StressTest: An Automatic Approach to Test Generation Via Activity Monitors

Ilya Wagner Valeria Bertacco Todd Austin

Advanced Computer Architecture Lab
The University of Michigan – Ann Arbor, MI
{iwagner,valeria,austin}@umich.edu

ABSTRACT

The challenge of verifying a modern microprocessor design is an overwhelming one: Increasingly complex micro-architectures combined with heavy time-to-market pressure have forced microprocessor vendors to employ immense verification teams in the hope of finding the most critical bugs in a timely manner. Unfortunately, too often size doesn't seem to matter for verification teams, as design schedules continue to slip and microprocessors find their way to the marketplace with design errors. In this paper, we describe a simulation-based random test generation tool, called StressTest, that provides assistance in locating hard-to-find corner-case design bugs and performance problems. StressTest is based on a Markov-model-driven random instruction generator with activity monitors. The model is generated from the user-specified template programs and is used to generate the instructions sent to the design under test (DUT). In addition, the user specifies key activity points within the design that should be stressed and monitored throughout the simulation. The StressTest engine then uses closed-loop feedback techniques to transform the Markov model into one that effectively stresses the points of interest. In parallel, StressTest monitors the correctness of the DUT response to the supplied stimuli, and if the design behaves unexpectedly, a bug and a trace that leads to it are reported. Using two micro-architectures as example testbeds, we demonstrate that StressTest finds more bugs with less effort than open-loop random instruction test generation techniques.

Categories and Subject Descriptors

B.8.2 [**Performance Analysis and Design Aids**]: Architectural Simulation; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Simulation*

Keywords

Architectural simulation, High-performance simulation, Directed-random simulation

Design Automation Conference 2005

1. INTRODUCTION

Microprocessor verification is one of the major bottlenecks in the development of computing systems, in terms of time and especially verification engineer effort. Recently, the ITRS (an association of semiconductor companies) assessed that it takes thousands of engineer-years to develop top-end systems, yet processors still reach the market with 'hundreds of bugs' [3]. Moreover, more than twice as many resources are spent on verification compared to design in the microprocessor arena, bringing the design-to-verification gap to crisis proportions.

A variety of techniques have been deployed to efficiently and effectively detect design errors in microprocessors. Simulation based random testing is a long-standing approach used to locate design errors [5, 7, 9, 10, 12]. This technique generates random instruction sequences that are then fed in parallel to a design-under-test (DUT) and a known-correct golden model. Any discrepancies between the two models indicates a design error. Simulation-based random testing tends to be myopic for complex micro-architectures, where stateful logic blocks and complex interactions would require more simulation time than can be accommodated by time-to-market pressure.

Formal verification techniques have become a powerful mechanism to provide high-coverage verification. However, the intractability of performing formal verification on complex designs has limited the use of these technologies primarily to the verification of individual components. For example, Intel invested 60 person-years of formal verification effort on the Pentium 4, focusing on the verification of floating-point units, instruction decoding, and dynamic schedulers. While this investment proved to be quite effective, as no post-silicon bugs were found in the formally verified portions of the design [5], the limited scope upon which formal verification can be employed has not allowed this approach to replace simulation-based random verification. Even with a substantial formal verification team, the Pentium 4 was still primarily tested using simulation-based random testing.

One of the drawbacks of simulation-based random testing is that there is often a gap between what the designer wants to test and how often the random instruction sequence generator produces an effective test for the condition of interest. Consequently, to achieve good test coverage and expose hard-to-find bugs, specialized hand-written tests must be developed or significant human intervention and control must be exercised over the testing process. Moreover, the special

hand-written test cases are often not portable between different hardware designs of similar nature and have to be virtually created from scratch. A number of tools have been developed to enable verification engineers to have more control over the generation of random tests. In particular, some of these techniques involve the use of program templates that define the structure of the desired test, along with primitives to control the randomization of the related data, such as opcodes, register operands, and memory addresses [4]. Improvements on these baseline techniques use coverage metrics to drive test program generation, either through Markov models [11] (as in this work) or with Bayesian networks [8].

In this paper, we introduce a tool called StressTest that employs an innovative approach to automatic test generation. StressTest requires minimum interaction and control from the user, it is easily fine-tuned and highly portable, since it considers the design under test at a very high abstract level. Our approach poses very limited demands on the verification engineering team, by requiring only to provide a simple template which describes the interface protocol of the design. To assist the engineer in describing concise and meaningful programs, our template language includes a number of helpful features including parameterized dependency variables. Based on this template, StressTest can generate a very broad spectrum of different testbench programs to verify the design.

The underlying generation engine of StressTest uses a dynamically adjusted Markov model representing the set of all valid inputs to the design under test. This, first of all, indicates that the tool generates only valid test sequences, eliminating the possibility of false-negatives. Additionally, this approach combines advantages of both probabilistic and self-guiding stimulus generation techniques, which allows us to improve coverage while lowering overall verification effort. Moreover, the template-based approach for representing the input set allows for a very compact representation even for large amounts of inputs, and increases the portability and flexibility of StressTest.

For guidance in the test generation, StressTest uses activity monitors, which probe the internal state at user-specified points in the design. Using closed-loop feedback techniques, the measure of activity at these points is used to adjust the weights on the edges of a Markov model and direct the test generator engine towards instruction sequences with higher activity and higher degrees of interaction. We find that our approach achieves better coverage of complex bugs in fewer cycles than random open-loop techniques.

1.1 Contributions

The main contribution of this paper is the development of a novel closed-loop random test generation methodology that effectively produces adaptive instruction sequences to exercise user-specified micro-architectural activity points. Additionally, we present an innovative template-based approach to random stimulus generation that includes a flexible instruction specification technique and specialized random variable types that can be parameterized to produce a varied range of dependency and locality characteristics. Both of these features contribute to performance of the developed software as well as its flexibility and portability. Finally, we evaluate these techniques against open-loop random instruction generation and demonstrate that it can find more bugs with fewer instructions.

The remainder of this paper is organized as follows. Section 2 reviews random instruction testing and related work on the subject. Section 3 introduces StressTest and details the approach it uses for random test generation. Section 4 performs a systematic study of the performance of our proposed approach, by comparing the effort and number of bugs found (in two microprocessor designs) to open-loop random and less sophisticated closed-loop techniques. Finally, Section 5 draws conclusions and suggests future enhancements to StressTest.

2. BACKGROUND AND PRIOR WORK

The issues involved in developing and evaluating the performance of different random test generators (RTGs) for processor verifications have been a strong focus in the academic community and industry for quite a while. An overview of the area can be found in [6], which discusses the general framework of RTGs, comparison between random and directed testing approaches, and identifies several key properties that test generators must have to simplify the work of a verification engineer and improve performance. The generators are shown to be useful if their output is deterministic and reproducible, and the engineers have clear and effective ways of biasing the generator for directing a test towards a specific area of interest. Key characteristics of an effective RTG include grouping, or collective naming of sets of inputs with a short hand notation, simplicity of the language directing the generator, and, of course, ability to generate valid input sequences. As we will show later in the paper, all of these features are implemented in our verification tool.

In addition to implementing these major features, some of today's general-purpose random test generators attempt to dynamically direct the hardware verification process by analyzing results of each generated test. Tools like Specman Elite [2] and Vera [1] provide on-the-fly data assertion and checking and methods for validation of generated tests. In both cases the generation process is directed by dynamically changing constraints based on functional coverage analysis. Although these tools simplify the work of the end user with GUI and powerful verification languages, most of the test set up and decision process is still left to the verification engineer, who must specify functional test plans [2] or implement constraint adjustment policies [1].

A variety of research tools for directed random test generation were developed in academia as well as in industry. Most of them also employ a coverage-directed generation process, but use sophisticated techniques for representing relationships between coverage and input generation like Bayesian networks and computer learning [8]. Some of the other engines are aimed specifically at the register level representation of a design and focus on tag coverage [11], instead of functional coverage.

Our approach is different from the tools mentioned above in the way that we base the engine on an abstract representation of the input model, not the circuit itself. This allowed us to make StressTest virtually independent of a particular implementation of a circuit and thus more flexible and portable. Additionally, we designed the tool to do as much of the decision process internally as possible, easing the burden on the verification engineer. Finally, we created an easy to use language for representing the test generation rules in a compact and flexible format, again, simplifying the job of the user.

3. STRESSTEST ENGINE

StressTest provides a convenient platform for specifying instruction sets using instruction protocol templates with various dependency and locality parameters. A number of activity monitors observe a small set of relevant circuit internal signals and drive the generator toward scenarios that excite these signals. This section gives an overview of the tool.

3.1 Overall Structure

StressTest's self-guiding generation engine consists of two major components: a Markov model and a set of activity monitors (see Figure 1). The Markov model encapsulates the set of legal inputs of the design as well as the probabilistic information on generating sequences of inputs. The activity monitors bind to several key nodes or signals of the DUT and analytically evaluate the "stress" on the design due to the current input. The information collected by the activity monitors is used as feedback to the Markov model guiding the stimulus generation to maximize the design activity.

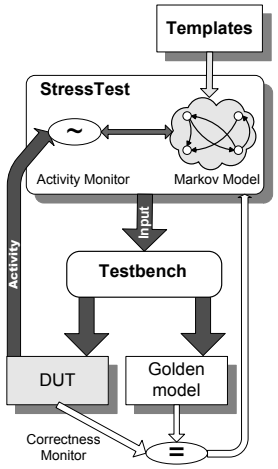

Figure 1: StressTest structure

StressTest connects to the external instruction bus interface of the DUT and internally simulates the instruction memory for the DUT. In other words, the execution of the test is not affected by the insertion of StressTest in the simulation flow. The input sequences generated by StressTest are relayed to a testbench, which is run both on the design under test and a golden model in lockstep. The golden model that we use is a functional representation of the DUT. The correctness monitor compares the events that alter the architectural state of both the golden model and the DUT. When a mismatch occurs, it is reported back to StressTest and the simulation terminates.

3.2 Markov Model

The Markov model in StressTest can generate all of the design's valid inputs. The model is represented by a directed graph where each vertex represents an interface transaction or set of transactions with similar behavior. The interface for our micro-processor DUT is the instruction bus, and each Markov model vertex represents a specific type of instructions. For instance, one node represents instructions with immediate operators, while another is used for branch instructions. Note, however, that this approach is not limited to microprocessors and, in general, transactions through an interface of any digital circuit can be represented similarly.

The weights of the edges of the Markov model are equal to the probability of transitioning from the source to the sink vertex, or in our case, from one instruction set sequence to another. At the beginning of the simulation the Markov model is a clique, and it is equally probable that a vertex will transition to any other vertex. To begin the instruction sequence generation, a starting vertex is selected at random and the system produces inputs according to the rules associated with this vertex. During the simulations the weights of the edges are adjusted based on the activities observed by the activity monitors.

3.3 Template Files

We use a special template language to describe the initial Markov model, an example of which is given in Figure 2. Our intention was to make this language as simple and comprehensible as possible, since the templates will be created and handled by the users of StressTest. However, in spite of the simplistic structure the template language we developed retains the ability to efficiently describe different sets of instruction sets, or, more generally, interface transactions, with varied interaction rules.

```
immVal   (cacheSize=5,probCache=0.9,lambda=2,
          minVal=-8192,maxVal=8191);
destIndex(cacheSize=30,probCache=0.8,lambda=4,
          minVal=0,maxVal=31);
randIndex(probCache=0,lambda=0,minval=0,maxVal=31);
r-funcs  (probCache=0,lambda=0,minVal=0,maxVal=63);
i-funcs  (probCache=1,lambda=0) =
{ 'b001000 /ADDI/,  'b001001 /ADDIU/, 'b001010 /SLTI/
   'b001011 /SLTIU/, 'b001100 /ANDI/, 'b001101 /ORI/,
   'b001110 /XORI/ };

vertex(r-type-inst)
{ input = 'b000000ssssstttttddddd00000ffffff;
   field(s) = $destIndex.read();
   field(t) = $randIndex.read();
   $destIndex.write(field(d));
   field(f) = $rfuncs.read(); }

vertex(i-type-inst)
{ input = 'bffffffssssstttttiiiiiiiiiiiiiiii;
   field(f) = $ifuncs.read();
   field(s) = $destIndex.read();
   field(t) = $randIndex.read();
   $destIndex.write(field(d));
   field(i) = $immVal.read(); }
```

Figure 2: Example of a template file

At the top of the template file are the random variable definitions. Random variables are the source of random bit-field values, and have special support for producing values with specified locality and dependency characteristics. Declaration of random variables and their operation are detailed in Section 3.4.

Following the random variable definitions are the vertex specifications for the Markov model. Each vertex contains one or more definitions of the format of the inputs. The definitions are in binary format, and specify the values of each bit in the instruction value to be generated. Values may be a constant 0 or 1, or they may be parametric bit fields specified with a single letter repeated for each bit position.

Below the input specification, each of the fields is assigned a value from the random variables.

Multiple vertex definition may be contained within one template file. In addition, each instruction class may have multiple vertex definitions for it included in the same template file. This capability enables instructions with different properties to be generated with different probability. For example, arithmetic instructions could be described as highly dependent with previous instructions in one vertex, while nearly always independent in another vertex. This makes it possible for the activity monitors to selectively adjust transitions to and from vertices with specific individual properties.

3.4 Dependency Variables

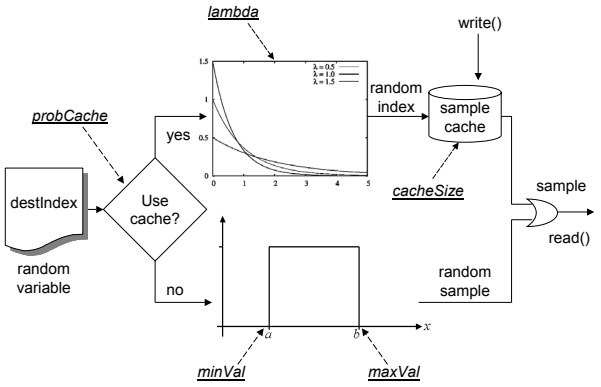

Figure 3: Random Variable Abstraction

The dependency variables, declared at the top of the template file in Figure 2, provide a concise mechanism to specify the generation of random values from i) a list of constants, ii) uniform random distributions, iii) randomly generated locality sets, or iv) some combination of the previous three constructs. The variables are used to pass information between the template fields and vertices and create complex interactions, such as locality and dependency, between generated inputs. All of the variables have global scope and can be accessed from anywhere in a template file.

The abstract construct used to represent random variables is illustrated in Figure 3. As shown by the underlined labels in the figure, variables are defined with five declaration parameters:

probCache is the probability that a *read()* access to a random variable will result in the access to the locality cache.

cacheSize is the size of the locality cache, which contains the most recently generated values that can be reused to simulate locality and dependencies.

lambda is the degree of locality/dependency parameter, implemented as the parameter to an exponential distribution that generates cache indexes, *i.e.*, the larger this value, the more skewed towards recent cache items is the selection, the smaller the value, the more uniform the accesses to the cache elements.

minVal,maxVal are the bounds over which the uniform random samples (accessed in lieu of the sample cache) are produced.

When the variable is *read()*, the returned value is either taken from the sample cache with probability *probCache*, or produced by the random generator with probability *1-probCache*. The uniform random generator associated with a variable can be bounded to produce only specific values from a list of constants or from a user-specified range. The sample cache is used to simulate storage locality and dependency. When the *write()* method is invoked on a random variable, the supplied value is added to the sample cache. In the event that this cache is full, the oldest cache element is removed.

When a value is taken from the sample cache, λ is used as the parameter of the exponential distribution function to generate the index of the entry in the cache to return. Note that, high λ values correspond to a high probability of generating low list indexes, and thus returning recent data. When the value of λ is small, reads from the cache are almost uniformly distributed among entries. Thus, it is possible to control the level of locality in the sample cache accesses by varying λ. This mechanism can be used in our context to control the degree of register dependence between generated instructions. The probabilistic distribution of the returned values allows to efficiently control the strength of the dependency between instructions that operate on same variable and, therefore, the "stress" on the control and forwarding logic of such DUTs.

3.5 Activity Monitors

In designing StressTest, we made an assumption that many bugs arise from complex interactions between instructions, which also create high activity at the interface between units and control blocks. Thus, we are more likely to find bugs by generating patterns that cause a high level of activity in those interface signals, rather than by simulating random input sequences.

The activity monitors are responsible for identifying input sequences of interest and then updating the Markov model edges to reinforce them. Figure 4.a illustrates the approach for one of the microprocessor cores that we targeted in our experiments. The Markov model sends stimuli, in the form of instructions, to the DUT. At each cycle, the activity monitors assess the control signals in the DUT for specific activities of interest. Examples of activities of interest would be high physical register file pressure, highly dependent instructions in the register renamer, or high cache miss rate. Once an activity of interest is identified, the transition in the Markov model leading to the generation of this activity is reinforced.

One challenging aspect of implementing the activity monitors was maintaining the association between a particular Markov model transition and the resulting activity monitoring event. The challenge exists because model stimuli are generated many simulation cycles before the triggered activity. To maintain this binding in our experiments we tagged all stimuli instructions with the edge transition that led to their generation. At the completion of each simulation cycle, stimuli and activity events are matched through the tags and the transition leading to the stimulus generation is reinforced in the Markov model.

Figure 4 illustrates an example of activity monitor, and its effect on the Markov model. In the experiment, shown in Figure 4.a, the Markov model for the DEC Alpha instruction set is used to generate stimuli for our testbed Alpha micro-

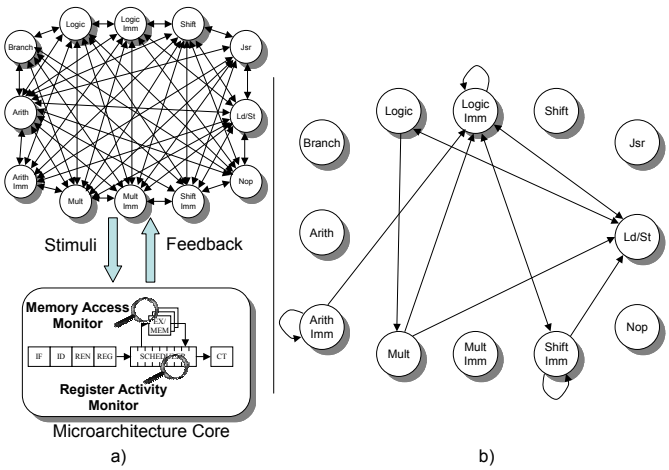

Figure 4: Activity Monitor Technique and Example

architecture. Two activity monitors are engaged during the experiment: a memory access monitor that encourages frequent memory accesses and a register file activity monitor that pushes for lots of accesses to the register file with various data values.

Figure 4.b illustrates the state of the Markov model after 8000 cycles of operation. In our implementation we kept the Markov model always fully connected, and, for clarity, we show here only the top ten most probable transition edges. As shown, the Markov model quickly morphs into a graph that generates a significant number of memory access, due to the many edges pointing toward the memory generation node. Moreover, the use of instructions with immediate operands increases the range of values written to the register file (because the immediate values are random), thereby reinforcing the register file activity monitor. Shift and multiply operations also contribute to this monitor by generating significant variations in the output results. Finally, branches and jumps are less frequent since they cause little excitement in the register file.

Activity monitors are highly flexible constructs, capa of quickly identifying both performance and design b Other examples of their use include stressing the activit collision signals between different ports of a network swi thus checking the correctness of the switch at high utiliza points, or stressing a pipeline recovery mechanism with quent mispredicted branches by generating branch inst tions with low locality in the target addresses.

4. EXPERIMENTAL RESULTS

In this section, we first introduce our experimental e uation framework, and the test designs we used. Then, evaluate the performance of our proposed technique aga an open-loop random instruction generator, comparing b coverage of bugs and number of simulated instructions quired to expose those bugs.

4.1 Experimental Framework

To test the performance of StressTest we conducted a ries of simulations on two Verilog processor core desi The first one is a 5-stage DLX pipeline running MIPS- ISA with branches resolved in the ID stage. The second sign is a 5-stage pipeline running Alpha ISA with branches resolved in the EX stage and two-cycle stores. We created single-cycle golden models for both pipelines, which are effectively a functional description of the cores and often can be derived from specifications of the design.

The pipeline under test and the single-cycle golden model were connected to independent data memories, and interfaced to StressTest. We connect these components through a small Verilog testbench framework. StressTest itself is implemented in VERA verification language with some inserts of C and C++ code.

It should be noted that in a classic static Markov model, which we used in the first implementation of the system, the transition from one state to another is conditionally independent on the history of the previous transitions. That, however, is insufficient to represent the complexity of interactions between instructions, and leads to lower coverage and high effort as shown from the our results. The dynamically adjusted weights and variables that we used to propagate information from past instructions to future inputs help us generate streams that are much more representative of real software and are able to greatly reduce effort while increasing coverage.

4.2 Results

The set of tests for the first experiment consisted of 29 buggy DLX core designs, where bugs varied from simple (such as incorrect operation for a given arithmetic opcode) to complex ones, involving forwarding logic and interactions through memory. We ran the system 25 times for each of the buggy designs with different random seeds to calculate average effort and coverage. All tests were run until the correctness monitor had recognized an error or a maximum of 3000 instructions were generated. Then the system would terminate and the generated stream would be recorded. The set of tests for the second experiment used nine buggy Alpha pipelines with moderate and very complex bugs, mostly special corner cases for the pipeline forwarding logic. The experiment was identical to the first one otherwise.

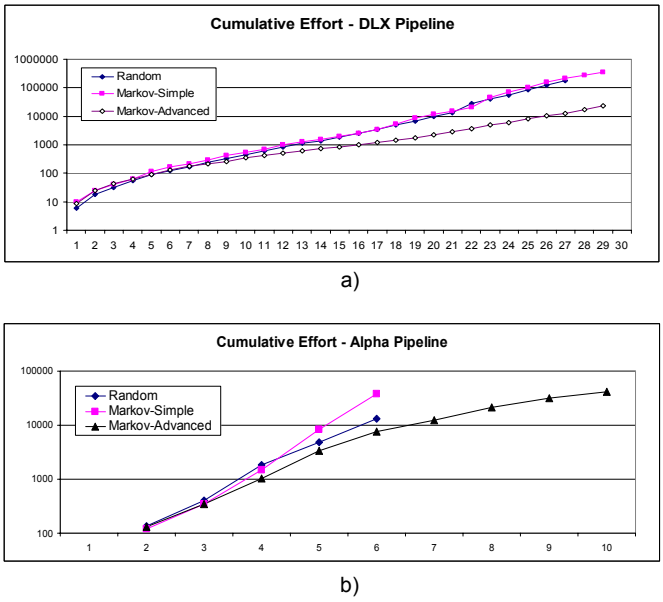

Figure 5: Effort versus Bugs Covered

Figure 5 graphs the results of our analyses. For performance comparison, random test generation was implemented in three ways. *Random* utilizes only a static Markov model of the ISA and does not collect performance feedback from the DUT. *Random* also uses several random variables, but without the sample cache feature, *i.e.*, the parameter *probCache=0.0*. *Random* constitutes a fairly capable open-loop random testing solution. *Markov-Simple* utilizes a feedback-adjusted Markov model based on activity feedback from the pipeline under test, however, it too lacks the sample cache capability in its random variables (as *Random* does), thus it generates all instruction fields at random. Finally, *Markov-Advanced* uses both the feedback-adjusted Markov model and random variables with sample caches. The variables were used to transfer destination register IDs to source register fields, and share arithmetic immediate values as well as memory and branch offsets between generated instructions. The three random test generation techniques monitored the DUT correctness in the same way, by running pipeline and golden model in lockstep and checking correctness of register and memory writes, as well as behavior of the program counter.

Figure 5 shows the results of the three random test generation approaches, applied to the DLX and Alpha processor pipelines. For each, the graph illustrates the cumulative *effort* (in total instructions executed) versus the total number of bugs detected. To effectively distinguish between easy-to-find bugs from harder ones, we have sorted the bug list in ascending order by total number of instructions to locate the bug. As a result, the bugs on the left of the graph were easier to locate than the bugs on the right of the graph. When a technique was incapable of detecting all of the bugs (as is the case for *Random* for both processors), the curve terminates short of the far right bug.

As shown in Figure 5.a, the *Markov-Advanced* and *Markov-Simple* achieve better coverage than the *Random* model, by detecting two additional bugs. In addition, the *Markov-Advanced* was far more efficient than *Markov-Simple* at detecting all the bugs, requiring 94% fewer instructions to cover all of the bugs. It is also interesting to note that *Random* performed more efficiently for the easy bugs, requiring about 1/3 fewer instruction to locate them.

Figure 5.b shows the random testing results for the Alpha pipeline. In this experiment, the *Markov-Advanced* achieves significantly more coverage than *Random* and *Markov-Simple* techniques, detecting four additional bugs. As in the previous experiment, *Markov-Advanced* is also more efficient at locating bugs, except for the easiest-to-find bugs, which, again, *Random* was best at discovering.

It should be also pointed out that during the initial phase of the system development the authors were able to identify three major bugs in the forwarding logic of the DLX pipeline that initially was assumed to be correct. Although this pipeline was a subject of verification projects for several years, these bugs were still undiscovered. The features of StressTest that led to the discovery of these bugs were: 1) running the DUT in lockstep with the golden model to check correctness, 2) generating instructions using templates, and 3) passing information between instructions using variables. Without these techniques the bugs would have been extremely hard to find and would have required significant user effort in directing the test.

5. CONCLUSIONS

In this paper, we described a novel approach to closed-loop random input generation. Our approach, implemented in a tool called StressTest, is based on a Markov model that contains templates for generating instruction sequences. These templates are designed by verification engineers to resemble directed tests. Our template language is particularly expressive, in that it supports generation of a wide range of input types with varied dependency and locality characteristics and can be used in verification of processor cores or other digital circuits. Moreover, the verification engineer needs to identify key activity signals in the design, *i.e.*, signals that are indicators of "stressful" operation or are suspected to hold performance or design bugs. A closed-loop feedback engine then adjusts the Markov model, based on the monitoring of the activities, to produce effective and efficient tests. Evaluation of StressTest found that it is capable of finding more bugs in fewer cycles that open-loop random and less sophisticated closed-loop test generation techniques.

Looking ahead, we are extending this work in a number of ways. First, we are extending the language to support a more effective specification of the data values, along with the instructions that access them. Second, we are exploring the application of the StressTest infrastructure to other application domains. In particular, we are working to deploy the same mechanisms to perform feedback-directed random test generation on communication protocols and interfaces. In the context of this work, protocols can be specified and verified at the hardware level, such as the MAC or IP layer for the 802.3 protocol, or at the software level, such as HTTP.

6. REFERENCES

[1] Constrained-random test generation and functional coverage with Vera. Technical report, Synopsys, Inc, Feb. 2003.

[2] Specman elite - testbench automation, 2004. http://www.verisity.com/products/specman.html.

[3] A. Allan, D. Edenfeld, J. William H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian. 2001 technology roadmap for semiconductors. *IEEE Computer*, pages 42–53, Jan. 2002.

[4] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov. Industrial experience with test generation languages for processor verification. *DAC 2004*, June 2004.

[5] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 224–228, 2001.

[6] E.A.Poe. Introduction to random test generation for processor verification. Technical report, Obsidian Software, 2002.

[7] J. M. L. et.al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM Journal of Research and Development*, 46:53–76, Jan. 2002.

[8] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC, Proceedings of Design Automation Conference*, pages 286–281, June 2003.

[9] Y. Levhari. Verification of the PalmDSPCore using pseudo random techniques. Technical report, VeriSure Consulting, Ltd., 2002.

[10] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin. System-level validation of the Intel Pentium M processor. *Intel Technology Journal*, 07:38–43, May 2003.

[11] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A functional validation technique: Biased-random simulation guided by observability-based coverage. *ICCD, Proceedings of the International Conference on Computer Design*, pages 82–88, 2001.

[12] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor: The DEC Alpha 21264 microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 638–644, 1998.