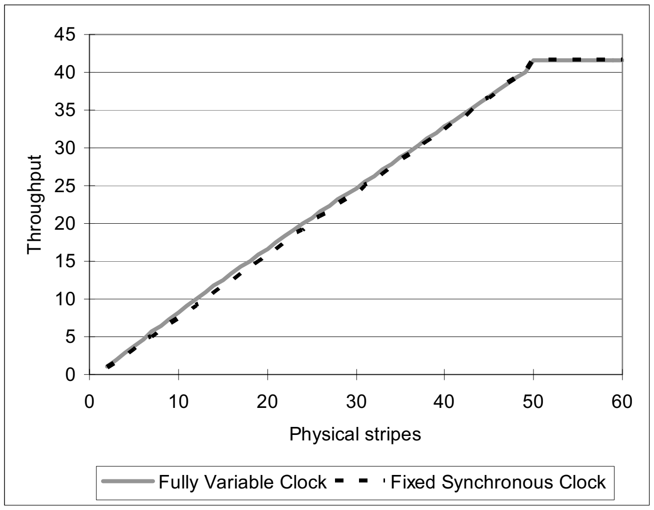


FIGURE 7. One Possible Variable Clock Implementation Schemes: using one counter with P-input maximum computation.

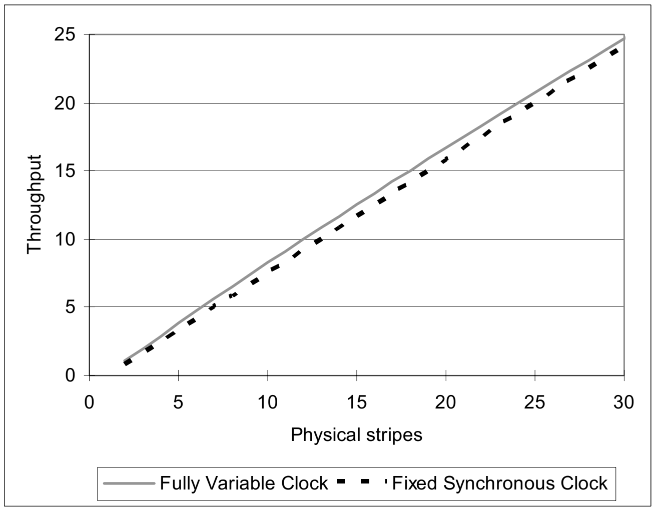
The performance benefit of this technique is greatest when the distribution of critical path delays is very great between stripes. The best-case scenario for this technique occurs when the compiler has done a poor job of balancing delays between stripes. Running without timing constraints, the DIL compiler does not do a very good job optimizing the schedule or placement. The compiler could be improved, but we found it more expedient to hand-optimize this application. Special attention was paid to balancing critical path delay and improving utilization. The resulting design is much smaller (50 stripes) and has a much shorter critical path. The delay histogram of this design is shown in Figure 5(b). The results of the variable clock implementation of this design are shown in Figure 8. It can be seen that there is almost no benefit to the variable clocking approach, except for compensating for bad compilation.

V. Fully-Asynchronous Approaches

The compile-time approach to exploiting variable timing does not seem to pay off. We now investigate whether dynamic computation of the completion of a cycle can substantially improve performance. In our first experiment in fully asynchronous implementation, we assume that each stripe is capable of computing when it was done. The next computation can proceed when every stripe in the fabric completes its computation. This leverages the fact that the critical path even within a stripe is a worst-case number. During real execution, worst-case timing paths may not be activated, allowing the computation to proceed even faster. In a later experiment, we will discuss the benefits of determining completion on an individual PE basis within the stripe.



(a)



(b)

FIGURE 8. Throughput versus Physical Stripes: Variable Clock, IDEA manually optimized. (a) full range, (b) zoomed in on less than 30 stripes.

A. Stripe-based Completion

To measure the potential speedup of this technique, we instrumented the conventional fixed-clock PipeRench simulation model so that we could determine when the outputs from each stripe stop changing during each cycle.The difference between the instant the outputs stop changing and the end of the fixed clock period is called the slack time. That slack time is accumulated during the course of a substantial computation. We measure the time that the fixed-clock simulation takes to perform the computation. Assuming we had a perfect completion circuit with no pessimism, we could subtract the accumulated slack time from the time it took the fixed-clock implementation to complete.

Graphs of throughput versus physical hardware are shown in Figure 9. In this figure we show both the compiled and hand-optimized IDEA kernels. The maximum benefit of this approach is approximately 20% compared to the fixed clock implementation. Unlike the variable-clock approach, the benefit of fully asynchronous implementation spans all implementations, and does not decrease as the size of the physical hardware grows. True asynchronous implementations even show improvement in performance for hand-optimized implementations of the PipeRench design.

B. Implementation Issues for Stripe-based Completion

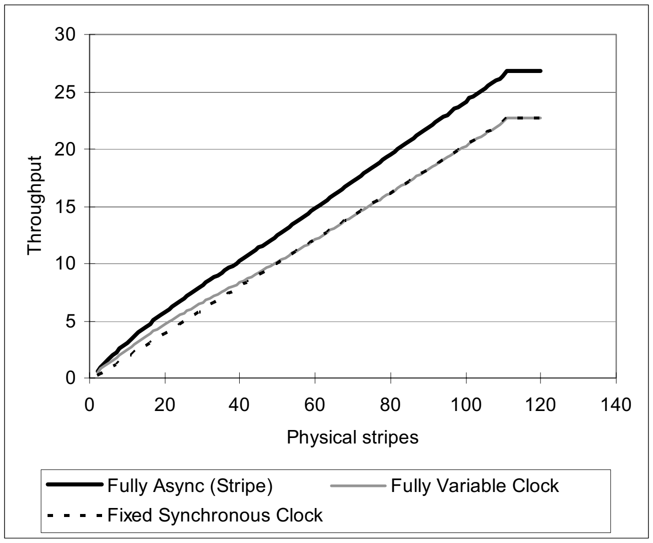
A possible implementation of PipeRench with stripe-based completion is shown in Figure 10. We assume that all the 1024 bits of output from the register file from one stripe are connected to the subsequent stripe and are bundled with a request and acknowledge wire to allow the communication to be self-timed. Request and acknowledge signal must also accompany the global input bus and the global output bus. A token indicating the current stripe that is being reconfigured is propagated around the ring of the fabric. The token moves from one stripe to the next when reconfiguration is done on one stripe, and the computation on the subsequent stripe is complete.

The costly part of this implementation is the hardware that computes the completion of an entire stripe. Since there are sixteen eight-bit PEs in a stripe, this computation may be rather complex, and will require some long wires. It may be more efficient, therefore to reduce the size of the component that computes completion. The next section investigates the implementation of PipeRench where each of the PEs compute their own completion.

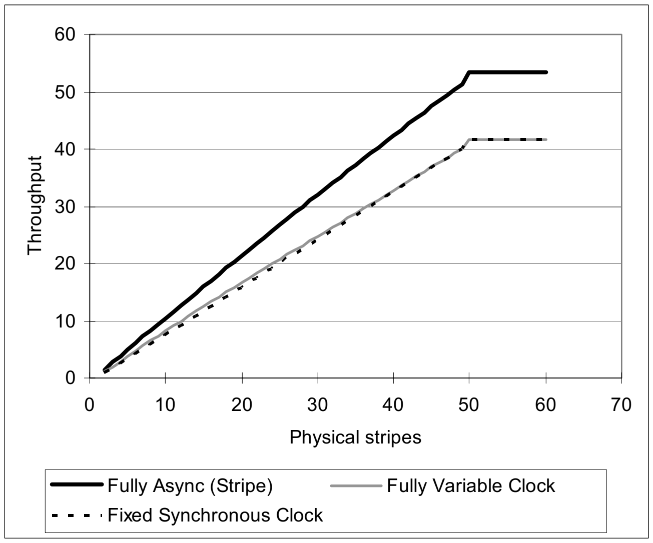
C. PE-based Completion

Estimating performance for a asynchronous implementation using PE-based completion is more challenging because the current PipeRench architecture fundamentally reconfigures on a stripe-wide basis. PE-based completion provides a new alternative, however, because one PE in a stripe can complete its computation and begin reconfiguring before the other PEs in the stripe are done with their computation. As a result, the "wave front" of reconfiguration can deform around slow computations, as illustrated in Figure 11.

We constructed a complete Verilog simulation model of PipeRench using PE-based completion. This simulation requires tremendous amounts of memory. In fact, simulating PipeRench with 25 stripes requires about 4 GB of memory. We could not effectively simulate a PipeRench fabric greater than 25 stripes. We again simulated the chip on the two ver-



(a)



(b)

FIGURE 9. Throughput versus Physical Stripes: Stripe-based asynchronous compared against variable clock and fixed clock. (a) compiled IDEA, (b) manually optimized IDEA.

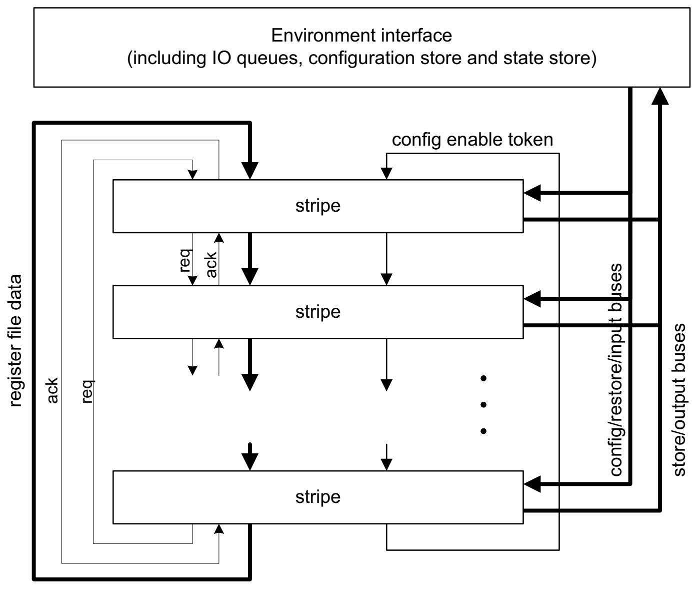


FIGURE 10. Possible Implementation of Stripe-based Asynchronous PipeRench.

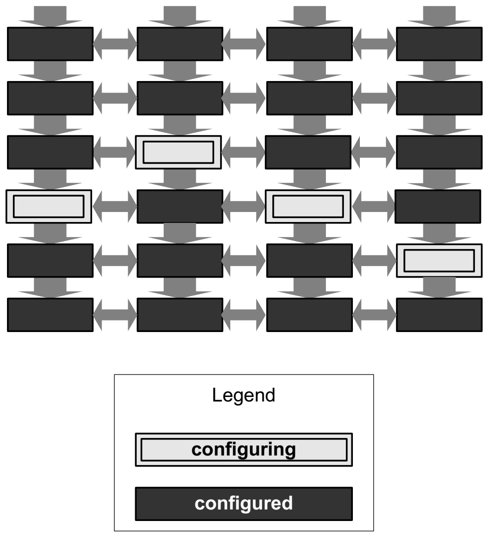


FIGURE 11. Variable wave front of reconfiguration through an asynchronous PE-based implementation of PipeRench.

sions of the IDEA encryption algorithm. The results are shown in Figure 12.

PE-based reconfiguration shows a dramatic improvement in the performance of the system. Since much of the IDEA application consists of 16-bit additions, and these are implemented by cascading two eight-bit PEs, PE-based completion allows the least significant PE to begin its reconfiguration before the most significant PE has completed the addition. Compared against fully synchronous operation, the PE-based asynchronous implementation improves performance by approximately 80%. Both compiler-generated and manually-optimized design benefit from the PE-based implementation.

To demonstrate that these basic performance tradeoffs hold for other applications, we also used DIL to map the Discrete Cosine Transform, or DCT, to PipeRench. The performance numbers for all four considered clock architectures are shown in Figure 13. Again, the PE-based asynchronous approach is the best technique, followed by the stripe-based asynchronous approach and the variable clock approach. This virtual hardware application has one very long stripe, which means that variable clock actually does better than fixed clock over a larger spread of physical stripes than was seen with IDEA.

D. Implementation Issues for PE-based Completion

The structure of our simulation model of the PE-based asynchronous PipeRench is illustrated in Figure 14. The circuit requires an interface to the external configuration circuit similar to the stripe-based architecture in Figure 10, except that each PE has a configuration token. In addition, this implementation requires a completion network internal to each stripe.

The costliest aspect of this implementation is the crossbar routing network. In synchronous PipeRench, each PE can broadcast a value to every PE on that stripe or the next stripe. The source PE has no configuration information that indicates which PEs it sends its value to. This is difficult to implement in an asynchronous methodology, because every connection requires both the source and the destination to be aware of the connection, so that it can watch the correct request and acknowledge signals. As a result, the configuration space of each PE must be increased to allow the PE to watch the correct acknowledge signals. Synchronous PipeRench frequently uses this technique of sending a result and not being aware of its destination. The configuration word for each PE grows from 42 bits in the synchronous implementation to 178 bits in the PE-based synchronous implementation.

The other costly aspect of PE-based asynchronous implementation is the additional wires required for all interfaces. Nearly twice as many wires must go into or come out of each PE compared to the synchronous implementation. This is crucial, because synchronous PipeRench is nearly a wire-bound design. When implemented in a technology with five metal layers, the synchronous PE was wire-bound. Even in 0.18 micron, with six metal layers, the PE would clearly become wire-bound if the wire count crossing the PE boundary was to double.

Without doing a more detailed design, it is impossible to determine more accurately the area impact of asynchronous implementation. Assuming an application is virtualized, asynchronous implementation would only be beneficial if it does not increase area by more than 80%.

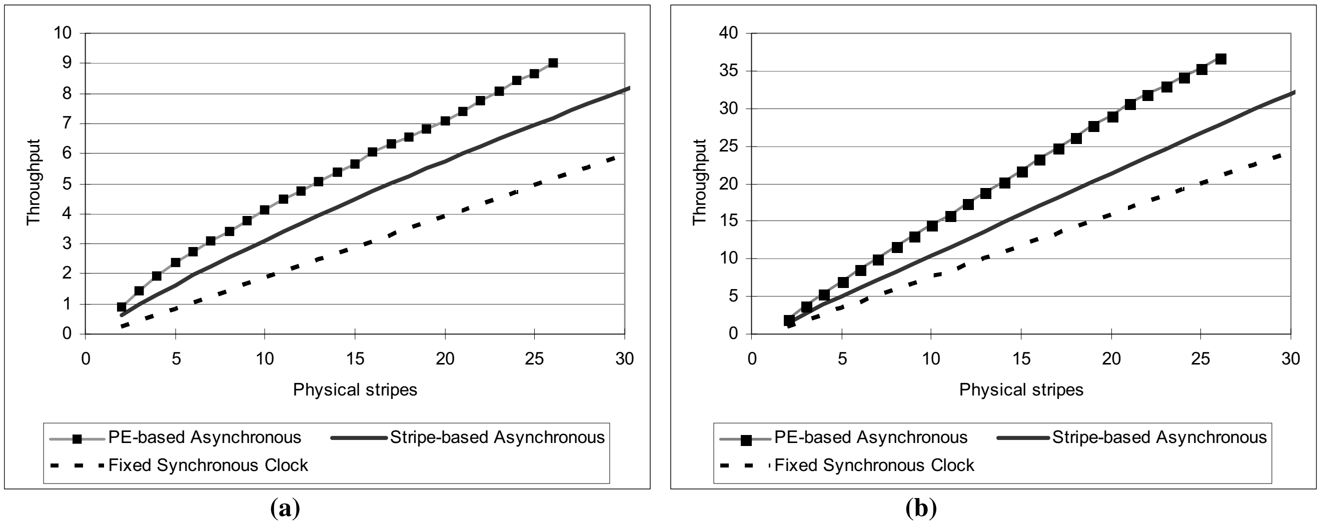


FIGURE 12. Throughput versus Physical Stripes: PE-based Synchronous compared against other alternatives. (a) compiled IDEA, (b) manually optimized IDEA.

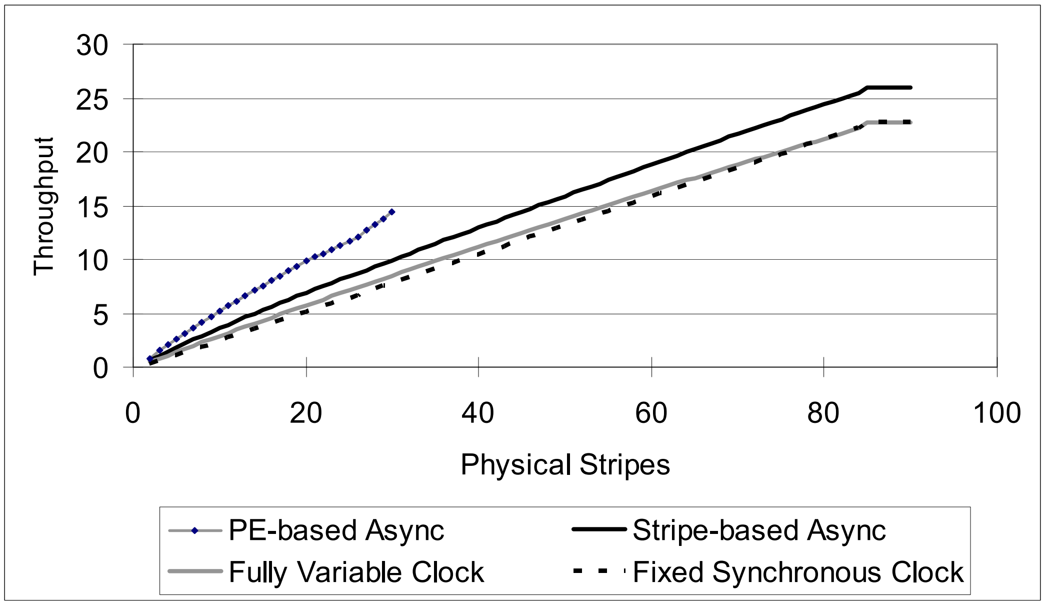


FIGURE 13. DIL-compiled DCT and the throughput vs. stripes trade-off for four clocking strategies.

E. Asynchronous Implementation versus Compile-ability

One of the key interesting trade-offs we have discovered is that the features that make PipeRench a great compilation target are also those that make the architecture expensive to implement in an asynchronous manner. A PE-based architecture would be much more efficient if it only had local connections to other PEs within the stripe. But the crossbar interconnect within the stripe is what makes PipeRench capable of guaranteeing one-pass compilation.

VI. Conclusions

Asynchronous implementation of PipeRench can improve performance for each stripe by approximately 80%. Only PE-based completion, however, seems to offer a significant speedup for both optimized and automatically compiled designs and for all levels of virtualization. Using a variable synchronous clock only seems to make up for poor compilation quality. Stripe-based completion does not seem reasonable to implement. The current PipeRench architecture is not well suited to asynchronous implementation, however. A difficult issue is that the same features that make the architecture a good compilation target are also the most costly features in the asynchronous implementation.

VII. Acknowledgments

This work was made possible by the Ministry of Education, Culture, Sports, Science and Technology of Japan, who supported Professor Kagotani's visit to Carnegie Mellon University. Professor Schmit is supported in part by an NSF CAREER grant and the Gigascale Research Center.

VIII. References

[1] C. J. Myers, *Asynchronous Circuit Design,* Wiley, 2001.

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

[3] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, R. R. Taylor, "PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology," *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, 2002.

[4] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for Implementing Asynchronous Circuits," in *IEEE Design & Test of Computers*, Vol. 11, No. 3, pp. 60-69, 1994.

[5] R. Payne, "Self-timed FPGA Systems," in *Field-Programmable Logic & Applications*, pp. 21-35, Springer-Verlag, Berlin 1995.

[6] B. Schneier, *Applied Cryptography, Second Edition*, Wiley, 1996.

[7] M. Budiu and S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," in *International Symposium on FPGAs*, 1999.

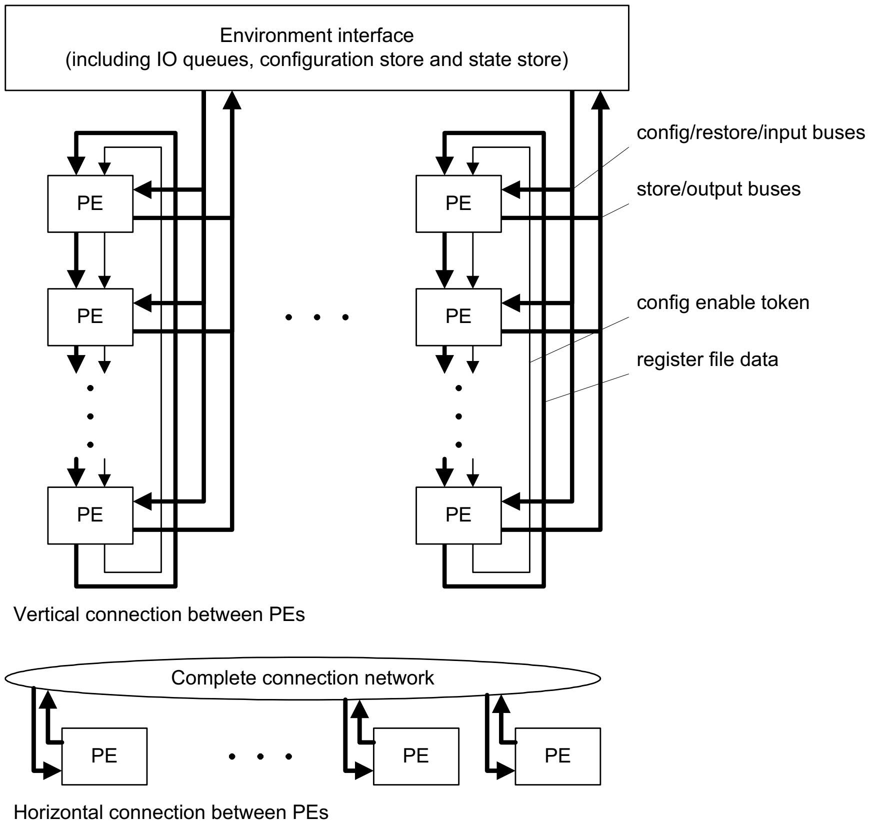

Environment interface
(including IO queues, configuration store and state store)
config/restore/input buses
store/output buses
config enable token
register file data
PE
Vertical connection between PEs
Complete connection network
Horizontal connection between PEs

FIGURE 14. Possible Implementation of PE-based Asynchronous PipeRench.