Appear in 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2009
Chicago, Illinois, October, 2009

Analyzing Formal Verification and Testing Efforts of Different Fault Tolerance Mechanisms

Meng Zhang[1], Anita Lungu[2], and Daniel J. Sorin[1]

[1]*Dept. of Electrical and Computer Engineering*
Duke University
{mz28, sorin}@ee.duke.edu

[2]*Dept. of Computer Science*
Duke University
anita@cs.duke.edu

Abstract

*Pre-fabrication design verification and post-fabrication chip testing are two important stages in the product realization process. These two stages consume a large part of resources in the form of time, money, and engineering effort during the process [1]. Therefore, it is important to take into account the design verification (such as through formal verification) effort and chip testing effort when we design a system. This paper analyzes the impact on formal verification effort and testing effort due to adding different fault tolerance mechanisms to baseline systems. By comparing the experimental results of different designs, we conclude that re-execution (time redundancy) is the most efficient mechanism when considering formal verification and testing efforts together, followed by parity code, dual modular redundancy (DMR)**,** and triple modular redundancy (TMR). We also present the ratio of verification effort to testing effort to assist designers in their trade-off analysis when deciding how to allocate their budget between formal verification and testing. Particularly, we find even for a designated fault tolerance mechanism, some small change in structure can lead to dramatic changes in the efforts. These findings have implications for practical industrial production.*

1. Introduction

Pre-fabrication design verification (henceforth referred to simply as "verification") and post-fabrication chip testing ("testing") are two important stages in the product realization process. Verification ensures the correctness of the design itself, and testing ensures that the fabricated design is free of manufacturing faults. Although there are many verification methodologies, we focus on formal verification techniques in this paper. It is acknowledged that verification and testing consume a huge amount of resources during the whole process, increasing the cost as well as the time to market. As an example, the verification of the Intel Pentium 4 required a large team spending several years [1]. It has been estimated that about 60-70 percent of the development cost of a system is spent on verification and testing [2, 3].

One important factor leading to the difficulty and high cost of verification and testing is that often design engineers do not assign a high priority to a design's verifiability. In general, designers make every effort to improve performance and reduce cost through the features they propose. However, it is often the case that designers do not explicitly consider the consequent increases in verification and testing efforts of proposed design changes.

Recently, researchers have noted and started to address this problem by considering formal verification and testing efforts as constraints early in the design phase. In the context of cache coherence protocols, Martin [4] argues that directory protocols are superior to snooping protocols from the perspective of formal verification effort. Lungu and Sorin [5] illustrate that even some subtle change in the processor design can lead to significant differences in formal verification effort. Schuchman and Vijaykumar [6] present a microarchitecture which facilitates the testing and defect isolation with little performance penalty.

However, to our knowledge, no research has been conducted on how to take both formal verification effort and testing effort into account when designing a system. In this paper, we choose a particular design aspect of a system -- its fault tolerance mechanisms -- as our target. We analyze and compare several fault tolerant mechanisms from the perspective of formal verification effort and testing effort in order to identify the most economical choices. We have two baseline systems in our experiments. One is a first-in, first-out queue (FIFO) and the other is a three-stage pipelined datapath. Four fault tolerance mechanisms are added to the two baselines separately. They are DMR, TMR, parity code, and re-execution (time redundancy), which belong to three different fault tolerance methods and represent a wide variety of commonly used fault tolerance techniques. Then we analyze the increase of formal verification effort and testing effort in a system when adding these fault tolerance mechanisms to our baselines. We also provide the ratio of formal verification effort to testing effort which can be used when distributing a common shared budget between formal verification and testing. We make the following contributions in this paper:

● We compare the formal verification and testing efforts for validating the baselines to the efforts required for validating the baselines augmented with the fault tolerance mechanisms. In the context of formal verification effort, re-execution causes the least increase in effort, followed closely by DMR and TMR. Adding fault tolerance through a parity code causes the largest increase in formal verification effort. For testing, re-execution and parity code are both economical. DMR causes a moderate increase, while TMR leads to a very large increase.

● We show that when considering both formal verification effort and testing effort, re-execution and parity code are good choices. We also present the ratio of the two efforts. Designers can use this information in their trade-off analysis when deciding how to allocate the validation budget available for formal verification and testing.

● We further find that for a particular fault tolerance mechanism, even a small design change can reduce the verification effort dramatically, which motivates us to think more about the design structure.

The conclusions we draw in this paper, combined with dependability and cost targets, can assist designers at early stages of design in deciding which fault tolerant mechanisms to add to their system so as to minimize the resulting added formal verification and testing efforts.

In the rest of this paper, we first describe the reasons for choosing fault tolerance mechanisms as our target (section 2). Then we present the system models in section 3. Section 4 describes our verification and testing methodologies, and section 5 presents the experimental results. Finally we conclude in section 6.

2. Motivation

The impact of adding fault tolerance mechanisms to different system components has been analyzed in the context of area, runtime, and power overheads [7, 8]. However, as mentioned, another very large cost in the creation of a new product is verifying a system’s correctness in terms of design (for example through formal verification) and manufacturing (through testing). Knowing the correlation between a particular fault tolerance mechanism and the additional verification and testing efforts it involves might prove useful information to designers. To our knowledge, there is limited understanding of the impact that adding different fault tolerance mechanisms to a system has on formal verification and testing overheads. This motivates us to pursue such a topic in this paper and probe the quantitative relationship between fault tolerance mechanisms and formal verification and testing efforts.

There are three components to the verification and testing overheads generated by adding fault tolerance mechanisms that we investigate. The first overhead is the additional verification effort spent in ensuring the functional correctness of the system in the presence of fault tolerance (i.e., the system with fault tolerance produces the expected results). The second overhead is the additional effort to verify the correctness of the fault tolerance mechanisms themselves (i.e., they do indeed tolerate faults and do not introduce false positives or false negatives in the system). The third overhead is the additional testing required to test the fault tolerance mechanisms. The results of our analysis can be used by designers such that they choose fault tolerance mechanisms based on not only the area overhead, power consumption, but also on the additional formal verification and testing efforts involved.

3. System models

3.1 Baseline systems

It is our goal that our results of the comparison among different fault tolerant solutions from the perspective of formal verification and testing efforts are fairly general and not limited to particular designs. Thus, each baseline should have at least four characteristics: 1. It should represent a wide range of systems (i.e., it should not be an unusual or extremely quirky system); 2. It can be augmented with several different types of fault tolerance mechanisms; 3. It can be easily tested and formally verified with and without fault tolerance mechanisms; 4. It should not involve such a large amount of states that formally verifying the system leads to state explosion.

Based on these requirements, we did experiments on two baseline systems that are derived from McMillan's tutorial for the Cadence SMV tool [9]. One baseline is a FIFO queue. Figure 1 shows a FIFO implemented as a circular buffer. This FIFO implementation also provides handshake signals, “empty” and “full”, that indicate the occupancy state of the FIFO queue.

Our second baseline system is a three-stage pipelined datapath (“PD” for short). The three stages are: operand fetch, arithmetic/logic operation**,** and result writeback. Figure 2 illustrates the functions of the three stages. We only implement one instruction (XOR) in this PD, for the sake of simplicity. The choice of the XOR instruction is arbitrary; we only need to ensure that the selected function is similar in complexity to a variety of other functions.

We believe that our baselines, although simple, are representative of common system components, and the conclusions made based on these baselines can hold true in a wide range of designs. However, we are not making the claim that all possible designs will strictly conform to these conclusions.

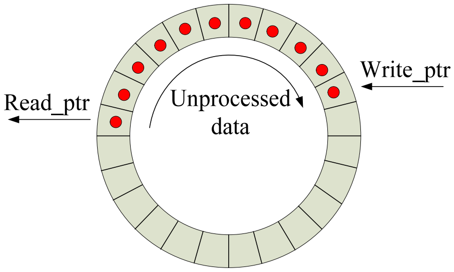

Figure 1. Circular buffer (FIFO)

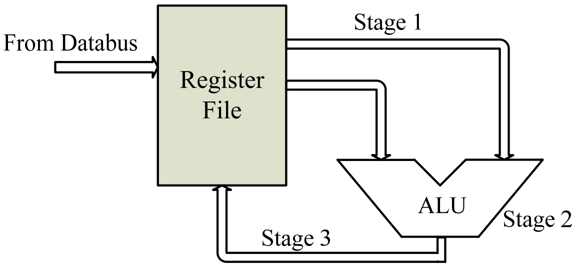

Figure 2. Pipelined datapath

3.2 Fault tolerance mechanisms

We added four fault tolerance mechanisms to each baseline. They are: DMR, TMR, parity code and re-execution. The four mechanisms belong to three types of fault tolerance methods. DMR and TMR fall into *physical redundancy* since they ensure fault tolerance by replicating the module of interest twice or three times and assessing the agreement of the results. Both DMR and TMR can detect errors in the modules by comparing the results. Additionally, TMR can correct single errors by voting. A parity code is a form of *information redundancy*. It adds an extra bit to a set of target bits to ensure that the number of “1” across all target bits remains even or odd. Parity code is the simplest error detection code. Re-execution falls into *time redundancy* by doing the same thing twice at different times. Time redundancy methods can be used to detect transient faults.

4. Verification and Testing Methodologies

Verification and testing are two indispensable steps in the product realization process which, although sometimes confused, are different in their goals and methodologies. The goal of verification is to ensure that the design is correct; the goal of testing is to ensure that the chip’s design is manufactured correctly [10].

Verification can be accomplished through system simulation or formal verification. Simulation is an incomplete verification solution that verifies only a subset of all possible system states. In contrast formal verification exhaustively verifies all possible system states and proves the absence of design bugs. Because of its completeness we chose formal verification as our verification approach.

Testing checks the correctness of the manufactured product. It includes two stages. First, certain software is used to generate tests for the design. Then, ATE (Automatic Test Equipment) is used to apply the tests to the design. We now describe the methodologies we use for performing formal verification and testing of the designs we analyze.

4.1 Methodology for formal verification

There are two main approaches to perform formal verification: model checking and theorem proving. *Model checking* automatically and exhaustively traverses the reachable state space of a design and is limited by the state explosion problem. Because model checking implies an exhaustive state space traversal, the resources used in the process (memory, run-time) quickly become a limiting bottleneck. *Theorem proving* involves a mathematical, largely manual proof of a design’s correctness and is limited by the high manual effort.

In our work we chose a prevalent verification tool, Cadence SMV (called SMV in the rest of this paper). SMV is a model checker that is augmented with a restricted theorem proving layer which implements a number of features that alleviate model checking’s state explosion problem. One such a feature is called *cone of influence reduction.* This feature implies that when verifying the truth value of a particular system property, SMV involves only the states in the property’s cone of influence (that has the potential to affect the property’s truth value) as opposed to the entire system states. SMV is also proficient at composite verification, which involves breaking large properties into small verification sub-goals that are verified independently, and then ascertaining correctness of the entire system from the correctness of the sub-goals [9]. Another feature that is particularly useful in SMV is its ability to formally verify (after an internal transformation in the theorem proving layer) infinite state machines through model checking. We use this feature to formally verify the correctness of a FIFO transmitting an infinite amount of data (as opposed to only verifying that the FIFO transmits correctly a fixed amount of data). With the help of the same SMV feature we can formally verify a system with an infinite number of registers in the pipelined datapath, which also increases the generality of the verified design.

The user of SMV provides the tool with two inputs. The first input is the description of the design under verification as a finite or infinite state machine. The second input is the description of the correctness properties of the system in temporal logic or as invariants. Performing the formal verification task with SMV means that the tool automatically and exhaustively traverses all reachable states of the system and ensures that no deadlock states are found and that all correctness properties are obeyed.

Correctness properties for baseline systems: For the FIFO, we specify two properties. Our first property states that the data output from the FIFO is the same as the data inserted into the FIFO. Our second property asserts that the first-in, first-out invariant is maintained. These two properties are verified separately. For the pipelined datapath system, we also specify two separate correctness properties. One guarantees that the operands fetch is performed correctly; the other guarantees that the ALU does the operation correctly.

Correctness properties for fault tolerance mechanisms: After adding fault tolerance mechanisms to our two baseline systems (FIFO and PD), the number of our correctness properties is increased. In addition to verifying the functional correctness of the baseline systems we also want to ensure that the fault tolerance mechanisms work, in that they correctly detect or correct errors as intended. These new features require us to add some new correctness properties which guarantee the absence of any false positive or false negative. Particularly, for TMR, a new property has to be specified to verify that the three modules in the design can vote to give the right output, which actually corrects the error.

Metric for quantifying formal verification effort: Choosing proper metrics for estimating formal verification effort is not a straightforward topic [5]. This is because different formal verification methods have different limitations. For example, theorem provers are mostly limited by the manual effort necessary to complete the proof, while model checkers generally run out of memory necessary for representing and keeping track of the entire reachable state space of the system. The metric chosen to quantify formal verification effort should capture and quantify the characteristic that is the limiting factor when applying formal verification.

We chose to focus on model checking methods for formal verification (due to the advantage of automation compared to theorem proving), and the metrics we select to measure formal verification difficulty reflect this choice. Every model checker uses a data structure to represent the state space that it must traverse (e.g., SMV uses ordered binary decision diagrams). The size of that data structure and time required to traverse it depend fundamentally on the size of the system's reachable state space (unreachable states often have less impact on verification effort) and secondarily on the exact type of data structure used by the model checker. To keep our analysis general, we report only the size of the *reachable state space* and avoid reporting metrics that are tool-dependent.   More specifically, we report the verification effort, Ev, as the *largest* number of reachable states required to verify any property. This maximum value is more appropriate than the average number of reachable states (averaged across all properties), because it is the largest memory required to verify each of the properties that determines whether a system can be formally verified, not the average memory requirements.

4.2 Methodology for testing

The methodology for design testing is more standardized than the one for formal verification. We followed the common steps involved in testing a system for all the designs that we were interested in (FIFO and PD as baselines each augmented with the four fault tolerance mechanisms). First, we used the hardware description language to describe our designs and performed the true value simulation. Second, we converted the register transfer level (RTL) description to a gate level description using Synopsys' synthesis tool, Design Compiler. Our target library in Design Compiler was Cadence GSCLib3.0, which uses 180 nm technology. To achieve acceptable fault coverage in testing, we needed to increase the controllability and observability of flip-flops. This was done by changing all ordinary D flip-flops to scan flip-flops using extra circuit logic, and then forming a scan chain. Note that this change does not influence the behavior of the circuit when it is not being tested. Finally, we used Mentor's ATPG tool, FastScan, to generate test patterns for our designs. We found the results of different fault types and fault numbers were similar, so we chose single stuck at-fault for our analysis.

Metric for quantifying testing effort: Our goal is to obtain a single metric, Et, that captures the effort required for testing. There are three challenges in obtaining this metric. First, we must consider fault coverage. If the fault coverage is too low or if it varies considerably for different versions of a design (e.g., between a baseline and the baseline when augmented with a fault tolerance mechanism), then the comparison of testing efforts is either uninformative or unfair. We will show in Section 5 that the fault coverages of a certain baseline and its augmented designs are almost the same, which provides for fair comparisons. Second, our metric, Et, must incorporate all of the important aspects of the cost of testing. These costs include: the time the CPU spends on generating the test; the time the ATE needs to complete the test; the number of test patterns; the test vector length; and the power consumption of performing the test. Third, we must be careful that Et does not "double count" certain aspects of cost that are highly correlated and originate from the same characteristic of the design. For example, CPU time is proportional to the test data volume. Also, power consumption is proportional to the data volume (which we will experimentally confirm in Section 5). We exclude these correlated metrics from Et.

Thus, to obtain Et, we multiply the number of test patterns and the test vector length. Both of these metrics can be derived from the statistical results of FastScan. When confirming that power consumption was highly correlated with these metrics, we estimated the power consumption by counting the transitions, including both 0→1 and 1→0, in the test patterns.

5. Experimental results

We divide this section by presenting the impact of adding fault tolerance mechanisms separately on formal verification effort (section 5.1) and on testing effort (section 5.2). Then in section 5.3, we provide integrated results for formal verification and testing efforts. In section 5.4, we describe a further experiment on a certain fault tolerance mechanism.

5.1 Experimental results for formal verification

Recall from Section 4.1 that our metric for estimating the effort required for formally verifying a system, Ev, is the *largest number of reachable states* among all correctness properties. In Figure 3 we show this metric on the y-axis (note the logarithmic scale) for different designs normalized to the baselines. The left columns represent the FIFO baseline and the fault tolerant FIFO designs, and the

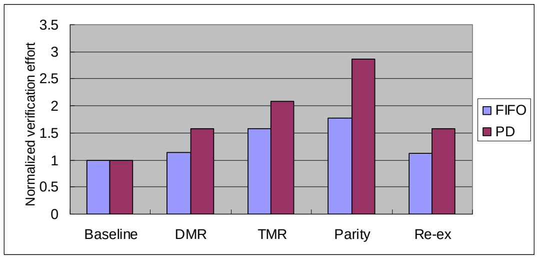

Figure 3. Normalized verification effort (Ev)

right columns represent the PD baseline and fault tolerant PD designs. We observe similar trends for both the FIFO and PD baselines, which is encouraging from the perspective of reaching general results.

Based on the results in Figure 3, we conclude that adding re-execution as a fault tolerance mechanism causes the least formal verification effort increase compared to the baseline without fault tolerance, followed by DMR. TMR causes a moderate effort increase, and the parity code causes the largest effort increase. For re-execution and DMR, each property we need to verify involves just a small number of state variables in its cone of influence. This result may seem surprising for DMR because DMR involves a significant addition of system state (for the extra module), yet the formal verification effort does not seem to increase much. The explanation is that a large number of state variables have no relationship with the truth value of a particular correctness property and are thus excluded from the verification process. For TMR, the verification effort increases for the property that checks the correctness of voting logic because this property involves states from the original and replicated components. For the parity code, the parity bit is a XOR of the eight bits. The property that checks the effectiveness of the parity bit thus involves more states in its cone of influence (from the 8-bit XOR), which leads to a sharp increase in the formal verification effort.

5.2 Experimental results for testing

In Section 4.2, we explained our metric, Et, for estimating the effort required to test a system. Using this metric was predicated on two assumptions that we now validate before presenting the results for Et. First, we needed to show that the fault coverage for the baseline is similar to that for the baseline augmented with fault tolerance mechanisms. In Table 1, we confirm that this assumption is true for both baselines we consider in this paper.

Our second assumption in Section 4.2 was that the components of Et were uncorrelated. We chose to exclude power consumption because it was expected to be highly correlated with the other aspects. Figure 5 shows the power consumption results. If we compare these results to the Et results in Figure 4 (discussed next), we see that they are nearly identical, thus justifying our assumption.

We now present our Et results. Et (Figure 4) is the product of the number of test patterns and the test vector length. This figure shows results that are normalized to the baselines. We observe that the two baselines have slightly different results. For the FIFO, re-execution leads to the least effort increase, then parity code, DMR and TMR. For the pipelined datapath, parity code leads to the least effort increase, then re-execution, DMR and TMR. The insight behind our results is that from the perspective of logic circuit overhead, re-execution and parity are very economical. Re-execution just adds some extra buffers to preserve the previous values, and the parity code just adds an extra bit for error detection. However, with DMR and TMR, there are redundant modules in the circuits and these modules also need to be tested, which greatly increases the test vector length and doubles or triples the testing efforts.

Table 1. Fault coverage of different designs

Type of Design	Fault Coverage
FIFO	93.06%
FIFO_DMR	92.73%
FIFO_TMR	93.14%
FIFO_Parity	93.04%
FIFO_Re-execution	93.05%

Type of Design	Fault Coverage
PD	78.37%
PD_DMR	77.01%
PD_TMR	79.18%
PD_Parity	80.70%
PD_Re-execution	80.67%

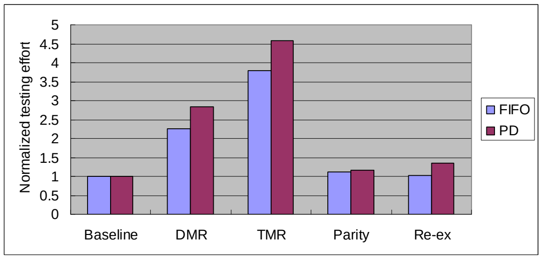

Figure 4. Normalized testing effort (Et)

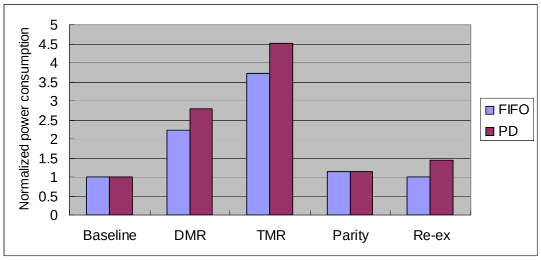

Figure 5. Normalized power consumption

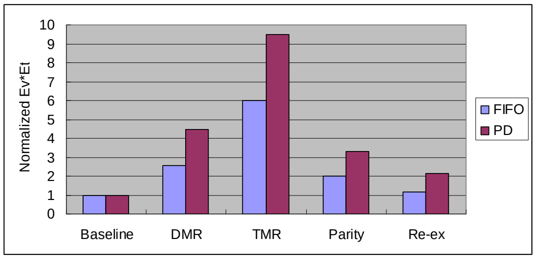

Figure 6. Normalized product of Ev and Et

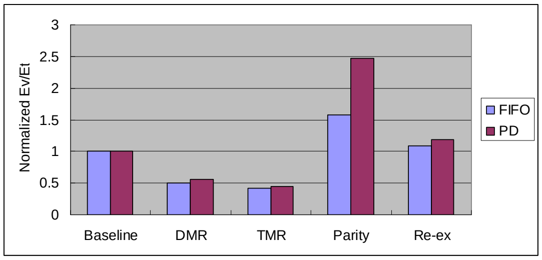

Figure 7. Normalized ratio of Ev to Et

5.3 Combining formal verification effort and testing effort

We are interested in the impact of adding fault tolerance methods on both formal verification effort and testing effort at the same time. This integration can be done in two ways. One way is to multiply the formal verification effort (Ev) by the testing effort (Et), which is shown in Figure 6. This result provides an intuitive understanding of which kind of fault tolerance mechanism is the most economical in regard of both formal verification and testing efforts. According to the figure, re-execution and parity are both efficient. DMR ranked the third, and TMR causes the largest effort increase.

The second way to integrate our results is to compute the ratio between the two efforts. Ev divided by Et is shown in Figure 7. These ratios can provide us with insights into which one of formal verification and testing is relatively more difficult to accomplish for a particular design. We normalize Ev and Et of the baselines to one. From this figure, we conclude that the parity code is relatively difficult to verify compared to how difficult it is to test, and TMR is relatively difficult to test compared to how difficult it is to verify. These ratios between efforts would prove valuable to designers when a single fixed budget allocated for both formal verification and testing needs to be divided between the two.

5.4 Further experiment with DMR

For DMR, we did a further experiment to compare two different designs. One design is DMR with a check per bit, which means the comparator compares every bit in the transmitted byte. The other design is DMR with a check per byte, meaning the comparator compares every byte and will activate the error signal if the byte contains any erroneous bit. Figure 8 and 9 shows the differences in efforts when using these two different DMR designs. In both figures, the testing efforts for DMR_bit and DMR_byte are almost the same, while the verification efforts change dramatically, although the two designs have the same function and nearly the same structure. This is an important implication. It indicates that even for a designated fault tolerance mechanism, if we make some small change in the structure, the required efforts may be quite different. Actually, this finding is in accord with guideline #1 presented by Lungu and Sorin in [5], which encourages designers to “construct the system such that it can be decomposed into small components that can be verified independently.”

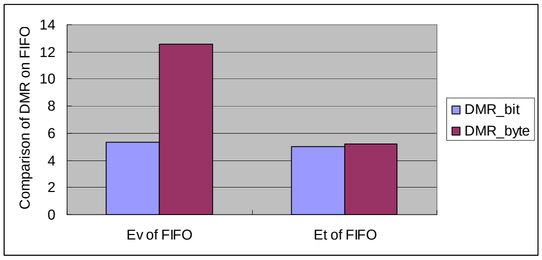

Figure 8. Comparison of DMR (FIFO)

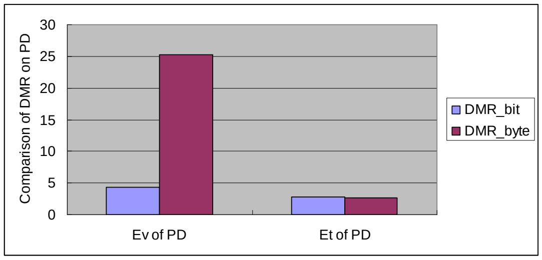

Figure 9. Comparison of DMR (PD)

6. Conclusion

In this paper, we experimented and analyzed the increase in formal verification effort and testing effort when adding four different kinds of fault tolerance mechanisms to our two baseline systems: a FIFO and a 3-stage pipelined datapath. Our results show that re-execution and parity codes cause very little increase in terms of the combination of the two efforts. DMR leads to a moderate effort increase. TMR causes the largest increase. Our analysis of the ratio between the formal verification effort to testing effort can also give designers insight into how to do trade-offs between formal verification and testing when they have a fixed budget shared between the two. Moreover, we find that for DMR, if we compare every single bit instead of every byte, the formal verification effort decreases a lot, reminding us of the importance of decomposable system structure. Although the final design decision will include more factors than the above mentioned, such as dependability and cost targets, these conclusions can be used as guidelines when designers need to choose fault tolerance mechanisms for their systems and are concerned with the increase in formal verification and testing efforts.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant CCF-0811290, an equipment donation from Intel Corporation, and a Warren Faculty Scholarship. The authors thank Krishnendu Chakrabarty for his feedback on this work.

References

[1] Bob Bentley and Rand Gray. Validating the Intel® Pentium® 4 Processor. *Intel Technology Journal Q1*, 2001.

[2] P. Bose, D. H. Albonesi, and D. Marculescu. Guest Editors' Introduction: Power and Complexity Aware Design. *IEEE Micro*, pages 8-11, Sept/Oct 2003.

[3] R. Hum. How to Boost Verification Productivity. *EE Times*, January 10 2005.

[4] Milo M. K. Martin. Formal Verification and its Impact on the Snooping versus Directory Protocol Debate. *International Conference on Computer Design (ICCD)*, October 2005.

[5] Anita Lungu and Daniel J. Sorin. Verification-Aware Microprocessor Design. *Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2007.

[6] Ethan Schuchman and T. N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

[7] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The Cost of Recovery in Message Logging Protocols. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 2, pp. 160-173, March/April 2000.

[8] Antonio Pullini, Federico Angiolini, Davide Bertozzi, and Luca Benini. Fault Tolerance Overhead in Network-on-Chip Flow Control Schemes. *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design*, pp. 224-229, 2005.

[9] K. L. McMillan. Getting Started with SMV. User's Manual, Cadence Berkeley Laboratories, USA, 2001.

[10] Michael L. Bushnell, Vishwani D. Agrawal. Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits. Springer, 2000.