CONCURRENT DESIGN ERROR SIMULATION FOR HIGH-LEVEL MICROPROCESSOR IMPLEMENTATIONS

Jorge Campos
University of California
One Shields Ave
Davis, CA 95616
jcampos@ece.ucdavis.edu

Hussain Al-Asaad
University of California
One Shields Ave
Davis, CA 95616
halasaad@ece.ucdavis.edu

Abstract — A high-level concurrent design error simulator that can handle various design error/fault models is presented. The simulator is a vital building block of a new promising method of high-level testing and design validation that aims at explicit design error/fault modeling, design error simulation, and model-directed test pattern generation. We first describe how signals are represented in our concurrent fault simulation and the method of performing operations on these signals. We then describe how to handle the challenges in executing conditional statements when the signals used by the statements are augmented by an error/fault list. We further describe the method in which the error models are embedded into the simulator such that the result of a concurrent simulation matches that of a sequence of HDL simulations with the set of errors/faults inserted manually one by one. We finally demonstrate the application of our concurrent design error simulator on a typical Motorola microprocessor. Our simulator was able to detect all detectable and modeled design errors/faults for a given test sequence and was able to reveal valuable information about the behavior of erroneous designs.

INTRODUCTION

Modern microprocessor implementations have become dependent on intricate instruction flow techniques and functional units to promote peak instruction throughput. These implementations, however critical for performance, result in processors that are more difficult to validate in their preliminary design stages, and are more difficult to test after fabrication. As a result of the increase in complexity of modern microprocessor implementations, extensive analysis needs to be done at the preliminary design stages to expose design errors and testing complications prior to design layout and fabrication. Given that preliminary microarchitecture implementations are developed under a high-level hardware description language (HDL) such as VHDL or Verilog HDL, it has become important to develop the tools that are capable of effectively validating and analyzing the testability of these high-level implementations.

Mutation-based validation techniques attempt to circumvent the complexity problem introduced by exploring any coverage measure exhaustively by using models for design errors as guidance. Error modeling is used to create an artificial collection of simple design errors that span throughout the corner cases of an implementation. As a consequence to the coupling effect between simple and complex design errors [1], a test sequence that is capable of detecting these known simple errors is implicitly capable of detecting complex design errors as well. Therefore, one application for our concurrent error model simulator is to grade a test sequence's ability to traverse the design space by concurrently and efficiently applying it to the complete set of simple design errors and reporting its coverage. The design error model used for this paper is described later in the paper.

One other promising application of mutation-based circuit simulation is that of mutation-based testing. Testing efforts require a coverage measure that is capable of affecting the maximal set of possible physical fault sites. Once this is defined, an error model can be designed to span the complete coverage measure. These error models, known as physical fault models, can be used in conjunction with our concurrent error model simulator to grade a test sequence's ability to detect possible physical faults and to give an architect valuable statistics on his implementation.

It is the goal of this paper to demonstrate that concurrent simulation of a large set of modeled design errors can be performed on a synthesizable HDL design. The rest of the paper is organized as follows. We first describe related work and then describe how signals are implemented and how operations on signals are executed to support our concurrent design error simulation. Then we introduce the problem encountered when propagating fault lists across condition statements, and we present a technique for handling conditional statements. We then briefly discuss the method in which the simulator is orchestrated, and we discuss the method in which the error models are embedded into the simulation such that concurrent error simulation generates identical results to sequential error simulation. Finally, we discuss how error models can be used for system validation and present the results obtained from our simulation experiments. The notion of the affected signal threshold and its relevance to our research is also discussed.

RELATED WORK

Analysis of controllability and observability measures through concurrent simulation methods has been previously investigated via a tag simulation calculus [2]. Under this simulation method, a single tag is propagated throughout the simulation to designate a possible change in a signal value due to an error. This method, however, results in an estimation of observability given that mutations on a signal are represented by a Δ tag that only represents a positive or negative polarity. Furthermore, this method requires the modification of the hardware description when condition statements are involved in order to compute the effects of the fault model when it causes the wrong path to be taken. The methods in [3] are an improvement as behavioral fault simulations are implemented with fault lists such that Petri Nets are used to propagate the fault lists in their event-triggered simulation environment. They fell short of creating a concurrent fault simulation environment because they use a fault free simulation phase to guide the subsequent simulation with fault-list propagation phase. Other related papers discuss methods of generating mutations of a hardware description as a means to find a test pattern that can distinguish a program from its faulty versions [4][5], but rely on a sequential simulation approach for the set of mutant implementations.

One concern raised from performing fault simulations on HDL implementations comes from the loss of resolution in the fault model due to the abstraction in the implementation. This concern is alleviated by related research that shows how the results of fault simulation on HDL implementations are comparable to those of performing fault simulation on gate-level implementations [6]. These results are therefore a basis to performing preliminary fault simulations on HDL implementations as a means to detect testing bottlenecks prior to the migration of the implementation to a lower level of abstraction where modifications become more costly.

FAULT-LIST ENABLED SIGNALS

The initial step in developing our high-level concurrent error simulator consists of determining how a signal should maintain its fault list, and how the basic signal operations should be performed on the complete fault lists. To accomplish this, a signal is first defined as an object that consists of a fault-free value along with a list of mutant values, where each mutant *m* in the signal S is a result of the corresponding parent error model. We denote the parent error model of a mutant value *m* by $\pi(m)$ such that $\forall m_i$ in the signal S: $\Pi(S) = \cup_i \pi(m_i)$. It is common that aliasing occurs between the fault-free value and one or more mutant values, in which case it is advantageous to collapse the error lists as a means to reduce the memory demand and the number of operations required by each list.

Our simulator takes as input a collection of error models E which are used to generate and insert a mutant into a specific fault site when appropriate. Let a_i be the set of fault values in signal A, such that $a_{i=0}$ denotes the fault-free value and $a_{i\neq 0}$ denotes the mutant value associated with the error model $\pi(a_i)$ that has an ID value i. Let $a \circ b$ denote an arbitrary operation on two signal values, and let $A \circ B$ denote the same arbitrary operation performed over all signal values a_i and b_i in signals A and B, respectively, such that an operation $a_i \circ b_{j,i\neq j}$ is not allowed because an operation cannot be performed across design error models. In the case where $\Pi(A) \neq \Pi(B)$, a request for an implicit (non-existent) mutant value $a_{i\neq 0}$ results in the generation of the requested value from the fault-free value. We will denote the generation process by

$a_{0\rightarrow i}$. In the rest of this paper, a value generated from the fault-free value is referred to as an implicit value and a value extracted directly from the fault list is referred to as an explicit value.

There is no distinction between an aliased mutant value and a mutant value corresponding to an error model that has not been activated, therefore we can assume that any mutant value not present in a fault list has been aliased, and it is correct to generate the corresponding mutant value from the implied fault-free value upon demand. This allows us to perform an operation across two fault lists that don't contain mutant values from the exact set of error models, and we can describe this operation by the following equation:

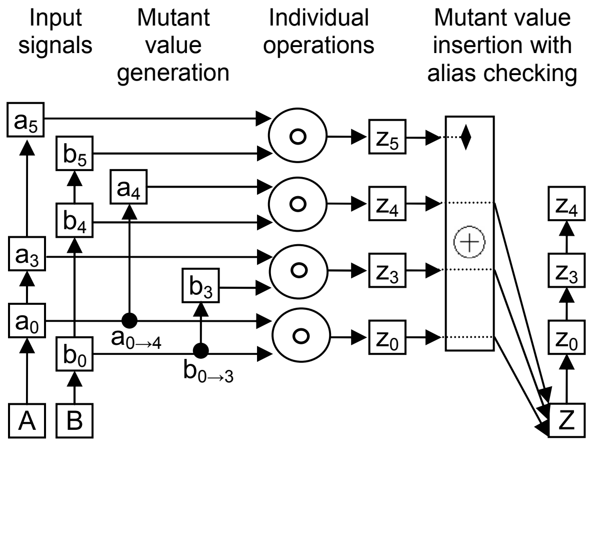
$$Z = A \circ B: \qquad \forall \pi_i \in \Pi(A) \cup \Pi(B) \;\Big|\; Z = \bigcup_i \{a_i \circ b_i\} \qquad (2.1)$$

To illustrate the above equation, let us consider the example where $A=\{a_0, a_3, a_5\}$ and $B=\{b_0, b_4, b_5\}$. The operation $Z=A\circ B$ is decomposed into the set of sub-operations $\{z_0=a_0\circ b_0,\ z_3=a_3\circ b_{0\rightarrow 3},\ z_4=a_{0\rightarrow 4}\circ b_4,\ z_5=a_5\circ b_5\}$ as depicted in Figure 1. Furthermore, if the value generated by the operation $a_5\circ b_5$ is aliased by the value generated by the operation $a_0\circ b_0$, then the resulting set of values in signal Z will be $Z=\{z_0, z_3, z_4\}$ after fault collapsing.

We next describe the basic operations on fault lists.

INSERT_MUTANT (L, m):
Inserts the mutant m into the fault-list L while preserving fault-collapsing and L's ordering of increasing mutant ID. Each fault list is implemented by a linked list of mutant values, and is referenced

Figure 1. Implementation of an arbitrary operator.



by a starting pointer and an ending pointer. This configuration allows for an insertion to the end of the fault list to be completed in $O(1)$ time. Otherwise if the mutant being inserted belongs somewhere within the fault list, the insertion is completed in $O(n)$, where n is the current size of the fault list. INSERT_MUTANT is typically called from either an operation on a signal, or from the mutant generator itself. The mutant generator only executes at most once per simulation iteration, thus if only basic operations are being performed on signals, then the many $O(1)$ time insertions by operations on signals far outweighs the few $O(n)$ insertions by the mutant generator.

ARBITRARY_OPERATION (f, L1, L2):
Performs the basic arbitrary operation *f* on all items of the fault lists L1 and L2 by the rules described earlier, such that *f* is a simple operation assumed to execute in $O(1)$ time. Given that the items in each fault list are sorted in the order of increasing error model ID, this operation is best implemented by assigning a traveling pointer to each input list and traversing these lists in the order of increasing ID. This technique allows the arbitrary operator to execute in linear time because INSERT_MUTANT can insert a mutant m into a list L in $O(1)$ time if m belongs at the end of L. Therefore, performing an operation *f* on all items of the operand lists L1 and L2 takes $O(|L1| + |L2|)$ time.

Other operations critical to high-level hardware descriptions are more difficult to implement than that of the arbitrary operator, and do not run in linear time, such as operations on signal arrays. The index of the array is a signal implemented by a fault-list, thus the value accessed by the index is the union of the values accessed from the fault-free index location with the mutant values accessed from every mutant index location. In other words, the operations on signal arrays can be represented by:

$$A = M[X]: \qquad (2.2)$$
$$\forall\, i \in \Pi(M[x_0]) - \Pi(X),\ \forall\, j \in \Pi(X) \;\Big|\; A = M[x_0]_0 \bigcup_i M[x_0]_i \bigcup_j M[x_j]_j$$

$$M[X]=A: \qquad (2.3)$$
$$\forall\, i \in \Pi(A) - \Pi(X) \;\Big|\; M[x_0] = a_0 \bigcup_i a_i$$
$$\forall\, j \in \Pi(X) \;\Big|\; M[x_j] = (M[x_j] - M[x_j]_j) \bigcup_j a_j$$

CONDITION STATEMENTS

The next important step in the development of our concurrent error simulator for high-level hardware descriptions required the conceptualization of a method to implement conditional execution on signals containing a fault list. The problem of executing a statement based on a fault list enabled condition is that the condition will be met by some of the error models and not by others. As a result, the fault list of the signals in the condition statement must be split into two partitions: the set of error models that meet the condition, and the set of error models that do not.

When executing a condition statement, the following actions need to be performed by the simulator:

i) The condition needs to be evaluated using comparison operators as described earlier, resulting in the creation of a Boolean fault list.
ii) All the signals used within the condition statement need to be initialized via partitioning such that the target partition for each fault-list item is specified in the condition fault list.
iii) The TRUE partition of each signal is used within the *then* portion of the condition statement, and the FALSE partition of each signal is used within the *else* portion of the condition statement.
iv) Upon termination of the condition statement, all initialized signals must have their fault list re-built via the recombination process. The partition that corresponds to the fault-free condition (C_0) is merged with the values extracted from the other partition that are explicitly identified by the condition fault list

Let us analyze the scenario provided in Figure 2. Notice that the presence of C_1 in the condition variable is used by the simulator to generate the mutant value B_1 from B's FALSE partition during the recombination phase in step (iv). A similar operation occurs when performing the recombination process on the signal Z such that Z_1 is generated from the fault-free value of the FALSE partition. In this situation however, Z_1 is collapsed as it is inserted into the fault list due to redundancy with the fault-free value Z_0. It is important to note that the TRUE and FALSE partitions exist as signal instantiations themselves, thus nested condition statements are handled in a nested fashion.

ORCHESTRATING THE SIMULATOR

The techniques mentioned in the previous sections are first developed and validated with small code segments and later with a high-level microprocessor implementation. We have decided to manually construct an internal representation of a high-level microprocessor implementation using the aforementioned techniques to obtain our simulation results. Our goal was not to produce a complete simulation environment, but to produce the basic tools that allow us to explore the techniques required in performing a correct concurrent simulation of a set of error models.

When taking the concepts learned from the previous two sections, it becomes obvious that the concurrent error model simulator needs to execute both paths of each condition statement in order to update both the TRUE and FALSE partitions. As a result of this observation, the *if(condition)* statements of a hardware description are used to initialize the signals using INIT_CONDITION, the *then{}* statements are used to guide every signal to operate on the TRUE partition, and the *else{}* statements are used to guide every signal to operate on the FALSE partition. Moreover, the CASE statements can be imple-

Figure 2. Concurrent design error simulation for a condition statement.

Initial Values:	$A = \{A_0=2, A_1=7\}$ $B = \{B_0=4, B_2=5\}$ $Z = \{Z_0=6, Z_3=3\}$
VHDL Condition Statement	**Corresponding Actions Required for Concurrent Simulation**
IF (A<B)	STEP i: $C = A<B = \{C_0=T, C_1=F, C_2=T\} = \{C_0=T, C_1=F\}$
	STEP ii: $A.T <= \{A\_T_0=2\}, A.F <= \{A\_F_0=2, A\_F_1=7\}$
	$B.T <= \{B\_T_0=4, B\_T_2=5\}, B.F <= \{B\_F_0=4\}$
	$Z.T <= \{Z\_T_0=6, Z\_T_3=3\}, Z.F <= \{Z\_F_0=6\}$
THEN	STEP iii: Use the TRUE partition of signals A, B, Z
B <= Z;	B.T <= Z.T => $B.T = \{B\_T_0=6, B\_T_3=3\}$
ELSE	STEP iii: Use the FALSE partition of signals A, B, Z
A <= Z;	A.F <= Z.F => $A.F = \{A\_F_0=6\}$
END IF;	STEP iv: $A <= A.T \cup \{A\_F_{0\rightarrow 1}=6\} = \{A_0=2, A_1=6\}$
	$B <= B.T \cup \{B\_F_{0\rightarrow 1}=4\} = \{B_0=6, B_1=4, B_3=3\}$
	$Z <= Z.T \cup \{Z\_F_{0\rightarrow 1}=6\} = \{Z_0=6, Z_{0\rightarrow 1}=6, Z_3=3\} = \{Z_0=6, Z_3=3\}$

mented by a series of *if-then-elseif* statements.

Preliminary error-model simulations are performed using a VHDL description of the Motorola 6800 microprocessor. The microprocessor description is imported into C++ with the following modifications:

i) VHDL statements are imported to C++ using overloaded operators within the *signal* class.
ii) Access of sub-vectors in the VHDL syntax is imported using the *bitvector_signa*l class.
iii) VHDL condition statement handlers are imported to execute TRUE and FALSE partitions.
iv) Placing each process in the hardware description into a module construct object that handles the signal initialization, process execution, and signal propagation tasks.
v) A netlist is a set of module constructs.

Given that the simulation granularity of this preliminary simulator is the same as the VHDL process level, a significant number of statements in a process are being fired unnecessarily because the sensitivity list in a process is used to fire *every* statement in that process. This limitation is of no concern at this time because the goal of this research is to develop and justify concurrent error simulation techniques. The correctness of the concurrent simulation techniques presented in this paper have been validated by comparing the results of simulating numerous error models under our concurrent error simulator with the corresponding set of sequential error simulations obtained by the Synopsys CAD tools.

MUTANT VALUE GENERATION

It should be clear at this point that the core concurrent error-model simulator does not produce mutant values; its purpose is simply to propagate them. The mutant values are generated by separate engine(s) we call *mutant value generator(s)*. This results in a simulation environment that is adaptable to any design-based/fault-based error models by creating the appropriate error model generator(s) that are in charge of inserting the appropriate mutant values into the appropriate signal(s) under the appropriate condition(s).

In order to conjecture on the methods of generating mutant values, let us take a feedback circuit into account. When an error model is first activated in the circuit, it generates a mutant value that might feed back to the same activation site to re-activate the error model. At this point, it generates a mutant value from an already mutant signal. As a result, a mutation generator is activated by signals where its corresponding mutant value is given higher preference over the fault-free value. That is to say that the mutation generator uses a signal's fault-free value if and only if a mutant value of corresponding ID tag does not exist. Furthermore, any mutant values that are inserted into a signal will replace the previous corresponding mutant value if it exists.

USING DESIGN ERROR MODELS FOR SYSTEM VALIDATION

Based on previous work in [7], a mutation control error (MCE) has been defined as the quintuplet (*i*, *c*, *s*, *vc*, *ve*) such that *i* is the current instruction, *c* is the cycle in the processor pipeline, *s* is the control signal that will experience the mutant signal, *vc* is the correct value of the control signal, and *ve* is the erroneous value that will be inserted into the fault list of the control signal *s*. The above definition can be applied directly to a structural description of a microprocessor where the instruction is deciphered by the hardware description and the processor cycle is obtainable from the implementation. Unfortunately, not all hardware descriptions have implementations with explicit instruction and processor cycles, such as with microprocessor implementations based on a finite state machine (FSM). Under this situation, the structural MCE design error model needs to be adapted into the quadruplet (*s*, *c*, *vc*, *ve*) such that *s* corresponds to the explicit processor state, *c* is the control signal that will experience the mutant signal, *vc* is the correct value of the control signal, and *ve* is the erroneous value that will be inserted into the fault list of the control signal *s*. This modification is possible because the combination of the instruction *i* and the processor cycle *c* of a structural microprocessor represent the processor state.

We have used the modified MCE model to implement an automatic design error generator for the FSM-based implementation of the Motorola 6800 microprocessor by John E. Kent [opencores.org]. The exhaustive set of MCEs for this implementation consists of 300,092 errors, and the four distinct simulation runs described in Figure 3 were performed for observation purposes.

The first two simulation runs were performed by only labeling the primary outputs (POs) as the observation points, and a second pair of simulations were later performed by adding the ac-

Figure 3. Description of simulation methods used.

	Number of MCE Errors	Test Sequence	Observation Points
Data Set 1,2	Exhaustive 300,092	2000 random vectors	Primary outputs
Data Set 3,4	Exhaustive 300,092	2000 random vectors	Primary outputs Registers (acca, accb, SP, PC)

cumulator registers A and B (acca, accb), the stack pointer register (SP) and the program counter register (PC) into the set of observation points to determine the effectiveness of increasing this implementation's observability.

Figure 4 graphs the total number of design errors detected across each simulation run. It is interesting to notice that increasing the observability did not result in a significantly greater number of design errors being detected. Furthermore, data set 1 demonstrates how the first simulation begins an unproductive simulation path at around test vector number 1100 but reaches a highly productive state sequence at test vector 1928 that allows it to almost reach the performance results of the second simulation. This sudden change in productivity along with the sudden peak in data set 1 of Figure 6a demonstrate the possibility of achieving a higher detection rate if a test sequence is generated that maintain a high error model activation count.

Figure 5 graphs the number of design errors dropped from the simulation after a percentage of internal signals are affected; this percentage is being denoted as T_{AS} (affected signal threshold). From the graph, we can see that the most common threshold occurs at 10%, letting us know that most of the error models were dropped after they affected 10% of the internal signals. The driving concept behind the affected signal threshold relies on the fact that as the number of internal signals experiencing the effect of specific design error increases, the probability that a primary output is also affected will also increase. Therefore, it is expected that design errors will reach a high probability of being dropped from the simulation after affecting a threshold of internal signals (T_{AS}). Thus naturally, if the hardware description is optimized by observability measures in such a way that the T_{AS} level is substantially low, then it is expected that an improved number of error models will be detected per test sequence. Furthermore, fault-dropping plays a larger role on a design with a low T_{AS} level, as it confines the number of signals that the average error model affects before that error model is dropped.

Figure 4. Detection of design errors across each test sequence for simulation runs 1 through 4.

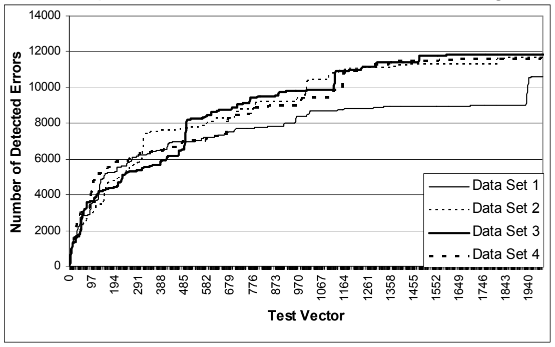


Figure 6 graphs the number of design errors activated by each vector for simulation runs 1 through 4. Figure 6a corresponds to two simulations labeled data sets 1 and 2. It is clear that the second simulation run affected a lower average number of design errors, but had more distributed peaks that helped it detect a larger number of design errors. Figure 6b corresponds to the two simulations labeled as data sets 3 and 4. The development of the third data set in the form of a step function might lead one to assume that the extra observation points have resulted in the sudden drops in active error counts as error models are removed from the simulation. This anomaly, however, does not correspond to the extra observation points and must be disregarded. To prove this, we must compare data between Figures 4 and 6b to notice that each sudden drop in data set 3 of Figure 6b does not have a corresponding steep incline in data set 3 of Figure 4. Instead, we must observe that each sudden incline of a curve in Figure 4 corresponds to a sudden peak in the corresponding graph of Figure 6.

Figure 5. Number of occurrences per T_{AS} value.

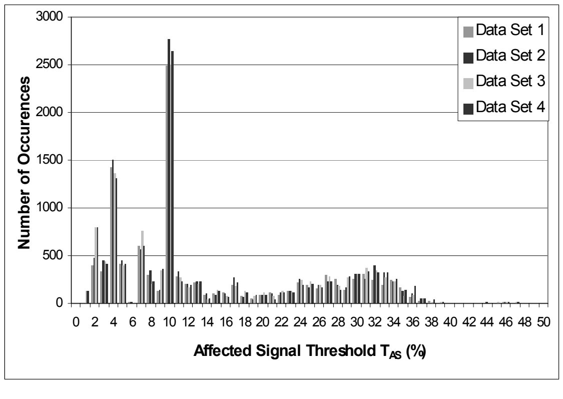
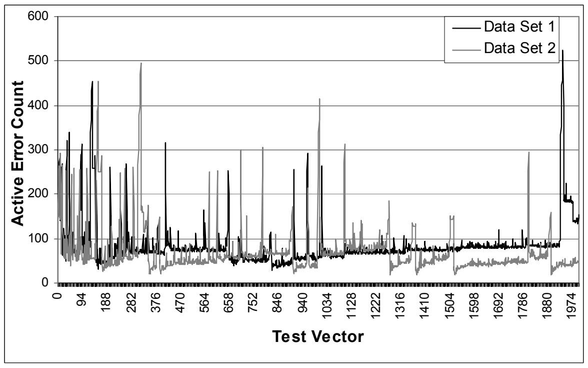
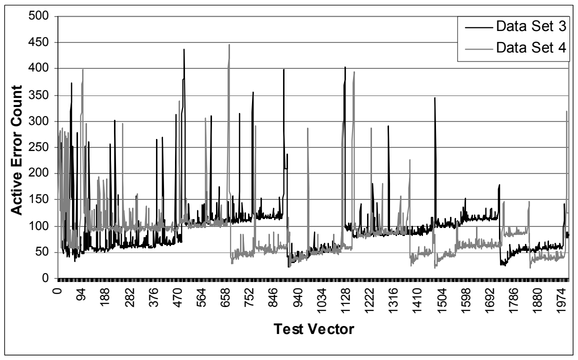


Figure 6. Number of active error models across each test sequence for simulation runs 1 to 4.



(a)



(b)

The previous graphs bring to our attention the difficulty in generating tests to detect design errors given that an approximate average of 100 out of a collection of 300,092 (less than 0.1%) design errors are active at any point in the simulation, and a peak of 525 design errors are active (0.175%). On a positive note, the low activation rate of design errors serves to encourage the implementation of a concurrent design error simulator for the validation technique introduced earlier because an exhaustive set of error models can be simulated with an acceptable performance cost.

CONCLUSIONS

In this paper we have introduced a method of simulating mutation-based modeled design errors on high-level microprocessor implementations. Furthermore, we have discussed the challenges of concurrent error model simulation in the presence of condition statements and we have presented an effective way of handling them. Finally, this paper has demonstrated the practicality of our simulation technique and demonstrated that an modeled error has a high probability of being dropped after affecting a threshold of the internal signals. Furthermore, we have provided a versatile simulation system capable of concurrently simulating distinct error model types ranging from design error models geared towards system validation to fault models geared towards controllability and observability analysis or post-silicon system testing.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

REFERENCES

[1] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, “Hints on test data selection: Help for the practicing programmer”, *IEEE Computer*, vol. 11, pp. 34-41, April 1978.

[2] F. Fallah, S. Devadas, and K. Keutzer, “OCCOM: Efficient computation of observability-based code coverage metrics for functional verification”, *Proc. Design Automation Conference*, 1998, pp. 152-157.

[3] F. Dominique, B. Paul, and S. Jean-Francois, “Behavioral fault simulation: Implementation and experiments results”, Proc. *IEEE International Workshop on Electronic Design, Test and Applications*, 2002, pp. 81-85.

[4] I. Ghosh and M. Fujita, “Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams”, *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 402-415, March 2001.

[5] J. Shen and J.A. Abraham, “An RTL abstraction technique for processor microarchitecture validation and test generation”, *Journal of Electronic Testing: Theory and Applications*, vol. 16, pp. 67-81, February-April 2000.

[6] L.-C. Wang and M.S. Abadir, “On efficiently producing quality tests for custom circuits in PowerPCTM microprocessors”, *Journal of Electronic Testing: Theory and Applications*, vol. 16, pp. 121-130, February-April, 2000.

[7] H. Al-Asaad, *Lifetime Validation of Digital Systems via Fault Modeling and Test Generation*, Ph.D. Dissertation, University of Michigan, Ann Arbor, September 1998.

[8] F. Corno et. al., “Automatic test program generation from RT-level microprocessor descriptions”, *Proc. International Symposium on Quality Electronic Design*, 2002, pp. 120-125.

[9] G. Al-Hayek and C. Robach, “From design validation to hardware testing: A unified approach”, *Journal of Electronic Testing: Theory and Applications*, vol. 14, pp. 133-140, April 1999.