Tunable Fault Tolerance for Runtime Reconfigurable Architectures

Steven K. Sinha[†*], Peter M. Kamarchik[†], and Seth C. Goldstein[†*]
Department of Electrical and Computer Engineering[†]
School of Computer Science[*]
Carnegie Mellon University
Pittsburgh, PA 15213
{ssinha, pmk, seth}@ece.cmu.edu

Abstract

Fault tolerance is becoming an increasingly important issue, especially in mission-critical applications where data integrity is a paramount concern. Performance, however, remains a large driving force in the market place. Runtime reconfigurable hardware architectures have the power to balance fault tolerance with performance, allowing the amount of fault tolerance to be tuned at run-time. This paper describes a new built-in self-test designed to run on, and take advantage of, runtime reconfigurable architectures, using the PipeRench architecture as a model. In addition, this paper introduces a new metric by which a user can set the desired fault tolerance of a runtime reconfigurable device.

1. Introduction

Increased prevalence of computer systems everywhere from industry to small home-office businesses has created a need for greater reliability in hardware and software. With many mission critical tasks being delegated to computers, a degree of fault tolerance in the computer's circuitry is required to make sure unforeseen problems can be dealt with gracefully. With more frequent use of reconfigurable hardware in embedded and wireless computing devices, there is a need to develop an easy and effective fault detection system for reconfigurable devices.

Some research has been done with Built-in Self-Tests (BISTs) on Field Programmable Gate Arrays (FPGAs) [1-2],[6-13]. FPGAs are a general class of reconfigurable hardware which contain an array of programmable logic blocks (PLBs), with programmable interconnect between PLBs, as well as programmable I/O cells. The configuration bits used to program the FPGA determine the function of the device; one such function can be a BIST. For most FPGA architectures, however, configuration time is large, and there is no ability for partial reconfiguration, which makes a frequent BIST difficult to implement, and makes a dynamic and tunable BIST nearly impossible [3].

The PipeRench architecture represents a new direction for FPGAs. PipeRench is a runtime-reconfigurable FPGA that manages a virtual pipeline, allowing time-multiplexed use of the physical pipeline stages. The logical size of a virtual pipeline is unbounded, and it can be executed on a compatible architecture of any size [4]. Fixed hardware constraints are no longer an issue for the compiler with hardware virtualization. PipeRench is also forward compatible. Additional hardware will only add to the performance of the application by allowing more virtual stripes to fit on the physical fabric at any one time.

The reconfigurable nature of PipeRench lends itself to fault tolerance. The ability to dynamically reconfigure the fabric makes a frequent BIST feasible. Our goal was to develop an effective BIST algorithm for PipeRench that did not require any additional hardware. The addition of BIST hardware to a device makes chip implementation more difficult and also potentially degrades the performance of applications that do not require the BIST. This paper describes the faults that can affect the PipeRench architecture, methods to detect the faults, and ways to work around them once they have been identified. This is accomplished with only minor modifications to the control logic.

Unlike traditional BIST as previously applied to FPGAs or to custom hardware, our testing procedure only tests the parts of the FPGA that are currently in use. With an FPGA, at any point in time only a portion of the hardware is being used, the portion that is configured for the current application. Because of this, exhaustive tests of the FPGA fabric are wasteful. By testing only the part of the fabric that is executing the application, we can retain better application performance while still covering a large percentage of relevant faults. We will refer to our testing scheme as BIAST (Built-In Applicable Self-Test).

Integral to our approach to testing is the fact that testing happens concurrently with the execution of the application. Furthermore, the amount of resources dedicated to testing can be tuned at runtime to trade off security with application throughput. We thus introduce a new metric to evaluate the effectiveness of a BIAST algorithm; a metric that gauges the amount of time taken to detect a fault rather than the absolute fault coverage. This metric, which we refer to as Mean Time to Detect a Fault (MTDF), gives the average amount of time to detect any fault that has occurred. The metric gives the user the means to measure and set the amount of fault tolerance of

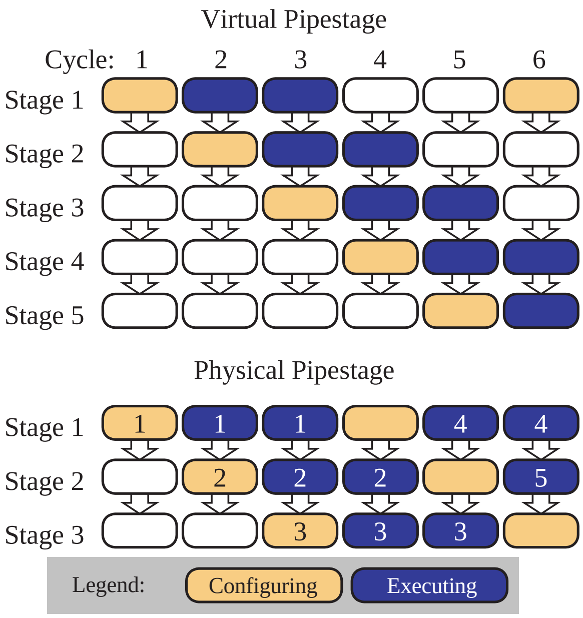


Figure 1.

PipeRench Reconfiguration. This diagram shows the progress of virtualizing a five-stage pipeline on a three-stage device.

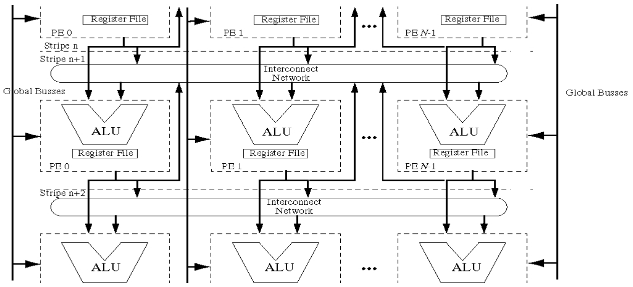


Figure 2.

PipeRench Architecture: PEs and Interconnect

the system, based on the desired level of fault tolerance and application performance.

In the next section, we discuss PipeRench, the reconfigurable fabric targeted by our BIST algorithm. In Section 3, we introduce the BIST algorithm. In Section 4, we analyze its performance and introduce a standard by which to measure it. The algorithms to isolate a fault are discussed in Section 5 and the methods to tolerate faults are described in Section 6. We cover related work in Section 7 and state conclusions in Section 8.

2. The PipeRench Architecture

The PipeRench architecture implements pipelined reconfiguration, a method of virtualizing pipelined hardware application designs by breaking each design into pieces that correspond to pipeline stages in the application. These pieces are then loaded, one per cycle, into the fabric. This makes it possible to perform the computation, even if the whole configuration is never present in the fabric at one time.

The virtualization process is illustrated in Figure 1, which shows a five-stage pipeline being virtualized on a three-stage fabric. The top portion of this figure shows the five-stage application and the state of each of the stages of the pipeline in five consecutive cycles. The bottom half of the figure shows the state of the physical stages in the fabric that is executing this application. An effective metaphor for this procedure is scrolling on a text window. Once the pipeline is full, every five cycles generates two results from the pipeline. In general, when an application having N_v stages is virtualized on a device with a capacity of N_p stages ($N_p < N_v$), the throughput of the implementation is proportional to $(N_p\text{-}1)/N_v$. Throughput is a linear function of the capacity of the device. Therefore, decreases in feature size (resulting in more physical hardware) and increases in clock speed will increase the throughput, until $N_p = N_v$. Thereafter, performance of the application continues to improve only through increased clock speed.

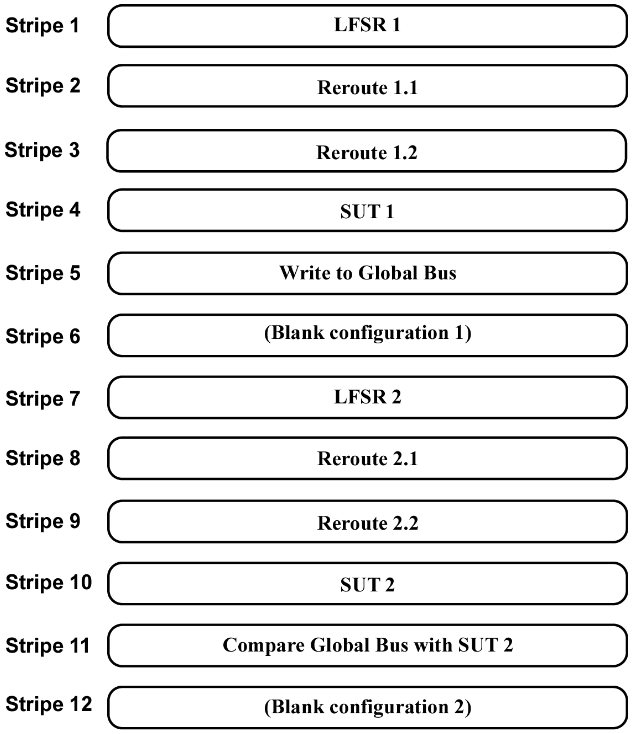
Because the configuration of stages happens concurrently with the execution of other stages, there is no loss in performance due to reconfiguration. As the pipeline is filling with data, stages of the computation are being configured ahead of that data. Even if there is no virtualization, configuration time is equivalent to the pipeline fill time of the application. Therefore configuration does not reduce the maximum throughput of the application.

In order for this virtualization process to work, the state of any pipeline stage must be a function only of the current state of that stage and the current state of the previous stage. In other words, cyclic dependencies must fit within one stage of the pipeline. Interconnect that directly skips over one or more stages is not allowed, nor are connections from one stage to a previous stage. Fortunately, many computations on streaming data can be pipelined within these constraints. Furthermore, by including structures we call *pass registers*, it is possible to create virtual connections between distant stages.

The primary challenge facing pipeline reconfiguration is configuring a computationally significant pipeline stage in one clock cycle. To do this, a wide on-chip configuration buffer (either SRAM or DRAM) is connected to the nearby fabric. The word *stripe* is used to describe both the physical stages in the fabric (the *physical stripes*), and the configuration words that are written into them (the *virtual stripes*). Any virtual stripe can be written into any physical stripe. Therefore, all physical stripes must have identical functionality and interconnect. This allows for excellent fault tolerance, because, if any physical stripe is damaged, unless it cannot pass data through itself unchanged, it can simply be removed from use. Although this does affect performance by reducing the number of physical pipeline stages by 1, the application can still run.

The fabric of PipeRench is composed of a set of physical pipeline stages, or stripes. Each stripe is composed of an interconnect and an array of processing elements (PEs), which contain registers, shifters, and ALUs. An ALU is composed of look-up tables (LUTs) and extra circuitry for carry-chains, zero-detection, etc. (See Figure 2). The PEs have access to a global I/O bus. The interconnect network is composed of *interstripe* and *intrastripe* connections. Interstripe interconnect moves the pass register data from one stripe to the next. Through the intrastripe interconnect the PEs can access operands from registered outputs of the previous stripe as well as registered or unregistered outputs of the other PEs in the stripe.

The current PipeRench chip has four global busses. Two of these busses are dedicated to storing and restoring stripe state during hardware virtualization. The other two are used for data input and output.



Blank configurations used only if an error is detected.

Figure 3.

BIST stripe layout.

3. Built-in Self-test Algorithm

Since PipeRench combines uniformity in its logic (PEs, LUTs, etc.) with the ability to reconfigure the fabric quickly and easily, a BIST becomes both a fast and efficient method for detecting faults. Faults in PipeRench can occur in the PEs, the interstripe interconnect, the intrastripe interconnect, the global bus, the configuration lines, the controller, and the memory (FIFO, store, configuration). This BIST can detect faults occurring in any non-memory component except the configuration lines and controller, which are vital to the algorithm's execution. In this paper we do not address the mechanisms, which are already well known, that can ensure that the memory is fault free.

In normal operation, a PipeRench device is continually configuring one stripe (pipe stage) of an application ahead of relevant data. After the last virtual pipe stage is configured, the device starts again with the first virtual stripe of the application. Since there is, by definition, no connection between the last stripe and the first, there is an opportunity to insert additional stripes between the last stripe and first without affecting the functionality of the device. The only cost is the one cycle delay per added stripe before the application's first stripe resumes computation. The inserted stripe (or stripes) can be used to test the hardware. A positive outcome for the test would help assure accuracy of the device's output. A negative result could be handled appropriately.

Our BIST algorithm takes twelve stripes, vigorously testing two stripes at a time (see Figure 3). In those twelve stripes, we configure two stripes under test (SUTs) identically, and stimulate each with identical pseudo-random input. The results are compared to each other for discrepancies: If one exists, there is a fault. Sections 5 and 6 describe how we isolate the fault and how the fault is tolerated.

A more detailed look at the BIST shows slightly greater complexity. Each PE of every stripe takes two inputs, A and B. Those inputs come directly or indirectly from the registered output of the last stripe or from the global bus. To stimulate the inputs of each SUT with known data, we need to create the data and store it in register locations that will be chosen as the A and B inputs of the PEs of the SUT. The A inputs should be different from the B inputs so as to test various bit combinations as input into each PE. Thus, in the first stripe of the BIST we configure two unique test pattern generators: two linear feedback shift registers (LFSRs) are identically configured with different initial values. Due to the nature of PipeRench, the outputs of the LFSRs are saved in the first registers of the PEs that are configured to output the LFSR value. Should the PEs in first SUT be receiving their inputs from any place other than those first registers, the LFSR values will have to be rerouted to the correct location. The second and third stripes of the BIST take care of this requirement. The second stripe reroutes the output of the first unique LFSR to the registers that will be used as input A for each PE of the SUT. The third stripe reroutes the output of the second unique LFSR to the registers that will be used as input B of each PE of the SUT. These reroute stripes are thus a function of the SUT's configuration bits. The SUTs are required to be stimulated identically. Thus stripes 1 and 7, stripes 2 and 8, and stripes 3 and 9, are each, respectively, configured identically. Note that any of the PEs of the SUTs that take inputs from the global bus will be getting the same data from the same global bus, so the SUTs will continue to be stimulated identically. The output of the first SUT is written to the global bus in Stripe 5, and is read in and compared to the results of the second SUT in Stripe 11.

This BIST runs in a special test mode. Following the configuration of the last stripe of the application and the subsequent configuration of the BIST stripes, normal (sequential) configuration is halted and enough time is given for the application to drain all of its data before testing begins. After testing is complete, normal configuration restarts, commencing with the configuration of the first virtual stripe of the application.

3.1. Built-in Applicable Self-test (BIAST)

There are three variables to this BIST: the number of test vectors generated for each SUT, the number of configurations tested on each SUT, and the makeup of those configurations. There are simple mathematical relationships between the first two and the amount of relative fault coverage obtained by running the BIST, as we will demonstrate in the next section. However, the third variable presents more of a problem. There are an almost infinite number of possible configurations for the SUTs, so it is important to test those configurations that are most pertinent to whatever application is running.

There are two common choices for test configuration generation: hardwired test configurations, which take up a large amount of memory, and pseudo-random test configuration generators, which take up a large amount of logic space, increase the area that might contain faults, and require many cycles to cover any significant set of configurations [6]. Beyond the mere introduction of their inherent problems to a device, the addition of hardwired test configurations or dedicated test configuration generators conflicts with our goal to implement a BIST without additional hardware.

The solution to our problem is simple and straightforward. Since PipeRench stores all the stripe configurations for the current application on-chip in a configuration memory, we simply use those configurations as the configurations to test our SUTs, thus avoiding costly changes to the architecture. Although these configurations are not guaranteed to detect all faults, there is no need to. We only need to detect those faults that would cause our current application to function incorrectly. With our BIST testing for the existence of "applicable" faults only, we refer to our scheme as Built-In Applicable Self-Test, or BIAST.

4. Evaluation

Before we evaluate the performance of BIAST on PipeRench, let us first introduce some terms and variables.

We define a *configuration cycle* as the time it takes to run the desired number of test patterns on one configuration of the SUT. Every configuration cycle consists of the following steps: reconfiguring the SUTs, reroute, write, and compare stripes once, generating test patterns, and comparing results.

The SUTs are repeatedly reconfigured until all the desired stripe configurations have been used and all test vectors have been applied, in what we define as a *stripe-test cycle*.

The time to perform a stripe test cycle on all the physical stripes of the device is referred to as a *test cycle*.

- N_v - number of virtual stripes in an application
- N_p - number of physical stripes available to the application
- N_d - number of data vectors the LFSR will generate
- P_t - percentage of cycles to be devoted to test
- T_s - time to complete one stripe test cycle
- T_t - time to complete one test cycle, i.e. to completely test the physical fabric for the current application

4.1. Cost of Testing

The cost of a test cycle is dependent on the number of test patterns used for each configuration, and the number of configurations tested on each SUT. The number of test patterns (N_d) to use varies. By using a synthesizable verilog model of the LUT, and performing fault coverage analysis using Synopsys Test Compiler, we determined that 56 test patterns would be needed to attain full coverage of all detectable faults. In our test model, the inputs to the PEs were 8-bit quantities; we used 8-bit LFSRs configured to produce 63 distinct vectors.

The first configuration cycle requires 11 cycles initially to configure each of the stripes in the test block, and N_d cycles to run all the vectors through the SUT. After the first configuration cycle, it takes six cycles to reconfigure the reroute, SUT and global bus read/write stripes. Then N_d cycles are needed to run through all the test vectors. This is done N_v-1 times. Putting this information together gives the following figure for time per stripe-test cycle (T_S):

$$N_v (N_d + 6) + 5 = T_S \qquad \text{(EQ 1)}$$

The time for a test cycle (T_t) can then be easily determined. With each stripe-test cycle covering two physical stripes, N_p / 2 stripe-test cycles would be needed to ensure that every physical stripe is tested:

$$(N_p /2) (N_v (N_d + 6) + 5) = T_t \qquad \text{(EQ 2)}$$

For an example, IDEA, the cryptosystem used in PGP, takes 177 stripes when run on a 16-stripe 100MHz chip. The test cycle time for complete coverage of applicable modes of operation is (16/2)(177(56+6)+5) = 87,832 cycles = 878.32 µs.

4.2. Mean Time to Detect a Fault

Since the total time to exhaustively test all physical stripes is known, it is possible to compute the Mean Time to Detect a Fault. If each stripe is equally likely to contain a fault, half of the stripes will have to be tested, on average, before a fault is detected if one exists [1]. Conservatively, this means than the MTDF can be expressed as follows:

$$((N_p /2) (N_v (N_d + 6) + 5)) / 2 = \text{MTDF} \qquad \text{(EQ 3)}$$

However, MTDF is only really meaningful while running another application, as it will correspond roughly to the number of erroneous outputs that can occur before the fault is detected. To compute the MTDF of a device that divides its time between running an application and executing BIAST, we must first define the percentage of time devoted to test, P_t. Then:

$$(((N_p /2) (N_v (N_d + 6) + 5)) / 2)/P_t = \text{MTDF} \qquad \text{(EQ 4)}$$

For any given application running on the current version of the PipeRench chip, N_d=56 (as determined above) and N_p=16. Thus, the MTDF can be computed in terms of N_v and P^t.

$$(248N_v + 20)/P_t = \text{MTDF in current PipeRench} \qquad \text{(EQ 5)}$$

In Figure 4, we show the case where N_v ranges from 10 to 160, and P_t ranges from 5% to 50%. For IDEA, assuming 10% of our cycles spent in test, the MTDF is (248*177+20)/0.1 cycles = 4.3916 ms. If the application designer decides this is too long and decides to increase the P_t to 30%, the new MTDF is 1.4639 ms.

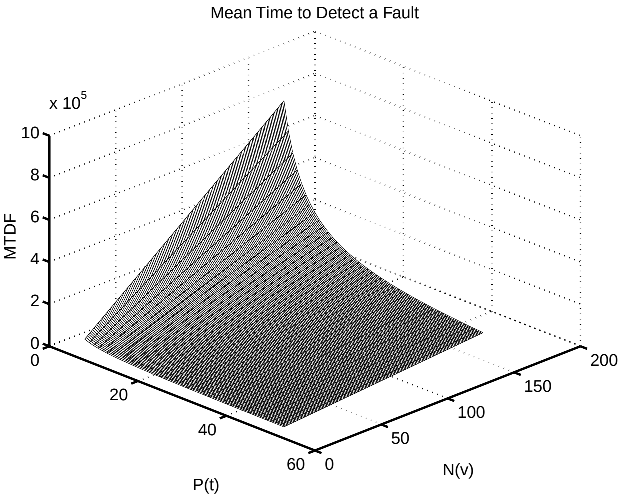


Figure 4.

MTDF in cycles as a function of percentage of cycles devoted to test (P_t) and number of virtual stripes (N_v)

These figures only apply to the detection of the presence of a fault. If and when that occurs, another sequence of tests is run to determine which part of the hardware has failed. This sequence takes approximately 450 cycles to run and is described further in the Section 5. Emphasis at that point however is not on application performance, but on isolating the fault and returning the device to working order as soon as possible.

4.3. Tunable Fault Testing

From the equations presented in the previous subsection, it is clear that the MTDF scales linearly in each of its factors. This makes it easy for users to specify the amount of fault tolerance for the system, based on how often they wish to test, how many test vectors they wish to apply, or how many virtual stripes their applications use.

When the chip is powered on and configured, the complete test cycle can be run before the application is allowed to begin executing, to ensure proper initial operation. Thereafter, a percentage of the total cycles can be devoted to running an ongoing BIAST cycle that goes through complete test cycles as the program executes. The configuration controller is responsible for ensuring that all physical stripes are used as SUTs, moving the BIAST stripe block so that stripes that have already been testing in the current test cycle will not be tested again

[1] A fault can be detected in any one of the 12 stripes used to test the two under test. Thus, there is a reasonable chance that any fault in a stripe will be discovered before that stripe is actually the stripe under test.

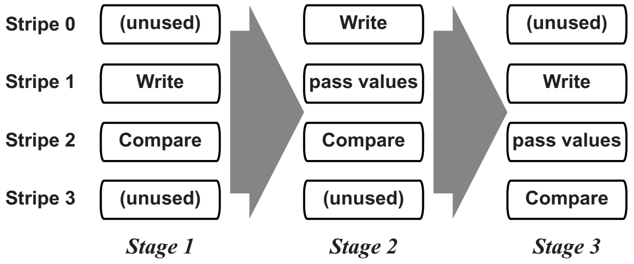


Figure 5.

Interstripe Interconnect test: 3-step fault isolation scheme Interconnect between Stripes 1 and 2 under test.

5. Fault Isolation

The BIAST algorithm detects the existence of faults in a device, but it does not locate the point of failure. A detected fault can originate in any of the stripes used in the test. An additional suite of tests designed to isolate the fault comprise a second stage of BIAST. First, the integrity of the interstripe interconnects is tested, followed by the global bus, the LFSR stripes, and the reroute stripes. If none of these is faulty, then the SUTs and the write and compare stripes are tested. The intention is to eliminate suspect areas one by one, using the resources of each cleared component to help test then next.

The interstripe interconnect test serves as a good example of all the tests. As shown in Figure 5, Stage 1, there are initially two stripes involved in testing, one to stimulate the interstripe interconnect being tested with data (Stripe 1), and one to verify the value of the data after it has been propagated through the interstripe interconnect (Stripe 2). It is assumed that both stripes will not have faults that manifest themselves in such a way as to exonerate a damaged interconnect for all stimuli. Thus, if there is a positive comparison for all stimuli, the interconnect is deemed working, and the next interconnect is tested. If an error is detected, the test moves to Stage 2 and an alternate source of stimuli is used to test the interconnect (Stripe 0) so as to ensure that the previous source (Stripe 1) was not the cause of the error. If the error is no longer present, we determine that Stripe 1 is the cause of the error, and mark it as such. However, if the error is still present, the test moves to Stage 3, in which we use an alternate compare stripe (Stripe 3) to see if the original stripe (Stripe 2) was the cause of the error. If the error does not resurface, we know that the original compare stripe had the defect. Should the error persist, we determine that the interconnect under test is faulty.

The general scheme of each test is the same. Configure two stripes, one to stimulate the resource under test, and one to check for the expected results. If an error is detected, use an alternate stripe to determine the possible culprit. Note that this scheme assumes a stripe configured to check for expected results (Compare) and its alternate are not both damaged at the same time. A scenario where each has a fault will cause the wrong resource to be marked as damaged. Setting aside additional alternate compare stripes may alleviate the problem.

It is good policy to rerun the whole BIAST structure after a fault is isolated and dealt with to ensure that no additional faults exist.

6. Fault Tolerance

Once a fault is isolated in a device, it is necessary for the device to be able to work around the error to be of any future use. Most damage can be mended. However, there are types of faults that cannot be tolerated. The faults that can be tolerated fall into two separate categories: non-interstripe interconnect faults, and interstripe interconnect faults.

Faults that are unrecoverable at the present time include faults in the configuration bus and global bus. Also included is the multiple fault scenario previously mentioned, in which two stripes are damaged, the damage manifests itself when either is configured to verify correct output of a resource under test, and one stripe is the alternate verifying stripe of the other. In this case, faults may be continually misdiagnosed until the entire chip has been declared damaged.

The classes of recoverable faults are quite considerable, however. Any single fault scenario (single stuck-lines, bridging faults) in which a PE, stripe, intrastripe interconnect or interstripe interconnect is damaged can be detected and is recoverable, as well as multiple fault scenarios not specifically alluded to in the previous paragraph.

6.1. Non-Interstripe Interconnect Faults

Stripes represent the smallest autonomous logic units in PipeRench. Faults occurring in parts of a PE (e.g., LUT, barrel shifter) will render the PE unusable. Without an extra PE through which to redirect all the interstripe and intrastripe interconnect lines of a faulty PE, the computational power of the stripe will be lessened to the point where the stripe must be disabled. As long as the interstripe interconnect works, however, the contents of the register file from the previous stripe can be passed through the damaged stripe to the next good stripe.

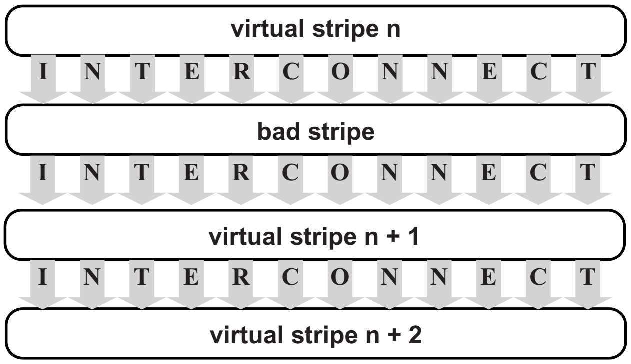


Figure 6.

Non-interstripe interconnect fault work-around. "bad stripe" is configured to pass register values through unchanged. Configurations and input delayed one cycle.

For non-interstripe interconnect errors, the damaged stripe is set to pass the values in its pass registers through to the next stripe unchanged (See Figure 6). Any input or configurations meant for that stripe are delayed one cycle and input into the next stripe. Thus, the performance of the application is degraded by $1/N_p$.

6.2. Interstripe Interconnect Faults

If an interstripe interconnect line is faulty, the pass registers cannot all move through to the next stripe. To illustrate this, imagine a 16 PE/stripe, 8 register/PE PipeRench chip. There are 128 registers per stripe, and 128 interstripe interconnect lines to move the values of those registers to the next stripe. If one of the interstripe interconnect lines contains a fault, there are not enough lines to move all the register values through.

Applications interstripe interconnect fault tolerance requires hardware or compiler support to restrict user access to all but one register in each PE. This restriction frees an interstripe interconnect line and register in each PE that can then be used in the event of interstripe interconnect damage. With this change, should an interstripe interconnect line be broken, the value of the register affected can be redirected into the unused register in the stripe before the damaged interconnect, and can be directed back into its initial position in the stripe following the broken interconnect (See Figure 7). In this way, chip functionality is maintained, although at a cost of a $3/N_p$ degradation in performance, so long as there is not more than one bad interconnect line to any one PE. For greater fault tolerance, additional changes are necessary to allow other registers the accept redirected data.

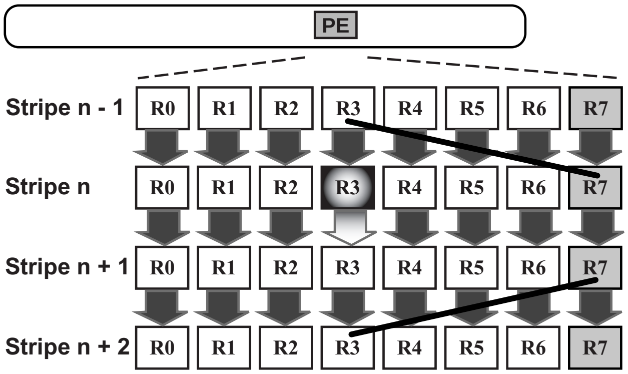


Figure 7.

Interstripe interconnect fault work-around. PE x's register file is shown. R3 interconnect of Stripe n is damaged. R3 values are rerouted through the free R7.

7. Related Work

There have been a few other research efforts that have focused on BIST designs in FPGA-style reconfigurable hardware [1],[2],[6-12]. Those BISTs were designed for industry standard Xilinx, Lucent, and Altera FPGAs, which lack the virtualized pipeline of PipeRench. It is precisely this feature, the ability to break an application up into discrete pipeline stages, that allows us the ability to tune the amount of fault tolerance.

The BISTs designed for industry standard architectures were able to scale in constant time, in contrast to the linear scaling of the BIST implemented on PipeRench. This was achieved, however, by reducing the test coverage of the interconnect between programmable logic blocks (PLBs), and increasing reliance on global buses to supply test data to PLBs. It is important to note as well that comparing the ability to scale in constant versus linear time is perhaps misleading, as the constant associated with configuration of industry FPGAs is usually quite large, when compared to the single clock cycle reconfiguration time for PipeRench

Additionally, the configurations used to test the PLBs have, in the past, been either generated pseudo-randomly or stored specifically for testing. A greater number of configuration patterns were needed for fault coverage equivalent to stored configurations if the configurations were generated on the fly. If the patterns were stored, an increase in fault coverage meant an increase in memory to store the additional patterns. The new BIAST described in this paper uses the application configurations already stored on the chip as the test configurations. While this method does not cover all areas of operation for the chip, it does cover 100% of the configuration space that matters to the program at hand.

Past methods have tested program-irrelevant modes of operation of hardware, finding errors and mapping out hardware that may be reliable in program-applicable modes of operation [13].

8. Conclusions

In this paper, we have described a new built-in self-test algorithm and fault tolerant hardware model designed for runtime reconfigurable devices. The importance of fault tolerance is growing as people rely more heavily on computers. Reconfigurable hardware is one area of computing with great growth potential. It is already seeing increased use in areas such as tele-communications, and as the reconfigurable hardware industry continues to grow, so must its fault tolerance.

We found the PipeRench architecture to be a good platform for processing needs with fault tolerance requirements. PipeRench's hardware virtualization allows a frequently executing BIST, such as the one described in this paper, to be run concurrently with an application. The amount of time devoted to testing can be tuned at runtime. This is ideal for mission-critical applications where speed of execution is secondary to the requirements of data integrity.

We mentioned that the BIAST described can detect many classes of faults, and that the majority of failures have fixes that require few hardware changes. The ability to tolerate faults adds little delay. BIAST itself requires little additional hardware, allowing it to run at program clock speed.

By reusing the configurations already stored for the program running on PipeRench, we found that complete applicable fault coverage was obtained on a per-application basis with no additional memory or hardware beyond a minor amount of control logic.

The BIAST implemented on PipeRench represents a new testing model for hardware with space constraints, with a focus on efficiency and flexibility.

9. Acknowledgments

The authors would like to acknowledge the constant help and support of Herman Schmit, Matt Moe, Reed Taylor, and the rest of the PipeRench group. In addition, they would like to thank Shawn Blanton for his help with testability issues, and the reviewers of this paper for their helpful comments. This work was supported by DARPA contract DABT63-96-C-0083.

References

[1] C. S. Stroud, E. Lee, M. Abramovici, "BIST-Diagnostics of FPGA Logic Blocks," *Proc. Or the 1997 IEEE International Test Conference*, pp 539-547, 1997.

[2] C. S. Stroud, E. Lee, M. Abramovici, "Using ILA Testing for BIST in FPGAs," *Proc. Of the 1996 IEEE International Test Conference*, pp. 68-75, 1996.

[3] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *International Symposium on Computer Architecture*, Atlanta, GA June 1999. To be published.

[4] M. Meyers, K Jaget, S. Cadambi, J. Weener, M. Moe, H. Schmidt, S. C. Goldstein, D. Bowersox, *PipeRench Manual*, 1998.

[5] R. D. Blanton, S. C. Goldstein, H. Schmidt, "Tunable Fault Tolerance via Test and Reconfiguration," *Fault Tolerance Computing Symposium*, 1998.

[6] M. Meyers, "Testing of Pipeline Reconfigurable Machines," *M.S. Thesis*, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1998

[7] C. Stroud, S. Honala, P. Chen and M. Abramovici, "Built-In Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST Without Overhead!)", *The 14th IEEE VLSI Test Symposium*, 1996

[8] W. K. Huang and F. Lombardi, "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays," *14th IEEE VLSI Test Symposium*, 1996

[9] T. Inoue, et al., "Universal Test Complexity of Field-Programmable Gate Arrays," *Fourth Asian Test Symposium*, Los Alamitos, CA, 1995

[10] X. T. Chen, W. K. Huang, F. Lombardi and X. Sun, "A Row-Based FPGA for Single and Multiple Stuck-At Fault Detection," *IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, Lafayette, LA, 1995

[11] C. Jordan and W. P. Marnane, "Incoming Inspection of FPGAs," *Third European Test Conference*, Los Alamitos, CA, 1993

[12] K. Kwiat, W. Debany and S. Hariri, "Effects of Technology Mapping on Fault-Detection Coverage in Reprogrammable FPGAs," *IEEE Proceeding - Computers and Digital Techniques*, 1995

[13] G. A. Mojoli, et al., "KITE: A Behavioral Approach to Fault-Tolerance in FPGA-Based Systems," *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Boston, MA, 1996