

Functional Verification Acceleration through the Removal of Redundant Simulation

Jesse Craig

IBM Systems & Technology Group
Burlington, Vermont
jecraig@us.ibm.com

ABSTRACT

The real-world challenge of verification is to meet a design's testing requirements as quickly as possible without inhibiting the completeness of the verification effort. This paper introduces a new method, called the reduction of redundancy (ROR), which enables this goal. The stochastic nature of today's leading functional verification technique, directed random verification, can lead to wasteful redundancy during simulation because of a bias toward scenarios involving few stochastic decisions. By converting stochastic decisions into exhaustive ones, and using checkpointing to replicate the verification environment as needed, the ROR method works to defeat this bias. A brief look at the theory underlying this method is given and experimental results are shown for its application in real-world verification scenarios.

Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids – Verification

General Terms

Algorithms, Performance, Verification.

Keywords

Functional Verification, Directed Random Verification, Simulation Checkpointing.

1. INTRODUCTION

The market's race for "the next big thing" has insured that each new generation of integrated circuits contains ever increasing function. The transistor scaling that makes this increase possible has also increased the cost of fabrication, especially for one-time costs like lithography masks. Couple the increase in function with the rising fabrication costs and the need for better verification techniques becomes clear. These techniques need to provide a higher level of performance without sacrificing design reliability.

Recently, a substantial amount of verification-related research has focused on formal methods. While formal methods hold the promise of eclipsing traditional verification methods [1], their

inherent scaling problems have prevented this from occurring. Until these challenges are overcome, the bulk of the verification burden still rests on simulation-based techniques [2]. The leading functional verification technique in the industry today is directed random verification (also called constrained random verification or model based test generation).

At the core of the directed random verification technique is a stochastic method for generating tests based on a model of the hardware design being tested. This model is used to randomly generate testcases until a specified number of conditions, known as coverage goals [3], have been tested. The power of this method comes from its ability to automate testcase generation. A verification engineer writes one or more master verification tests in which the functions of the design-under-test to be verified are determined by stochastic decisions. Each master verification test can then be executed multiple times, with each execution testing different permutations of functions (i.e. a testcase). The more executions of a master verification test, the more coverage goals achieved. In this way, the task of generating testcases is automated, greatly reducing the amount of time and effort needed to verify a design [4]. Unfortunately, the same stochastic decisions that give this method its power also lead to significant amounts of duplication in the testcases generated, resulting in redundant verification.

In traditional directed random verification, the probability of a testcase executing a specific outcome of a randomly-controlled condition (i.e. a stochastic decision) is independent of the previous testcases. Meaning that for a given outcome, A , the chance of it occurring during the execution of n testcases is defined as:

$$P(A_1 \cap A_2 \cap \dots \cap A_n) = P(A_1) \cdot P(A_2) \cdot \dots \cdot P(A_n)$$

This fact leads to an interesting conclusion; there is no guarantee of uniqueness for the testcases generated by a master verification test. Without a guarantee of uniqueness it is inevitable a testcase will verify conditions previously verified. Verifying identical conditions multiple times on a deterministic machine is redundant. Worse yet, as the number of executed testcases increases, the number of redundant conditions executed increases exponentially. This increase in redundancy is credited with causing the diminishing returns often seen in directed random verification [5].

This paper introduces a technique, referred to as reduction of redundancy (ROR), which works by converting some or all of a master verification test's randomly-controlled conditions into exhaustive ones. Instead of choosing an outcome at random, the new exhaustive condition explores every possible outcome. This exhaustive execution is made possible by a technique known as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference '07, Month 1–2, 2007, City, State, Country.
Copyright 2007 ACM 1-58113-000-0/00/0004...\$5.00.

checkpointing. Checkpointing allows a program’s running state to be captured in persistent storage and restored later. Capturing the program’s state removes the need to re-execute the portion of the testcase leading up to the checkpointed condition for the exploration of each outcome.

This paper describes two experiments that were conducted to show the performance gains of the ROR method. The first experiment verifies the function of an embedded processor design, while the second verifies a bus bridge design based loosely on the PCI-Express root-complex specification [6]. Both experiments are run with and without use of the ROR method, comparing the simulation time needed to achieve the coverage goals of each design.

The remainder of the paper is broken-down as follows. Section 2 describes directed random verification and summarizes previous work done to improve its performance. Section 3 focuses on the ROR method, giving details on its implementation and application. Section 4 describes the experimental results from the previously mentioned experiments. Section 5 concludes with general remarks and suggestions for further work.

2. BACKGROUND AND RELATED WORK

Functional verification is composed of four major steps: generating stimuli for a design, simulating the design’s reaction to those stimuli, checking the reaction against an expected result, and recording what coverage goals the stimuli tested. In this system, coverage goals act as a quantitative embodiment of the verification engineer’s testing requirements. Coverage goals enumerate the functions, combinations of functions, and parameters to those functions that need testing. Discrete events, such as executing a design feature in a specific way, are referred to as standard coverage goals. Permutations of these discrete events form a second class of coverage goals referred to as cross-coverage goals [3].

Directed random verification is a refinement of functional verification that utilizes a design specific random test generator to create stimuli. This is opposed to other techniques which rely on manual methods, or which create design independent random stimuli [7].

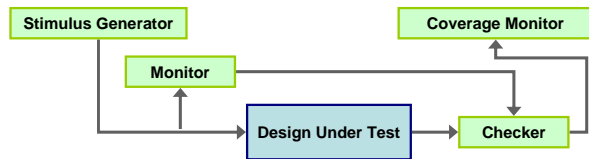


Figure 1. Direct Random Verification Environment

A directed random verification environment, shown in Figure 1, is composed of five major components. These components correspond to the four steps that compose functional verification: a stimulus generator (i.e. a master verification test) to create random stimuli, a monitor to observe the stimuli generated, a simulation of the design under test to generate the design’s reaction, a checker to compare that reaction to an expected result, and a coverage monitor to track which functions of the design have been tested. Stimuli are generated, simulated, checked, and their coverage recorded iteratively until all the coverage goals are achieved. Ideally this process consumes the smallest amount of

simulation time possible.¹ With this goal in mind, the efficiency of various directed random verification techniques can be compared using the verification performance metric, defined as:

$$\text{Verification Performance} = \frac{\text{\# of Coverage Goals Achieved}}{\text{Simulation Time Consumed}}$$

Previous work on increasing verification performance has concentrated on directing the stimulus generator to target unachieved coverage goals. These methods are known as coverage-directed test generation (CDG) and they can be divided into two major categories: feedback based methods, and externally driven methods. Feedback based CDG utilizes feedback from the coverage monitor to guide the generation of stimuli by modifying the probabilities of certain conditions occurring. Machine learning techniques have been used to learn relationships between stimuli and coverage goals by studying the results from previous testcases’ execution. Using this information the CDG methods are able to steer the stimulus generator to target certain coverage goals. Bayesian networks [8], genetic algorithms [9], and neural networks [10], have all been applied to this problem. Other feedback methods have used coverage results to modify Markov chains [11], or to trigger manually coded reactions to specific coverage events [12]. Externally driven methods rely on expert knowledge to target the stimulus generation based on the progress of the verification effort. These methods range from manual intervention [4], to the use of sophisticated software models that are in-effect, tailored stimulus generators [13]. Optimization work has also been done that does not rely on coverage results. Anti-random verification is a method that attacks the lack of guaranteed uniqueness in the generated stimuli by incorporating tree traversal techniques in the stimulus generator [14]. The past outcomes of randomly-controlled conditions are recorded and future outcomes are chosen based on which unexplored outcomes are the most unique (i.e. have the greatest ‘distance’ from the previous executions).

3. DIRECTED RANDOM VERIFICATION WITH REDUNDANCY REDUCTION

The stimulus generator is the only stochastic component in a directed random verification environment. Because the ROR method only operates on randomly-controlled conditions, ROR’s scope is restricted to this component. The stimulus generator is essentially a machine that randomly enumerates permutations from a discrete number of actions. Together these permutations form a permutation tree. Each vertex of this tree is a randomly-controlled condition with one outgoing edge per outcome. The edges of the tree represent the generation of various stimuli for testing the design’s functions. Each testcase generated is a traversal from the tree’s root to some randomly chosen leaf. The lack of guaranteed uniqueness in the testcases means that the traversals may duplicate each other, or focus on a limited part of the tree. In both cases, the traversals result in poor verification performance. It is because the stimulus generation can be reduced to a permutation tree that tree traversal techniques have been applied to optimizing verification performance.

¹ Assuming the coverage goals truly cover all the functions of the design that need testing, implying that the verification team is not relying on serendipity to discover bugs.

The ROR technique is a tree traversal technique that utilizes checkpointing to efficiently explore the permutation tree. When the ROR method is applied to a randomly-controlled condition, that condition is no longer stochastic. The conditional statement is transformed such that it explores all of its outcomes. During the execution of the condition a checkpoint of the verification environment is created. That checkpoint is then restored once for each possible outcome. The restored verification environments are exact replicas of the original, executing from the point where the checkpoint was created. The only difference in the replicas is that each one explores a unique outcome of the checkpointed conditional statement.

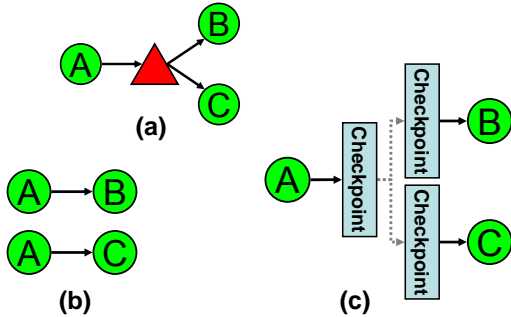


Figure 2. (a) A randomly-controlled condition selecting between two outcomes, explored both (b) without ROR and (c) with ROR

Figure 2 shows how this applies to a testcase composed of a single condition randomly selecting between two outcomes. Without the use of ROR the testcase must be executed multiple times until both outcomes are explored. With ROR the testcase checkpoints at the condition and starts two new testcases, each exploring a unique outcome from the restored checkpoint. In many ways, ROR is analogous to the recursive breadth-first-search (BFS) technique used in computer science to explore trees. BFS employs the program’s stack to record portions of its state, allowing it to backtrack to previous states as it traverses the tree. In the case of ROR, checkpointing performs that same function as the stack, enabling an efficient traversal of the permutation tree.

The effects of the ROR method can best be understood by applying it to a trivial stimulus generator. Figure 3 gives pseudocode for a stimulus generator composed of four randomly-controlled conditions $\{D1, D2, D3, \text{ and } D4\}$, ten coverage goals $\{G1, G2, G3, G4, G5, G6, G35, G36, G45, \text{ and } G46\}$ and five actions $\{R, S, T, U, V\}$. Each type of component: action, condition, or coverage goal, performs a unique role in the stimulus generator. Much like the flipping of a coin, the randomly-controlled conditions select which of two outcomes will be executed next. The actions stand for the generation of some unique stimulus, while the coverage goals symbolize that some important series of stimuli has been generated. Both standard and cross-coverage goals are represented in this example by $\{G1, G2, G3, G4, G5, G6\}$ and $\{G35, G36, G45, G46\}$ respectively. The cross-coverage goals are composites generated by the cross:

$$\{G3, G4\} \times \{G5, G6\} = \{G35, G36, G45, G46\}$$

Comparing the ROR method to traditional methods is done by creating two versions of the stimulus generator. One version uses the unaltered pseudocode; the other applies ROR to each of the

four randomly-controlled conditions. The versions are then run iteratively until all ten coverage goals are achieved.

```

stimulus_generator
begin
if( random_boolean() == true ){ //D1
R; S; T; G1;
}else{
if( random_boolean() == true ){ //D2
U; V; G2;
}else{
if( random_boolean() == true ){ //D3
G3; a := '3';
}else{
G4; a := '4';
}
if( random_boolean() == true ){ //D4
G5;
if( a == '3' ) then G35 else G45;
}else{
G6;
if( a == '3' ) then G36 else G46;
}
}
}
}
end

```

Figure 3. Example Stimulus Generator

The results of running the version without ROR are shown graphically in Figure 4. This graphical representation shows the union of the executions that the unaltered stimulus generator performed before achieving all ten coverage goals.

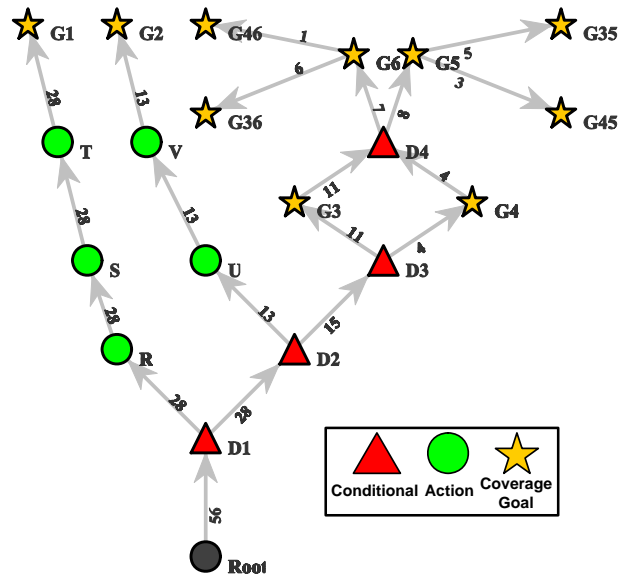


Figure 4. Execution of the Example Stimulus Generator without the use of ROR

Each node in the graph is labeled with the name of its associated action, coverage goal, or condition. Each edge is labeled with the number of times that particular combination of nodes was executed (e.g. the graph shows that the edge $R \rightarrow S$ was executed twenty-eight times). The special *Root* node is prepended to the graph to represent the common initial state of the stimulus generator executions. The graph shows that redundant execution is abundant when using this version of the stimulus generator. For

example, the edge $U \rightarrow V$ is executed thirteen times, twelve more than is necessary to verify the design under test. The only edge to avoid redundant execution is the edge $G6 \rightarrow G46$, undoubtedly the edge that led to the last coverage goal being achieved.

Executing the version using ROR results in the same graph, except that each edge is always executed the minimal number of times possible (e.g. $R \rightarrow S$ is executed only once). ROR converts the random exploration of the graph into an exhaustive exploration. This conversion results in a dramatic improvement in verification performance while in no way decreasing the amount of unique verification performed. Note that this comparison does not take into account the cost in simulation time of checkpointing and restoring the verification environment. While it has been shown that these costs are relatively small, increasing linearly as the environment grows, they are not negligible [15]. The experimentation described in section 4 takes these costs into account, providing a more accurate comparison.

Figure 4 shows another interesting result. The more randomly-controlled conditions that exist between a coverage goal and the root of the permutation tree, the less likely that coverage goal will be achieved. There is a strong bias in directed random verification towards coverage goals that are closer to the root of the tree. The difference in the number of executions of $G1$ versus $G2$ illustrates this point. Proving this bias is intuitive given that the joint probability of two conditions is always less than or equal to their individual probabilities. Stated formally, for any two statistically independent events, α and β , the following is true:

$$\forall_{0 \leq \{P(\alpha), P(\beta)\} \leq 1} \quad P(\alpha) \geq P(\alpha \cap \beta)$$

This bias is the major cause of redundant execution. The ROR method provides a means of reducing this bias by reducing the number of randomly-controlled conditions, essentially pushing the verification effort further from the root of the tree. Interestingly, reducing the number of randomly-controlled conditions does not always increase verification performance. If the ROR method is applied to only the condition $D1$, the number of redundant executions of the path $D1 \rightarrow R \rightarrow S \rightarrow T \rightarrow G1$ actually increases, reducing the overall performance.

The bias against coverage goals that are far from the root of the tree is especially adverse to cross-coverage goals. The set of conditions involved in a cross-coverage goal is equal to the union of the conditions involved in its constituent coverage goals. The outcomes needed to achieve $G35$ are equal to the union of those needed to achieve $G3$ and $G5$. Because of this, cross-coverage goals generally involve more conditions than standard coverage goals, making them further from the root of the tree. ROR's ability to reduce a coverage goal's distance from the root makes it well suited for achieving cross-coverage goals. By adeptly applying the ROR method, simulation can be better focused on achieving the cross-coverage goals of a verification environment.

4. EXPERIMENTATION

Two experiments were employed to compare verification performance with and without use of the ROR method. Unlike the simple example in section 3, in these experiments the ROR method is applied to only a small subset of the randomly-controlled conditions. Specifically, ROR is applied to those conditions involved in the achievement of cross-coverage goals. In each case it is shown how applying ROR to even a small subset

of the conditions can have a significant benefit to verification performance.

4.1 Verification of an Embedded Processor

The first experiment attempts to verify a simplified embedded processor. The verification environment for this processor, shown in Figure 5, is composed of several blocks related to verification, the processor itself, and two blocks modeling the data and instruction memories. This processor implements a Harvard architecture, providing standard arithmetic instructions, direct and indirect memory addressing, and a single interrupt line.

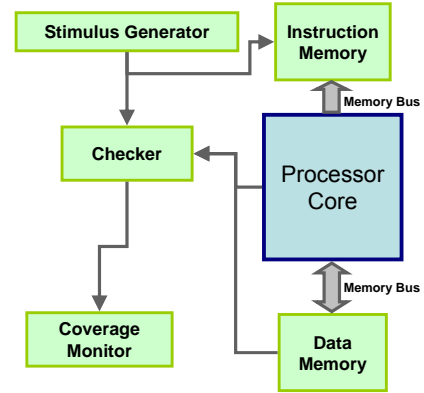


Figure 5. Processor Verification Environment

The stimulus generator for this experiment functions by generating a stream of instructions. Instruction types, their operands, and their addressing modes are all selected at random. Interrupts are also injected randomly into the generated instruction stream, triggering the processor to perform a context switch, execute a randomly selected instruction, then context switch back into the original instruction stream. Ensuring that the execution of the instruction stream renders the correct results, and that the context switch for interrupts correctly restores the values of all registers and flags, is dealt with by the checker. Standard coverage goals for this environment include executing the various instructions, accessing memory using the various addressing modes, and executing interrupts. Note that the specific values of operands are not of interest to the coverage goals, except in the case of division by zero or an attempt to access an invalid address of memory. These special events are each covered by their own coverage goal. Cross-coverage goals consist of executing every permutation of two instructions possible, both with and without an interrupt occurring between the instructions (e.g. $ADD \rightarrow Interrupt \rightarrow SUB$).

Two versions of this experiment were created that differ only in their inclusion or exclusion of the ROR method. The version including ROR applies the ROR method to only the randomly-controlled condition deciding which instruction to generate. This condition was chosen because it plays such a key role in deciding which of the cross-coverage goals are achieved by a given testcase. All other conditions are left untouched, including those deciding the operands and addressing modes of the chosen instructions. While this limited use of ROR seems of little benefit, it offers a noticeable increase in verification performance.

Figure 6 shows how much simulation time was needed by both methods to achieve the coverage goals of the verification environment. Because directed random verification is stochastic,

the results given are for the average of twenty repetitions of the experiment. Individual repetitions are run until all goals are achieved or 30 minutes of simulation time is consumed.

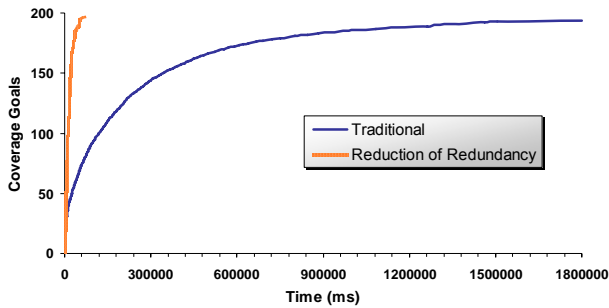


Figure 6. Results of the Processor Verification

Across the twenty repetitions the results showed strong correlation – the ROR version outperformed every time, consistently achieving all 197 coverage goals. In contrast, the traditional version was capable of reaching only 193.75 ($\sigma=1.99$) goals². ROR gave significant and repeatable improvements in the verification performance. The types of verification challenges found in this example are ideal for the ROR method, as the cross-coverage goals constitute the majority of the verification effort.

4.2 Verification of a Bus Bridge

The bus bridge experiment, shown in Figure 7, represents a different type of digital design application. The design in this experiment converts data from a trivial on-chip parallel bus to an off-chip high speed serial link.

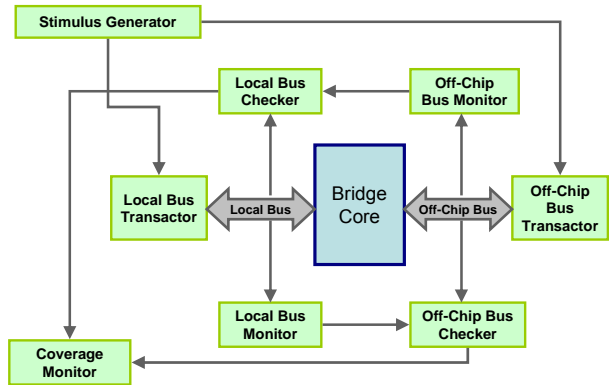


Figure 7. Bus Bridge Verification Environment

The serial link is based loosely on the protocol described in the PCI-Express specification [6]. The on-chip bus side of the bridge is simplistic in the sense that it has only the most rudimentary protocol. This protocol is used to control dedicated read and write ports and a FIFO to queue transactions. The off-chip side is far more complex – converting on-chip bus transactions into packets, calculating and checking CRCs, and managing an ACK/NAK protocol to ensure packets are transmitted successfully. In addition, the off-chip bus side of the design must transmit framing

packets at a fixed period to enable timing synchronization with the opposite side of the link.

A single stimulus generator is used to stimulate both buses, with each bus having its own pair of a monitor and checker. The monitor from one bus communicates with the checker of the other to ensure data moves through the bridge core correctly. Each bus also has a transactor that takes high level stimuli from the stimulus generator and converts it into bus specific transactions. The transactor also models the device on the other side of the off-chip bus. This modeling includes sending ACK/NAKs for incoming packets and framing packets as required. Error injection is used to irritate the off-chip bus side of the design and ensure it obeys the correct protocol for each error case. This injection can include corrupted CRC values, missing ACK/NAKs, out-of-order packet sequence numbers, and delayed or missing framing packets. The coverage goals for this experiment include ensuring data moves through the bridge without corruption, testing that FIFO overflows and underflows do not occur, and checking that the bus protocol is obeyed for valid and invalid transactions. Permutations of valid and invalid transactions form the cross-coverage goals (e.g. a valid transaction followed by a transaction with a corrupt CRC).

As with the processor experiment, two versions of this experiment were created to discover the effects of the ROR method on verification performance. The results of this comparison are summarized in Figure 8. Application of the ROR method was restricted to the randomly-controlled condition deciding error injection. This condition selects which of the four types of errors will be injected into the current transaction and is directly related to attaining the error injection cross-coverage goals, making it a good candidate for ROR.

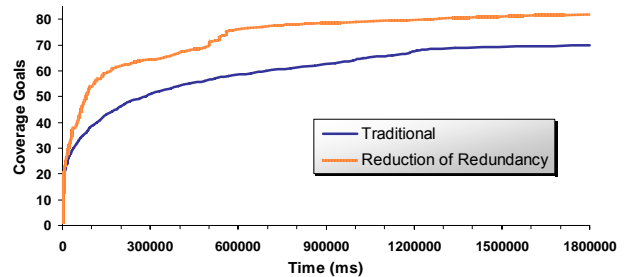


Figure 8. Results of the Bus Bridge Verification

The results, while less impressive than those for the processor experiment, highlight the increase in verification performance provided by ROR. In each of the twenty repetitions, the ROR method was able to achieve more coverage goals using less simulation time. The ROR version attained 81.6 ($\sigma=8.02$) coverage goals on average, compared to an average of 70.4 ($\sigma=3.57$) for the version without ROR. The results also hint that regardless of how long the simulations are allowed to run for, the ROR version will always achieve more coverage goals.

5. CONCLUSIONS AND FURTHER WORK

Experimentation established that the reduction of redundancy method is able to improve the performance of verification environments. These experimental results are summarized in Table 1. While there is a great deal of variation in the degree of improvement between the two experiments, statistically

² σ is the standard deviation across the total number of coverage goals achieved by the twenty repetitions.

significant improvement was made in both cases. The theoretical analysis offered in section 3 shows that this increase in verification performance is achieved by reducing both the amount of redundant simulation, and the bias against coverage goals that involve numerous randomly-controlled conditions.

Table 1. Summary of Experimental Results

	Processor		Bus Bridge	
	Coverage Goals	Verification Performance	Coverage Goals	Verification Performance
Traditional	193.75 ($\sigma=1.99$)	1.08E-04	70.4 ($\sigma=3.57$)	3.91E-05
ROR	197 ($\sigma=0$)	2.71E-03	81.6 ($\sigma=8.02$)	4.53E-05

Further work with redundancy removal will include two major thrusts. The first will attempt to discover a more formal approach to identifying which randomly-controlled conditions make good candidates for the ROR method. This could be done using either historic execution information or by looking for patterns in the relationships between the conditions and the coverage goals. Both methods would attempt to identify conditions that are especially prone to redundant execution. The second thrust will hybridize the ROR method with the CDG methods summarized in section 2. Both methods seek to increase verification performance using strategies that are not mutually exclusive. It is hoped that by combing the two strategies, information can be shared between them and even greater verification performance optimizations can be achieved.

6. ACKNOWLEDGMENTS

I would like to thank Prof. Craig Damon of the University of Vermont for his assistance with the research used in this paper. I would also like to thank Dr. Hans van der Schoot of XtremeEDA, Pascal Nsame and Mariette Awad of IBM Systems & Technology Group, Mike Mintz of Apple Valley Verification, Geoff Hulette of UC San Diego, and Zachary Weinman for their assistance with this paper.

7. REFERENCES

- [1] E. Clarke, O. Grumberg, D. Peled. *Model Checking*. Published by The MIT Press, January 2000.
- [2] M.G. Bartley, D. Galpin, T. Blackmore. A Comparison of Three Verification Techniques: Directed Testing, Pseudo-Random Testing and Property Checking. In *Proceedings of the 39th Design Automation Conference*, pages 819 – 823, June 2002.
- [3] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User Defined Coverage - A Tool Supported Methodology for Design Verification. In *Proceedings of the 35th Design Automation Conference*, pages 158 – 165, June 1998.
- [4] Synopsys, Inc. A Reference Verification Methodology for Vera. *The Synopsys Verification Avenue Technical Bulletin*, pages 16 – 22, Vol. 4 Issue 1, February 2004.
- [5] J. Craig. *Reduction of Redundancy in Directed Random Verification through Checkpointing*. M.Sc. Thesis, University of Vermont, Burlington, VT, April 2006.
- [6] PCI-Express Special Interest Group. *PCI-Express Base Specification v1.1*. Available online at <http://www.pcisig.com/specifications/pciexpress>, July 2002.
- [7] F. Haque, J. Michelson, K. Khan. *The Art of Verification with Vera*. Published by Verification Central, September 2001.
- [8] S. Fine, A. Ziv. Coverage Directed Test Generation for Functional Verification using Bayesian Networks. In *Proceedings of the 2003 Design Automation Conference*, pages 286 – 291, June 2003.
- [9] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. A Genetic Approach to Automatic Bias Generation for Biased Random Instruction Generation. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 442–448, May 2001.
- [10] S. Hao, F. Yuzhuo. Priority Directed Test Generation for Functional Verification Using Neural Networks. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 1052 – 1055, Vol. 2, January 2005.
- [11] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A functional validation technique: biased-random simulation guided by observeability-based coverage. In *Proceedings of the International Conference on Computer Design*, pages 82–88, September 2001.
- [12] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv. Cost evaluation of coverage directed test generation for the IBM mainframe. In *Proceedings of the 2001 International Test Conference*, pages 793–802, October 2001.
- [13] S. Ur and Y. Yadin. Micro-architecture coverage directed generation of test programs. In *Proceedings of the 36th Design Automation Conference*, pages 175–180, June 1999.
- [14] Y. K. Malaiya. Anti-Random Testing: Getting the Most Out of Black-Box Testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, Pages 86 – 95, October 1995.
- [15] S. Bouchenak. Making Java Applications Mobile or Persistent. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, January 2001.