A Methodology for Stochastic Fault Simulation in VLSI Processor Architectures

Christian J. Hescott, Drew C. Ness, David J. Lilja
Department of Electrical and
Computer Engineering
University of Minnesota
Minneapolis, Minnesota 55454
{hescott,dness,lilja}@ece.umn.edu

Abstract

We present a simulation methodology for fault analysis within VLSI designs. Our approach uses stochastic fault injection to decrease the required simulation time when performing fault analysis. By using a fraction of the total number of available injection points, we obtain a statistical characterization of the design under test without rigorously testing every gate within the circuit at every point in time. Our approach targets the characterization of fault behavior in large-scale circuits including full CPU architecture designs that would normally be too complex for traditional fault-analysis techniques. We present two methods for performing stochastic fault injection. One requires component library modification and is implemented entirely in VLSI source code. The second does not modify component libraries but rather interacts directly with the simulation environment and is implemented using programming language interface (PLI). We compare the simulation cost of each as well as the trade offs in design analysis. Furthermore, we introduce an optimization strategy when performing fault analysis that utilizes the checkpointing feature typically found in VLSI simulation environments. This optimization eliminates the need for multiple simulations of a single benchmark program with different random seeds. Our techniques can be implemented in any VLSI simulation environment supporting PLI and is compatible with both VHDL and Verilog designs. We present our algorithms and demonstrate their usage on the fully implemented OpenRISC processor [2]. Finally, we validate our method by demonstrating an increase in accuracy when increasing the number of injected faults. We show that even with as few as 250 faults, we see at most an 8 percent difference in the standard deviation and as the number of faults increase, the accuracy quickly approaches 100 percent.

1 Introduction

With the predicted error rate trends in CMOS technology [26] [22], fault tolerance will be pushed further to the forefront of problems in architecture research. It is necessary to have the tools in place for the research community to become more involved without needing to climb the steep learning curves presented by device physics, circuit design analysis and verification. Having a simulation methodology that allows for architecture designers and researchers to begin to look at the problem from their level of abstraction can help to decrease the barrier of entry.

Most research has focused on verification techniques which prove correct functionality or measure the fault coverage of a circuit. However, this can be too stringent when evaluating new fault tolerance techniques especially in the case of studying fault characterization in new architectures and technologies. Typically the circuits are too large to simulate in any reasonable amount of time using the traditional verification techniques. This forces a trade off between level-of-detail and circuit size.

Often, sub circuits must be used for detailed simulations while entire system-level simulations cannot incorporate all of the details such as logical gates and their connecting wires. This can be seen in recently published papers [25],[15],[21] that studied the sensitivity of various super scaler CPU architectures to try to identify which components are the most critical within the design. Due to the large scale of the designs however, simplifications had to be made in order to inject faults and measure their behavior within the system. For example in [25], faults were only injected into latches and not any combinational logic gates. A more inclusive model would use a logical circuit level simulation in which faults can be injected into any gate. Consequently the effects of logical errors as they propagate through the system can be observed in a more realistic manner.

The use of traditional verification techniques becomes

more intractable when complex circuits are studied to inquire the role that each level of design abstraction plays in fault sensitivity. An example application might be determining the most productive abstraction level to target for fault tolerance: circuit, architecture, operating system or software. Performing a study that would incorporate all these levels of abstraction while trying to study the complex interactions of fault propagations within a circuit and how they manifest in each of the abstraction levels would prove to be a truly intractable problem [16].

Statistical fault injection is a simulation methodology that can lessen the total simulation time of a design while still obtaining a characterization to how the entire circuit behaves to faults with a very high level of detail. This is achieved by using a subset of the total fault population. It is assumed that this subset is a good representation of faults encountered under typical operating conditions and consequently the measured circuit's response reflects this.

The remainder of the paper is organized as follows: Section 2 provides motivation for our two statistical fault injection algorithms and a third algorithm for increasing granularity of fault injection. Section 3 provides the details of the algorithm and is focused on implementing them in a simulation environment. In section 4 we provide an example of statistical fault injection on a target design and discuss the trade offs between our two statistical fault injection algorithms. Section 5 presents related work and section 6 concludes the paper.

2 Motivation

2.1 Statistical Fault Injection

We propose a simulation methodology that relaxes the rigid constraints in circuit verification by using a subset of the total possible faults injected into a circuit. By randomly selecting faults from the entire available fault population, we can obtain a statistical characterization of a circuit's behavior to faults and fault propagation. This can lead to shorter simulation times and more rapid testing of newly proposed fault tolerant techniques. Our initial algorithm treats all fault injection points as equally likely, however, this is not required and non-uniform distributions are possible. Our goal is not to obtain a fault coverage metric as in [17], [8], [12], [7], [3], [13] but rather to provide a simulation environment in which a better understanding of fault interaction can be obtained while at the same time providing a method for rapid testing of fault tolerant techniques.

We introduce two algorithms for injecting faults statistically. The first method (referred to as MODLIB) uses a modified component library from a target technology used for synthesis of a design. The MODLIB algorithm injects faults into the outputs of all gates within the design. The decision to inject faults occurs within each synthesized gate and consequently is distributed across the entire design. Our second algorithm (PLInject) uses PLI to create a central fault injector routine that randomly chooses gates to inject faults. This centralized decision routine results in a tremendous performance benefit in terms of simulation time over the MODLIB algorithm. However, the MODLIB algorithm can still have benefits over PLInject when used on smaller circuit sizes as will be discussed in section 4.

2.2 Finer Granularity Using Checkpointing

Characterizing an overall system requires several workloads to be tested while faults are injected. Some type of evaluation criteria is required to determine how a fault effects the system. A typical criteria is to determine weather or not the fault shows up as a software-visible error manifesting itself in one of the architected registers or in memory. Other criteria might include special purpose registers such as the program counter. The extreme case would be costly including every register value within the architecture including memory and caches. A much simpler criteria would be checking that the workload completed successfully. Optimized criteria aim to reduce the amount of storage required by hashing across the memory states and registers of the architecture as in[23].

When performing fault analysis, typically a golden run must be obtained. This golden run constitutes a complete simulation of the workload without any faults injected. Upon completion of the workload, the testing criteria results are recorded as a base case. The simulation is then repeated with fault injection enabled. Upon this second completion of the workload, the results are checked with that of the golden run to determine if the injected fault or faults caused visible errors. Because statistical fault injection is used, several runs of the same workload must be performed with different fault injection points to obtain confidence levels in the results. This can be very costly in simulation time as obtaining good confidence intervals can sometimes take thousands of individual simulation runs for each workload.

The problem becomes even worse when the effects of only a single fault injection are being studied. In this case it can be overkill to simulate an entire workload to see the effects of a single fault injection. Often times, the effects of the fault can manifest as a visible error within a much shorter time period. Consequently the entire workload does not need to be simulated to completion in order to see the effects it has on the system. By shortening the golden run, a single injected fault can be observed on a much shorter time scale. The overall effect is a much more productive simulation.

This is especially true when studying the time sensitivity of an injected fault. An example would be comparing

fault sensitivity during the initialization phase of a workload as opposed to the core processing phase. Measuring sensitivity based on when a fault is injected could conceivably require injecting the fault at every time step during the simulation run. Doing this one fault at a time would require at least $\frac{T}{timescale}$ complete simulations where T is the total simulated time of the benchmark in seconds and $timescale$ is the time scale used in the simulation environment (e.g. 1 nanosecond). It should be noted this does not include the additional runs to obtain confidence intervals. Alternatively, by reducing the time length of the golden run, the total simulation time can be reduced quite significantly while still effectively achieving the same results.

This is exactly the motivation behind our checkpoint algorithm presented in section 3.3. We introduce this algorithm to provide a finer granularity of fault analysis within a DUT that effectively reduces the cost of simulation.

3 Implementation

This section presents our fault injection techniques. The algorithms were intentionally designed to be model-independent so different timing models and fault injection types can be interchanged. Examples include transient pulses, bit flips, stuck-ats and any other injection type that can be modeled as a logical value on a wire.

Section 3.1 presents an algorithm which requires modification to the behavioral specification of a target technology's library. In section 3.2 we show an algorithm that does not require library modification but rather uses PLI to inject faults directly onto wires within the simulated environment.

It is important to note that the two algorithms described here can be used for general fault characterization in a DUT for an entire benchmark run. They are independent of the checkpointing feature described in 3.3. The checkpointing algorithm can be used for higher detailed fault characterization.

3.1 Modified Library Error Injection (MODLIB)

Synthesized designs rely on a behavioral level description of the individual gate types for the target technology when used for simulation. For example a two-input AND gate would look something like that shown in Figure 1. Here the two inputs (A and B) are logically anded together and the output placed onto Z. This along with the path delays constitute the behavior of the gate during simulation.

Our algorithm simply breaks the connection of Z and instead connects it to our own module that is responsible for fault injection (figure 2). The fault injection module can be input sensitive or time sensitive. Time-sensitive fault injection uses a delay parameter that controls how often an fault can be injected. For example a five nanosecond delay would

```
module AND2 (Z, A, B)
output Z; input A,B;
  and (Z, A, B);
  specify // delay parameters
  ...
endmodule
```

Figure 1. Behavioral description of AND2

```
module AND2 (Z, A, B)
output Z; input A,B;
  and (tmp_Z, A, B);
  // Fault inject module
  FAULT_INJECT(Z, tmp_Z);
  specify // delay parameters
  ...
endmodule
```

Figure 2. Fault injected behavioral description of AND2

mean that fault injection would be possible (however not guaranteed) every 5 nanoseconds. Input sensitive fault injection injects faults whenever the inputs of the gate change. The decision to inject a fault is based on the desired model that must be provided to the algorithm. A basic example would use a uniformly distributed pseudo random number generator to sample random values and compare them to the desired fault injection rate. If the random samples fall below the desired rate than an fault is injected.

This method of fault injection is entirely implemented in the VLSI language environment. Consequently no special compilation or simulator modification is required. It does however require a synthesized design with the behavioral source code of the target library.

3.2 PLI Fault Injection (PLInject)

This algorithm targets synthesized gates within a design however it could be applied to behavioral models with slight modification and some limitations. The algorithm assumes the lowest level modules in a design hierarchy to be gate descriptions. This is typically the case in synthesized designs where netlists are created using calls to gates supplied by the target technology's library. Our algorithm injects faults directly onto the wires that connect the gates together.

In order to select which gates are targeted for fault injection, the algorithm performs a depth-first search of a user-supplied module to find all child leaf modules. For each leaf

module, the input and output ports are identified (selectable by the user). The leaf module, ports and external wires connecting to the ports (see Figure 3) are all placed into a hash table for future referencing. Each port of the leaf module is placed in an array and used for selecting a random wire in the fault injection process. Figure 4 shows a design hierarchy represented as a tree. The CPU module is synthesized and consequently only contains netlists of the target library. Its child nodes are leaf nodes and consequently specifying the CPU module targets all its children for fault injection.

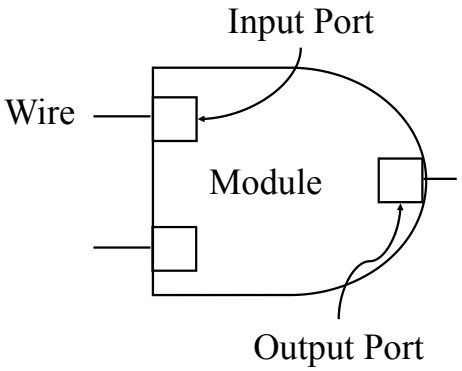

Figure 3. Module, port and wire representations in an AND gate

Targeted fault injection can be accomplished by specifying a subset of the total modules in the hierarchy. Moving higher up the hierarchy selects a larger number of gates for fault injection. Conversely moving down the hierarchy selects a smaller number of gates to the point where specifying a leaf module injects faults into a single gate. Multiple calls can be made to the module selection routine to add additional gates. Any duplicate modules that are found as a result are ignored so each module instance can only be added once.

Gate selection for fault injection is assumed to be uniformly distributed across all ports. However, this assumption can be relaxed by applying distribution weights to the ports in the port array and modifying the port selection routine.

System calls are provided to the VLSI environment allowing easy invoking of a fault injection. A call to the fault injection routine selects a port at random and writes a value onto its connecting wire. Different types of fault injection models can be specified by modifying the fault injection routine.

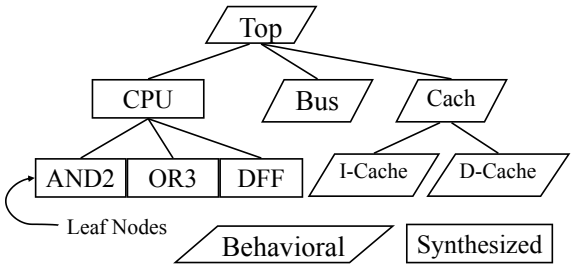

Figure 4. Example design hierarchy with synthesized and behavioral modules

3.3 Checkpoint Algorithm for Fault Injection

In this section we present the algorithm for checkpointing fault injections. The algorithm consists of creating saved checkpoints of the entire simulation environment as a basis to fall back on once the fault injections and testing are complete. Any catastrophic failures or any dormant faults within the system are eliminated when the previous saved state is loaded. With each checkpoint a golden run is performed in which fault injection is disabled. At the end of the golden run, the architected states are saved. These values are used as the "correct" values to compare against once faults are injected into the system.

In figure 5 we show the steps of the checkpointing algorithm. We have broken down the algorithm into three phases: *Checkpoint and Golden Run Creation*, *Injection and Checking*, and *Move to Next Check Phase*. The first phase is responsible for creating an initial checkpoint and performing the golden run simulation. The second phase injects a fault and allows it to propagate before stopping the simulation and checking the results against the golden run. The third phase is responsible for removing the effects of the fault injection and continuing the simulation until the next fault injection point.

The algorithm can be adjusted to control the granularity ($T_{granularity}$) of fault injection and checking. For example, if it is expected that a benchmark is highly sensitive to the time at which a fault is injected , a finer granularity (smaller $T_{granularity}$) can be used and consequently more time steps within the benchmark will be injected with faults and the results checked. A second timing parameter ($T_{latency}$) controls the length of time that faults are allowed to remain in the system before a check is applied. Latent faults may or may not have an impact on the architected state of the program. By adjusting this parameter, checks can be applied to a circuit either shortly after a fault injection or after a relatively long period of time has passed. For example to see immediately visible effects faults have on

the system, a small $T_{latency}$ would be used. Conversely using a larger $T_{latency}$ would measure the long term effects that faults have when they are allowed to propagate through the system and remain dormant for large periods of time.

Checkpoint and Golden Run Creation

1. Create a checkpoint saving entire state of the simulation environment
2. Disable fault injection
3. Run for $T_{latency}$ time steps
4. Create a saved state of all testing criteria (this is the golden run)

Injection and Checking

5. Reload checkpoint created in step 1
6. Enable fault injection
7. Run for $T_{latency}$ time steps
8. Compare architected states with those created in step 4 (record all visible errors)

Move to Next Checkpoint

9. Reload checkpoint created in step 1
10. Disable fault injection
11. Run for $T_{granularity}$ time steps
12. Go to step 1

Figure 5. Steps of the Checkpointing Algorithm

Several fault injection-check pairs can be made for each saved checkpoint by repeating steps 5 through 8 (denoted *Injection and Checking*) and allowing the fault injector to select a new injection point each time. Furthermore, fault injection complexity can be increased by allowing multiple faults to be injected for each pass through the *Injection and Checking* steps. Multiple fault injections can reveal sensitivities in the circuits that are not captured when performing single fault injections [18], [27].

4 Experimental Setup and Results

We demonstrate our algorithms on the OpenRISC processor [2]. This is an embedded scalar processor model that has been successfully implemented on FPGA and ASIC technologies. The design contains a five-stage integer pipeline, instruction and data caches, and virtual memory support. A ported version of the Gnu C compiler is available for compilation of C programs. The OpenRISC 1200 is implemented in behavioral Verilog.

We synthesized the design with Synopsys Design Compiler using Virtual Silicon's UMC 0.18 μm process library available from IMEC [1]. The worst case parameters were used in the compilation. For our MODLIB results, the Verilog behavioral code of the component library was modified as discussed in section 3.1. The random fault injector was set to inject on a 1 ns timescale. For a fair comparison, the PLInject results use the entire synthesized design for random injection with a 1 ns timescale. For the simulation performance comparison (section 4.1), our results are obtained without actually injecting faults as this would not give a fair comparison of simulation cost. Instead, all of the necessary steps for deciding weather or not to inject a fault are measured. The actual injection of a fault is a negligible overhead. For the results presented in section 4.2, however, fault injection is enabled.

The designs were simulated using Cadence's Logic Design and Verification package version 4.1. A 733 MHz Pentium III processor with 512 MB of RDRAM running Linux Redhat Version 8.1 was used for running the simulator.

4.1 Comparison of Two Statistical Fault Injection Algorithms

In this section we compare the advantages of each algorithm presented in section 3. We evaluate the performance cost of each and discuss other factors that contribute to the usability of each algorithm.

The simulation rates (in instructions per second) for the two designs are shown in figure 6 along with three reference cases. The Simplescalar bar is the typical 150 thousand instructions per second mentioned in [6]. The bar denoted RTL is the unsynthesized version of OpenRISC 1200 design and the bar denoted Synth is the synthesized version without fault injection. The Synth result represents the best possible rate we could obtain for our fault injection algorithms. As figure 6 shows, an RTL design simulator is several orders of magnitude below a typical performance simulator (Simplescalar). Moving to a synthesized design results in about a half order of magnitude slowdown compared to the RTL design.

For our PLInject we see another half order of magnitude slowdown compared with the synthesized design. This isn't bad considering that we are adding an additional event every 1 ns of simulated time to check for fault injection. The MODLIB design is almost three orders of magnitude slower than the synthesized case. This is caused from an additional G events occurring every nanosecond where G is

the number of synthesized gates. By distributing the decision mechanism to every gate we incur a tremendous performance cost.

It should be apparent that the PLInject is the only choice when simulating large designs. With a two-order magnitude decrease in simulation performance, the MODLIB method of injection can push simulation times into the range of months or years. However, the MODLIB design can still be usable if the evaluation timescale isn't as small as that used in these results. Furthermore, the MODLIB design can be modified to be input sensitive instead of time sensitive. In this case, a fault is injected with some probability only when the inputs of a gate change. Using this fault injection method is much less costly in terms of simulation time as the decision making only adds a modest amount of overhead and does not create any additional events in the simulator.

The MODLIB design has an advantage over the PLInject method when developing stochastic models for multiple concurrent fault injections. It is more intuitive to map a model onto the MODLIB algorithm as the fault injection decision occurs in each gate. Consequently specifying a model that has a 10% chance of a gate being injected is straightforward and works as expected with multiple fault injections. The algorithm naturally handles multiple faults injected at the same time step without any further specification within the fault injection model.

This is not the case with the PLInject algorithm. Because it uses a single routine for fault injection, the model must specify when more than one fault should be injected concurrently. Developing a realistic model to handle multiple concurrent faults is not entirely straightforward but can be done.

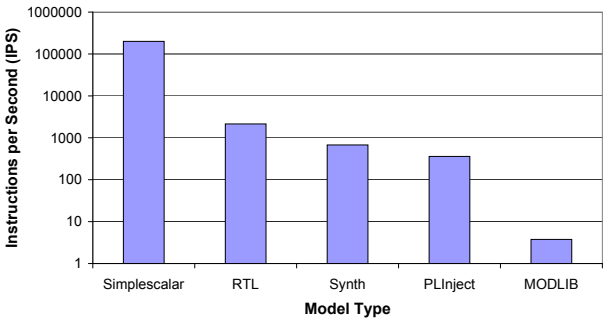

Figure 6. Simulation performance comparison

These results and discussion suggest that neither algorithm is completely superior to the other. Rather the choice is dependent on the experimental setup and the type of information that is desired.

Name	Description	Dyn. Inst.
cbasic	Basic C constructs	15000
dhry	Dhrystone benchmark	50000
except	Exception testing	32000
mmu	Memory management unit test	26000
mul	Multiplication unit test	17000
syscall	System call test	23000
tick	Test tick timer with interrupts	19000
uart	Serial port test	75000

Table 1. OpenRISC Synthetic Benchmarks

4.2 Fault Sensitivity

To demonstrate an example of statistical fault analysis, we conducted an experiment to measure the sensitivity of the OpenRISC processor to injected fault rates. For this experiment we used the PLInject fault injector without checkpointing. A set of 8 benchmark software programs provided with the OpenRISC distribution were used as sample workloads. The different workloads are listed in table 1. Most of the workloads are simple synthetic software benchmarks used to test various functionality of the design. For example the *uart* program tests the serial interface of the OpenRISC processor. A version of the dhrystone benchmark is also included. It should be noted that the OpenRISC simulation environment does not contain an operating system or file I/O. Consequently the benchmarks must use modified versions of printf routines and only static memory assignments. The Linux operating system has been ported to the openRISC environment and provides all suitable routines for standard program simulation. However, a full detailed analysis of fault behavior and its effects on the operating system and industry standard benchmark programs is beyond the scope of this paper. We leave this to future work and instead concentrate on demonstrating the algorithms presented here.

Figure 7 shows the measured response of the openRISC processor to fault injection rates ranging from 10^{-9} to 10^{-1} (horizontal axis). Here the injection rate corresponds to the probability of a fault being injected at every 1 ns of simulated time. For example a benchmark running for 2 ms of simulated time with a 10^{-5} fault injection rate would expect to have roughly 20 total faults injected. Each benchmark was run a total of 10 trials with a different random seed (Note: Random seeds did not vary across benchmarks). Due to the nature of statistical fault injection, both the number of faults injected and their location vary with each different random seed. The measured response is shown on the vertical axis corresponding to the fraction of the eight benchmarks that failed to finish correctly either due to invalid results or runaway program behavior caused by invalid pro-

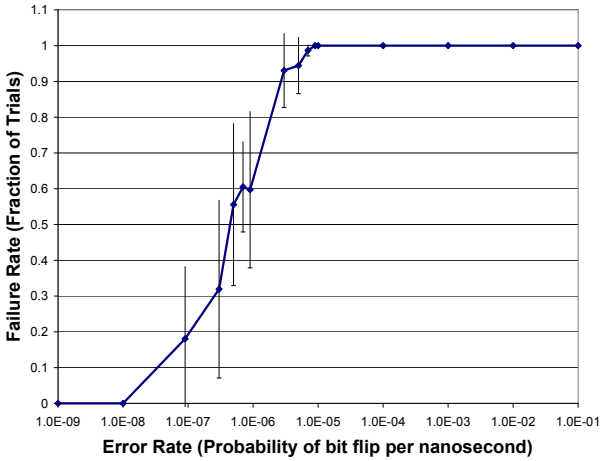

Figure 7. OpenRISC CPU fault sensitivity

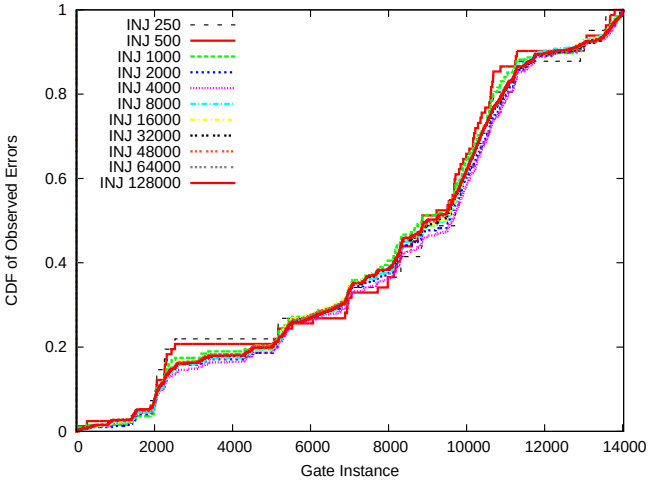

Figure 8. Cummulative error response for each gate running the Dhrystone benchmark

gram counter references. The results are averaged over all trials and the 90% confidence interval is provided in the figure.

These results themselves are not rigorous enough to draw any conclusions about openRISC fault response. However this example demonstrates how this technique can be used to fairly quickly obtain a characterization of a system across different workloads. The total simulation time for each trial of all 8 workloads is on the order of hours.

This type of experiment can help to reveal workloads that are showing higher susceptibility to faults that could possibly be programmatically reduced. For example, a benchmark that fails the experiment every trial at a lower injection rate than the others could suggest that there is something in the design of the program that is extremely sensitive to system errors. Performing a closer analysis of how the faults manifest as errors could pinpoint the weakness of the program and suggest a method to tolerate it.

The response curve in figure 7 could also be used for comparisons of circuit level fault-tolerance techniques. A good fault-tolerance design would shift the curve to the right while poor fault-tolerance designs would be shifted to the left or remain unchanged.

4.3 Validation of Statistical Fault Injection

Statistical fault injection has a distinct advantage requiring fewer fault injections compared to deterministic approaches. However the accuracy of the results is less than what would be obtain with a purely deterministic approach. In this section we derive a measurement of accuracy for our fault injection analysis and show the results for our eight chosen benchmarks.

We define the total fault population available in a given design as the number of gates multiplied by the number of time points at which each gate can receive a fault injection. For example, a design with 50K gates that runs a benchmark for 10 milliseconds and injects faults at a rate of 1 fault/μs would have a total fault population of 500 million injection points.

Figure 8 shows the cumulative distribution function of the error response for a single benchmark. The gates are arranged along the x-axis in increasing order of activity. A single curve in the graph shows the distribution of fault sensitivity within the 14000 gates in the OpenRISC processor design. A gate that is very sensitive to faults will show a steep increase in the response whereas fault-insensitive gates will appear as a flat line. It should be noted, that the shape of the curve is dependent on the ordering of gates along the x-axis, however the results and analysis presented here are independent of this shape.

The plot shows several curves with increasing number of fault injection points. As the number of injection points increases, the response curves tend to converge to a single overall response. Increasing the number of injection points to the total fault population would produce a deterministic experiment. As the number of fault injections increase, the response curve should converge to that of the deterministic fault response. We assume that a larger number of fault injections yields a more accurate measurement and consequently the experiment using the largest number of fault injections is the best and most accurate measurement.

We can quantify the variation seen in figure 8 by measuring the vertical distance from each line to that of the highest measured experiment (128K in figure 8), squaring it, and averaging it over the entire curve (i.e. all gates). Normaliz-

ing this by taking the square root gives us a measurement of standard deviation from the best measured response curve. As the standard deviation approaches zero, the accuracy of the experiment (as compared to a deterministic experiment) approaches 100 percent.

Figure 9 shows this standard deviation versus the fraction of the total population of fault injections. A clear relationship can be seen as the total number of injections increases. For every two orders of magnitude increase in the total number of injected faults the standard deviation decreases by roughly one order of magnitude.

It should be noted that even with our lowest injection fraction (i.e. 1.6×10^{-9} corresponding to 250 total injections) the standard deviation from the best obtained result is only 0.08. In other words even with injecting as few as 250 faults, the overall response of the circuit is within 8 percent of the best measured response. Furthermore, the relationship is an inverse power function and consequently the standard deviation decays very quickly to small values as the number of fault injections increases.

It should be clear from these results that statistical fault injection can be used to approximate fully deterministic fault injection with only a fraction of the total number of injection points and a minimal decrease in accuracy.

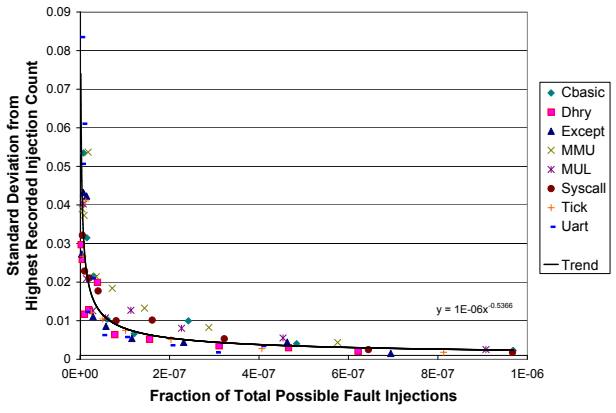

Figure 9. Accuracy quickly approaches 100 percent as the number of injected faults increases

5 Related Work

A fault injection methodology for VHDL is presented in [10]. The use of saboteurs has the same effect as our PLInject technique while the described mutants is somewhat analogous to our MODLIB. However the tool relies strongly on the VHDL language and consequently is not available for Verilog designs. Our methodology presented here can be applied to either VHDL or Verilog designs. Furthermore we incorporate statistical fault analysis into our models.

Alexandrescu et al. present a method for studying fault latching behavior (the likelihood that an injected fault is propagated and latched). Their technique is similar to our checkpointing technique however in [4] the fault injection occurs with latches disabled. Consequently faults are not allowed to propagate beyond the combinational logic being tested. This limits the information obtained during fault injection and prevents the evaluation of overall architecture behavior. Our technique does not have this limitation as faults are allowed to propagate anywhere within the simulated environment.

Mohanram and Touba discuss an approach for implementing targeted concurrent fault detection [14]. They demonstrate their algorithms on a set of synthesized benchmark circuits. However, it is not obvious from the paper if faults are actually injected into the circuits for observation or if the results are only based on error estimation using the presented models.

Mukherjee et al. present a non-statistical approach to characterizing faults in architecture designs [15]. They use a performance simulator to deterministically evaluate fault behavior. The authors note several advantages this has over statistical fault injection including no need to obtain confidence intervals. The authors note the importance of continued use of RTL designs with statistical fault injection when the RTL becomes available in the design cycle. The RTL designs offer more accurate fault-rate estimation as they include all aspects of the design and not just performance-related ones. Our work falls in the category of statistical fault injection with RTL designs. It provides a fairly simple and straightforward implementation of statistical fault injection that to our knowledge has not been introduced into the research community.

Wang et al. perform fault characterization on a custom-designed processor similar to the Alpha 21264 [25]. The processor is implemented as RTL and statistical fault injection used. However, their setup only injects faults into latches and registers. Consequently they're method cannot be used to consider combinational logic effects such as fault masking.

Cha et al. present a simulator for fault injection of transient faults in [9]. It uses two separate simulators: one for detailed timing of a transient pulse propagation and a second for zero-delay logic propagation. The justification is that the detailed timing of a transient pulse only needs to be simulated until it is captured in a latch at which point it resides as a simple logical error. For our implementation, detailed timing of transient pulses must be captured in the model provided to our algorithms. The statistical fault injection algorithms we present are analogous to the zero-

delay logical error simulator in [9] however, our algorithms are easily adapted to current VLSI simulators.

Our technique differs from software fault injection discussed in [5], [11], [19], [24] as we rely on simulation environment to inject faults anywhere into the simulated design. With software fault injection, faults are only injected into portions of real hardware that are directly accessible to software (e.g. registers or memory). This method has its advantages injecting faults onto real machines. However, it is greatly restricted in the types of faults that can be analyzed. Our approach is meant to be a generalized fault injector that is not limited in fault injection locations.

Kim and Somani look at fault characterization of the picoJava-II core processor [21]. They inject faults into behavioral code of the major components (FUBs) of the processor. Consequently they do not inject faults into individual synthesized gates. This has the advantage of less simulation overhead however it limits the type of fault injection experiments that can be performed. Our approach allows a finer granularity both in terms of time sensitive injection and injection location.

6 Conclusions

We've introduced two approaches to performing statistical fault injection into synthesized architecture designs. The PLInject method is clearly superior in terms of simulation time, however the MODLIB can be a more intuitive approach to fault injection if the simulation time permits. Furthermore, the MODLIB is designed entirely within VLSI language requiring no external interfaces.

Statistical fault injection can be largely beneficial for performing rapid analysis of a complex circuit or design. We've shown an example of this where we obtain the fault response to the OpenRISC 1200 processor under varying rates of fault injection. This example demonstrates how the statistical fault injection although not as rigorous as deterministic fault injection still can obtain useful results for circuit characterization. We have shown that the accuracy of our fault injection technique is good even with a relatively small number of fault injections and that the accuracy increases an order of magnitude with roughly two orders of magnitude increase in the number of faults injected.

We've also introduced a method to provide finer granularity of fault analysis which can lead to a more productive simulation environment. By utilizing checkpoint features found in most VLSI simulation software, we can increase the number of fault-check pairs in a circuit simulating an entire workload.

Acknowledgments

This work was supported in part by the Semiconductor Research Corporation under contract no. 2004-HJ-1190 and the Cross-Disciplinary Semiconductor Research Program, National Science Foundation grant CCR-0210197, IBM, the University of Minnesota Digital Technology Center, and the Minnesota Supercomputing Institute.

References

[1] Interuniversity microelectronics center. www.imec.be.

[2] Openrisc 1200 architecture. www.opencores.org.

[3] S. A. Al-Arian and M. A. Al-Kharji. Fault simulation and test generation by fault sampling techniques. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings., IEEE 1992 International Conference on*, pages 365–368, 1992.

[4] D. Alexandrescu, L. Anghel, and M. Nicolaidis. New methods for evaluating the impact of single event transients in ics. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 99–107, 2002.

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, 1990.

[6] D. Burger and T. Austin. The simplescalar toolset, version 2.0. www.simplescalar.com.

[7] C. Constantinescu. Estimation of the coverage probabilities by 3-stage sampling. In *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*, pages 132–136, 1995.

[8] M. Cukier, D. Powell, and J. Ariat. Coverage estimation methods for stratified fault-injection. *Computers, IEEE Transactions on*, 48(7):707–723, 1999.

[9] Hungse Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi. A gate-level simulation environment for alpha-particle-induced transient faults. *Computers, IEEE Transactions on*, 45(11):1248–1256, 1996.

[10] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mephisto tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66–75, 1994.

[11] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248–260, 1995.

[12] L. M. Kaufman, B. W. Johnson, and J. B. Dugan. Coverage estimation using statistics of the extremes for when testing reveals no failures. *Computers, IEEE Transactions on*, 51(1):3–12, 2002.

[13] R. M. McDermott. Random fault analysis. In *Proceedings of the 18th conference on Design automation*, pages 360–364. IEEE Press, 1981.

[14] K. Mohanram and N. A. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 893–901, 2003.

[15] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 29–40, 2003.

[16] F. N. Najm and I. N. Hajj. The complexity of fault detection in circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(9):995–1001, 1990.

[17] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for fault tolerance coverage evaluation. *Computers, IEEE Transactions on*, 44(2):261–274, 1995.

[18] C. Scherrer and A. Steininger. Dealing with dormant faults in an embedded fault-tolerant computer system. *Reliability, IEEE Transactions on*, 52(4):512–522, 2003.

[19] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat - fault injection based automated testing environment. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 102–107, 1988.

[20] N. Seifert, Xiaowei Zhu, and L. W. Massengill. Impact of scaling on soft-error rates in commercial microprocessors. *Nuclear Science, IEEE Transactions on*, 49(6):3100–3106, 2002.

[21] Seongwoo Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 416–425, 2002.

[22] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[23] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 2004. ACM Press.

[24] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J. J. Beahan. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *Dependable Systems and Networks, 2001. Proeedings. The International Conference on*, pages 501–506, 2001.

[25] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Dependable Systems and Networks, 2004 International Conference on*, pages 61–70, 2004.

[26] H. S. P. Wong, D. J. Frank, P. M. Solomon, C. H. J. Wann, and J. J. Welser. Nanoscale cmos. In *Proceedings of the IEEE*, volume 87, pages 537–570, 1999.

[27] C. Zhao, X. Bai, and S. Dey. A scalable soft spot analysis methodology for compound noise effects in nano-meter circuits. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 894–899, New York, NY, USA, 2004. ACM Press.