Microprocessor Verification via Feedback-Adjusted Markov Models

Ilya Wagner, *Student Member, IEEE*, Valeria Bertacco, *Member, IEEE*, and Todd Austin, *Member, IEEE*

***Abstract*—The challenge of verifying a modern microprocessor design is an overwhelming one: Increasingly complex microarchitectures 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 does not seem to matter in verification, as design schedules continue to slip and microprocessors find their way to the marketplace with design errors. In this paper, we describe a novel closed-loop simulation-based approach to hardware verification and present a tool called StressTest that uses our methods to locate 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 files and is used to generate the instructions sent to the design under test (DUT). In addition, the user specifies key activity nodes 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 user-selected points of interest. In parallel, StressTest monitors the correctness of the DUT response and, if the design behaves against expectation, it reports a bug and a trace leading to it. Using two microarchitectures as example testbeds, we demonstrate that StressTest finds more bugs with less effort than open-loop random instruction test generation techniques.**

***Index Terms*—Architectural simulation, directed-random simulation, high-performance simulation.**

I. Introduction

MICROPROCESSOR verification is one of the major bottlenecks in the development of computing systems, in terms of time and engineering effort. Recently, the International Technology Roadmap for Semiconductors (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" [2]. In the microprocessor arena, more than twice as many resources are spent on verification compared to design, bringing the design-to-verification gap to crisis proportions. For example, Intel had recently estimated that by 2007, microprocessors would require 2000 person–year of verification effort [14].

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 [4], [5], [10], [13], [16]. It generates random instruction sequences that are then fed in parallel to a design under test (DUT) and to a known-correct golden model. Any discrepancy between the two models indicates a design error. This technique, however, tends to be myopic for complex microarchitectures, where stateful logic blocks and complex interactions require more simulation time than can be accommodated by time-to-market pressure.

Formal verification techniques have become a powerful mechanism to deliver high-coverage verification. However, the intractability of applying formal methods to complex designs has limited the use of this technology primarily to the analysis of individual components. For example, Intel invested 60 person–years of formal verification effort in the Pentium 4, focusing mainly on the verification of floating-point units, instruction decoders, and dynamic schedulers [4]. While this investment proved to be quite effective, as no post-silicon bugs were found in the formally verified portions of the design, the limited scope upon which formal verification can be deployed 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 constrained random validation.

The major drawback of the mainstream simulation-based approach is the difficulty of producing effective stimuli for a specific behavior of the circuit which the designer wants to test. Consequently, to achieve good test coverage and expose hard-to-find bugs, specialized hand-written tests must be developed, or significant control must be exercised over the test generation through human intervention. Unfortunately, hand-written test cases are often not portable, even across multiple proliferations of the same hardware design, and must be virtually recreated from scratch each time. To enable accurate control over constraint-based random test generation, a number of tools have been developed. Some of these techniques involve the use of program templates which 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 [1], [3], [8], [12], [17]. Improvements on these baseline techniques use coverage metrics to drive the generation of the test programs either through Markov models [15] (as in our solution) or with Bayesian networks [7].

In this paper, we introduce StressTest, a software tool which employs an innovative approach to automatic test generation. StressTest requires minimum interaction and control from the user, and it is easily fine tuned and highly portable, since it considers the DUT at a very high abstraction level. Our approach only requires the engineering team to provide a simple template describing the interface protocol of the design.

Manuscript received November 1, 2005; revised March 29, 2006. This paper was recommended by Associate Editor N. K. Jha.

The authors are with the University of Michigan, Ann Arbor, MI 48109 USA.

Digital Object Identifier 10.1109/TCAD.2006.884494

0278-0070/$25.00 © 2007 IEEE

To assist the engineer in describing concise and meaningful programs, our template language includes a number of helpful features, including parameterized dependence variables. Using this template, StressTest generates a very broad spectrum of testing programs to verify the design. The underlying generation engine of StressTest uses a dynamically adjusted Markov model representing the set of valid inputs for the design. This engine implicitly limits the generator to only produce valid test sequences, excluding the risk of false negatives. Additionally, this approach combines advantages of both probabilistic and self-guiding stimulus generation techniques, which allows us to improve the design coverage while lowering the overall verification effort. Finally, the template-based approach allows for a very compact representation, even for designs with complex input constraints, and increases the portability and flexibility of StressTest.

For guidance in the test generation, StressTest uses a novel technique based on activity monitors, which are simulation monitors that probe the internal state of the design at specific points. A first selection of probing nodes is made by the user, based on the key aspects or components, which, he wants to test. These nodes typically include key internal signals that reflect major changes in the design's state or outputs (for instance, a write-enable signal to the register file). StressTest complements this user selection with additional probing nodes highly related to the first ones so to achieve better performance and coverage. During simulation, StressTest observes the activity of all the probing nodes. Closed-loop feedback techniques are used to direct the test generator engine toward stimuli, generating higher switching activity at the probing points. Unlike previous approaches, which usually adjust the stimuli generator based on the results of a full completed test run, StressTest is capable of adjusting the Markov model dynamically during the test. The use of activity monitors enables the StressTest to gradually increase the stress on the tool-selected probing nodes and, subsequently, on the user-specified nodes themselves. We find that our approach achieves a better coverage for complex bugs in fewer cycles than constrained open-loop random generation techniques. Moreover, the addition of tool-selected probing points is a key in boosting the StressTest's performance and in generating effective tests in an assertion-based verification methodology.

A. Contributions

The main contribution of this paper is the development of a novel closed-loop random test generation methodology which effectively produces adaptive instruction sequences to exercise user-specified microarchitectural activity points. Additionally, we present an innovative template-based approach to random stimulus generation, which includes a flexible specification technique and specialized dependence variables that can be parameterized to produce a broad range of dependence and locality characteristics. The feedback for the stimulus generation engine is provided by a combination of user-selected and tool-generated activity monitors, which together produce a highly accurate mechanism used to reinforce favorable stimuli. We call the tool-generated activity monitors as "depth-driven," because they are selected based on their depth in the cone of logic of a user-selected node. Our approach is novel.

1) The granularity of the closed-loop feedback adjustment is much finer. In fact, we adjust the stimulus generation engine after each simulation cycle, in contrast with previous solutions which apply adjustments only after an entire test completes or through human intervention.
2) We developed a simple but powerful language to describe complex rules that must be enforced at the design's input in a constructive and straightforward manner, thus simplifying the engineer's job.
3) The stimulus generator is based on the high-level functionality of the system under verification, not its circuit structure. This enables the StressTest methodology to be virtually independent of a particular implementation, and thus more flexible and portable than other solutions.

All of these features contribute to the performance of the developed software as well as its flexibility and portability. The final contribution of this paper is insightful analysis of the impact of StressTest simulation parameters on its performance and quality of generated tests. Additionally, we evaluate our verification techniques against a closed-loop and an open-loop random instruction test generators, and show that our approach can find more bugs in shorter time and produce more consistent and predictable results than both other methods.

The remainder of this paper is organized as follows. Sections II and III review related work on the subject and provide an overview of StressTest. In Section IV, we describe the stimulus generation engine and discuss our template framework, including special constructs to force dependence between instructions. Section V presents depth-driven monitors and shows how they narrow the search for hard-to-find bugs. Sections VI and VII discuss our results through a case study and performance analyses over two designs and a range of bugs. Section VIII concludes and outlines future enhancements.

II. 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] and [17], which discuss the general framework of RTGs, compare random and directed testing approaches, and identify several key properties that test generators must have in order to simplify the verification task and improve the performance. RTGs are shown to be useful if their output is deterministic and reproducible, and if engineers have clear and effective ways of biasing a test toward a specific area of interest. Moreover, several key features can be exploited to increase the usability of a generator and, thus, the productivity of the verification process. These include grouping or collective naming of sets of inputs with a short hand notation, the ability to generate only valid input sequences, and, of course, the availability of simple test specification languages. As shown later, all of these features are included in our verification tool. In addition to implementing these features, some of today's general-purpose RTGs attempt to dynamically direct the

verification process by analyzing the coverage achieved by each generated test. Tools such as Specman Elite [9] and Vera [8] provide on-the-fly data assertion and checking, and methods to validate the generated tests. In both cases, the process is directed by dynamically biasing a set of constraints, based on the functional coverage or lack thereof, achieved by previous tests. Although these tools simplify the engineer's work through powerful verification languages, most of the test setup and decision process are still left to engineers, who must specify functional test plans [9], or implement constraint adjustment policies [8].

Typically, RTGs are deployed in conjunction with a golden model of the design. A golden model is essentially an abstract representation of the design, which may lack performance-oriented features but has the same architectural state space as the DUT. Therefore, it is possible to simulate design and golden model in parallel, supplying both with the same stimuli and observing the changes in their architectural state. Any discrepancy between the two models flags a potential design error. An alternative approach involves the use of hardware checkers, or assertions, embedded in the design. Hardware checkers monitor different activities of a design during simulation and flag unexpected behaviors observed. Checkers are more suitable for monitoring complex conditions that may arise in the control portion of a design, while a golden model is a complete, although simplified, description of the entire system's functionality. While golden models are almost always available in the context of microprocessor verification, hardware checkers have started only recently to be developed.

A variety of research tools for directed random test generation have been developed in academia as well as in industry. Most of them employ coverage-directed test generation processes, as in StressTest, but use sophisticated techniques to relate input generation to coverage, such as Bayesian networks and computer learning approaches [7]. Some of these other engines are aimed specifically at register-level representations of a design, and focus on tag coverage [15] instead of functional coverage as StressTest.

III. StressTest Structure

StressTest provides a convenient platform for specifying the set of valid inputs for a DUT by mean of templates. A number of activity monitors observe a small set of relevant circuit's internal signals and drive the generator toward scenarios that excite those signals. StressTest's self-guiding generation engine consists of two major components: a Markov model and a set of activity monitors (see Fig. 1). The Markov model encapsulates the set of legal inputs of the design as well as probabilities of generating different sequences of inputs. The activity monitors bind to several key nodes, or signals, of the design and evaluate the "stress" on the design based on the switching activity of these signals. The information collected by the activity monitors is used as feedback to the Markov model, which adjusts the probability to generate stimuli which maximize the stress on these nodes. The signals selected for the activity monitors drive the focus of the test generation and, consequently, which components of the design will be most thoroughly tested.

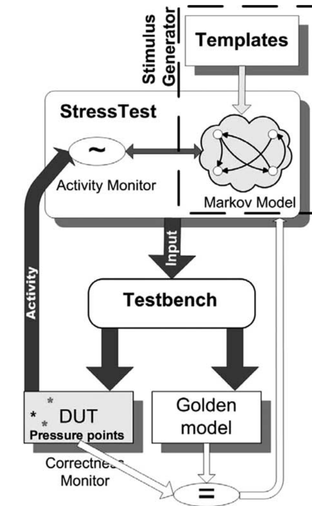

Fig. 1. StressTest structure. Templates are used to create a Markov model which generates valid stimuli: Templates' transactions are mapped to the vertices of the model. Stimuli are fed in parallel to the DUT and the golden model. Activity monitors observe the behavior of the DUT during the simulation and adjust the Markov model improving the quality of the stimuli. The outputs of DUT and golden model are compared to expose bugs.

Therefore, in order to locate bugs within a critical component, activity monitors should be selected from key signals which activate the component and within the component itself.

As with other RTGs, the stimuli generated by StressTest are supplied to both the DUT and a golden model, which is a functional description of the design usually provided in a high-level language such as C or C+. The output and the architectural state of the two descriptions are monitored, and discrepancies are flagged as potential design errors. For instance, in our tests on microprocessor pipelines, the golden model was a single-cycle functional model, and StressTest was connected to the external instruction bus interface of DUT and golden model. Hence, neither of the models could detect that the instructions were provided by a test generator instead of main memory. This greatly simplifies the test setup, allowing StressTest to reuse the framework of the directed tests. When the correctness monitor detects a mismatch in the architectural states of the two descriptions, it halts simulation and outputs a trace leading to the problem.

IV. Stimulus Generator

The stimulus generator engine of StressTest comprises an adaptive Markov model whose vertices describe monolithic blocks of stimuli or transactions. The transactions corresponding to each vertex are generated based on the template files provided by the user. Template files are loaded by the StressTest at the beginning of each simulation run and are used to generate the initial Markov Model. The probabilities associated with the model's edges are then adjusted over time based on the direction from the activity monitors. Several reasons led us to choose Markov models as the underlying engine for stimulus generation. First of all, Markov models are simple to implement and straightforward to train; second, they provide a convenient abstraction for modeling instruction generation: Instructions (or classes of instructions) constitute the vertices of the Markov model, while programs correspond to paths in the model's

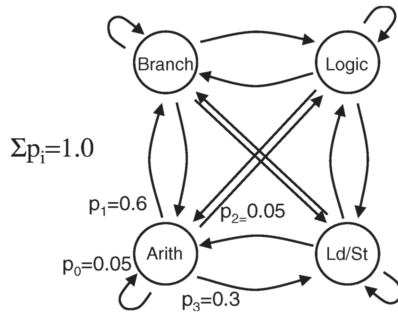

Fig. 2. Example of Markov model. The model has four vertices, each capable of generating a particular instruction type. The sum of probabilities to transition from a single vertex must be 1.0. In the example, once an arithmetic instruction is generated, a branch has probability of 0.6 to occur, a load/store has probability 0.3, while other instructions are less probable.

graph. In contrast, structures such as Bayesian networks must be "trained" on an actual circuit to determine the cause/effect relation between inputs and observed outcomes; therefore, unlike Markov models, they depend on the circuit's implementation. On the other hand, a noticeable drawback of Markov models is their lack of "memory." Each generated stimulus is independent from past stimuli and relates only to the last vertex visited in the model. Therefore, we introduced the concept of dependence variables to provide deterministic dependences between past stimuli and present input, as discussed later in this section.

A. Markov Model

StressTest uses a Markov Model as the main engine for generation of stimuli to the design. In general, a Markov model is a directed graph, where each edge has an associated probability of transitioning from its source vertex to its sink vertex. In StressTest, the nodes of the Markov model correspond to valid stimuli or groups of stimuli for the design. During the simulation, a stimulus associated with the current node is generated and supplied to both DUT and golden model. Within the framework of microprocessor verification, the stimuli corresponding to a single vertex can be individual instruction or groups of instructions forming a program fragment. A single vertex can represent a range of similar individual instructions by parameterizing the various instruction fields. An example of a simple Markov model for an abstract instruction set architecture (ISA) is shown in Fig. 2. The vertices in this case represent the different types of assembly instructions, such as branches, arithmetic instructions, and so on. Edges are labeled with the probability of being traversed. For instance, the probability of generating a branch instruction after an arithmetic instruction is p_1 which is equal to 0.6. Note that the sum of the probabilities associated with the outgoing edges from a given vertex must be equal to one.

This Markov-model-based approach is not limited to microprocessors only and can be used to represent transactions through an interface of any digital circuit. In our experiments, we used mutually disjoint and cumulatively exhaustive ISA partitioning, so that we could generate every instruction type, increasing the possibility of catching a variety of design errors. In simulations aimed at exposing a specific kind of bug, the Markov model can be structured to contain only a small subset of inputs representing distinct transactions. At the beginning of a simulation, the model is a clique, with each transition being equally probable. A starting vertex is selected at random; the system produces the corresponding inputs and randomly selects a transition to the next vertex. During the simulation, probabilities associated with the edges are adjusted based on the feedback from the activity monitors. StressTest allows for a sequence of instructions to be associated with a single vertex. When such a vertex is reached, the StressTest generates instructions until it reaches the end of the sequence, and only then it transitions forward. This allows for generation of deterministic instruction sequences. For instance, a vertex could map to a sequence of a branch instruction followed by a noop to generate legal assembly code for a microarchitecture relying on the compiler to resolve control hazards. Notice that, since the entire sequence is clustered into a single vertex, all the activity observed when executing the sequence will affect the probability of reaching that sequence again.

The probability adjustment algorithm considers a weighted sum of the switching activity reported by the monitors and converts it to a value which we refer to as "score." The score is used to adjust the probability associated with the edge E traversed during the last transition. The update is computed as follows: We first scale the score as a fraction of the maximum score that all edges could achieve; then, we adjust the probability of the relevant edge P_E; finally we readjust all the values so that the sum of probabilities is normalized to one. The probability increment is then

$$P_{\mathrm{inc}} = \frac{\mathrm{score}_{\mathrm{monitor}}}{\mathrm{score}_{\mathrm{max}} * N_{\mathrm{edges}}}$$

where N_{edges} is the number of outgoing edges from the vertex under consideration and $\mathrm{score}_{\mathrm{max}}$ is the maximum score achievable. The adjusted probability is the saturated sum

$$P_{E_\mathrm{new}} = \max\{T_{\mathrm{sat}}, P_E + P_{\mathrm{inc}}\}$$

where T_{sat} is the saturation threshold unique for the system. We set the saturation threshold T_{sat} to a value slightly less than one, so that we can always attribute at least a small probability to any edge. Doing so guarantees that we never eliminate potentially useful transitions from the system. In most of our experiments, we set $T_{\mathrm{sat}} = 0.95$, allowing other edges to have a probability in the range $[0.05/(N_{\mathrm{edges}} - 1), 0.95]$. If a vertex has an outgoing edge with probability T_{sat}, then all other edges are set at $P_{\min} = (1 - T_{\mathrm{sat}})/(N_{\mathrm{edges}} - 1)$.

The last phase of the probability adjustment requires that the sum of the outgoing edges' probabilities is renormalized. We do so by decrementing all eligible edges (the ones with $P_{\mathrm{edge}} > P_{\min}$) by an amount proportional to P_{inc}. Specifically, we first compute the "slack" available as

$$\mathrm{slack} = \sum_{i \neq E} (P_i - P_{\min})$$

where the contributing edges include all but the one we just incremented (note that edges for which $P_i = P_{\min}$ contribute

```
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();
   field(d) = $randIndex.read();
   $destIndex.write(field(d));
   field(f) = $r-funcs.read(); }

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

Fig. 3. Example of a template file. This file defines a Markov model with two vertices: r-type-inst (register type instructions) and i-type-inst (immediate type instructions). The vertices use five dependence variables declared at the top. In particular, variable $dest-Index creates a potential dependence between consecutive stimuli.

zero). The adjustment is then performed by decrementing each P_i proportionally to their contribution to the "slack"

$$P_{i_\text{new}} = P_i - \left(\frac{P_i - P_{\min}}{\text{slack}}\right) * P_{\text{inc}} \qquad \forall i \neq E.$$

This computation normalizes the probability while guaranteeing that each edge has $P_i \geq P_{\min}$.

B. Template Files

We use a special template language to describe the stimuli to generate at each vertex of the Markov model. StressTest's templates describe a short sequence of stimuli for each vertex, as shown in the example of Fig. 3. Stimuli definitions are in binary format and specify the values of each bit of the input signals. Values can be specified as zero, one, or through a parametric field. Each bit field is identified by a single alphabetic character repeated for the whole width of the field. Note that our stimulus generator only describes legal input stimuli, requiring no information about the DUT structure, making template files highly portable among multiple designs. The user must only specify the desired partitioning of the input set, and no explicit sequencing control is required. In addition, StressTest templates do not specify the biasing of input stimuli, since this is derived during the simulation. This is different from industry tools, such as Genesys-Pro [1], where sequencing control and biasing information must be provided in the template to achieve high-quality test sequences. The structure of our template files is particularly suitable for describing ISAs and very structured interface protocols. For instance, we could develop the template for the simple DLX pipeline described in Section VII in less than one person–week, and we could reuse much of the structure for the other testbench, an Alpha pipeline.

Template files can also force interactions between vertices, allowing StressTest to generate stimuli sequences with complex interdependences. To describe these interactions, we have created a structure very flexible and easy to use, called dependence variables. Dependence variables, declared at the beginning of a template, generate values for the parametric fields, and include special support for specifying locality and dependence characteristics. These variables and their operation are detailed in Section IV-C. Note that edges do not appear anywhere in template files, since our Markov model always starts as a single large clique. In designing the template language, we strove to keep a simple and intuitive structure, since this is the only part of StressTest which requires user input. In spite of its simplistic structure, the template language retains the ability of describing a very broad range of stimuli and many different interaction constraints.

The template language allows to represent groups of stimuli through multiple vertices, each using different parameters. This enables the StressTest to associate distinct probabilities to stimuli with different characteristics. For instance, arithmetic instructions could be highly dependent on previous instructions in one vertex, while nearly always independent in another, allowing the activity monitors to selectively adjust transitions to and from vertices with specific individual properties. An example of a template file is given in Fig. 3. It defines five dependence variables, each with different sets of parameters. After the declarations, the template file contains the vertices' specifications, which correspond to nodes in the Markov model. In the example, we define two vertices: r-type-inst and i-type-inst, each generating a 32-bit input stimulus. Below the bit structure definition, each of the fields is assigned a value from dependence variables. For instance, vertex r-type-inst binds field f to dependence variable r-funcs, while field f of the i-type-inst vertex is bound to variable i-funcs.

C. Dependence Variables

Dependence variables, such as those declared at the top of the template file in Fig. 3, provide a concise mechanism to specify the generation of values from: 1) a list of constants; 2) a uniform random distribution; 3) a randomly generated locality set; or 4) some combination of the previous three constructs. The variables are used to pass information between the template's vertices and fields and create complex interactions, such as locality and dependence between generated inputs. All of the variables have global scope and can be accessed from anywhere in a template file. The functional block used to generate a value for a dependence variable is illustrated in Fig. 4. The underlined labels in the figure represent the five declaration parameters of a variable.

1) **probCache** is the probability that a read() to a dependence variable will retrieve the value from the locality cache.
2) **cacheSize** is the size of the locality cache, which contains the most recently generated values that can be reused to simulate locality and dependence.

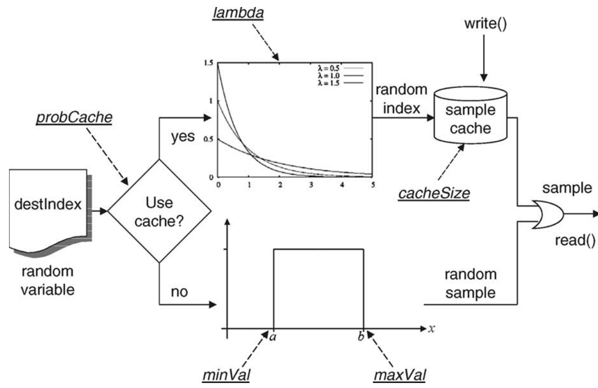

Fig. 4. Dependence variable functional block. A write() access to the variable inserts a value in the cache. On a read(), the parameter probCache is used to select whether the value should be generated randomly or retrieved from the cache. In the latter case, the lambda parameter affects the distribution of the selection from the cache. In the former, the distribution is uniform between minVal and maxVal.

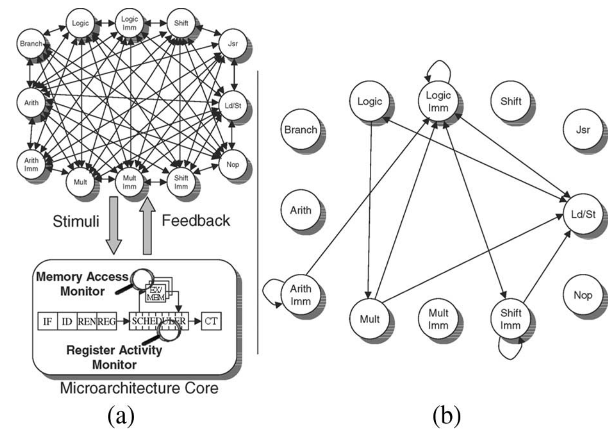

Fig. 5. Impact of activity monitors. A simulation using two activity monitors observing register file and memory interface transforms a Markov model started as a clique in (a) to the one generating dense register and memory writes in (b). The diagram in (b) has extremely low probability associated with all edges but the ones indicated.

3) **lambda** represents the length of the locality window, and it corresponds to the rate parameter to an exponential distribution which selects the cache indexes: The larger the value of lambda, the greater the probability that the selection is skewed toward recent element inserted; the smaller this value, the more uniform the distribution of selection among all of the cached elements.
4) **minVal and maxVal** are the bounds over which the uniform random samples are produced when the cache is not used.

Some of the parameters can be omitted when declaring the variables, in which case, a default value is used. For example, minVal is omitted when declaring variable i-funcs in Fig. 3; hence, the value of that parameter defaults to zero. When the variable is read(), the returned value is either taken from the sample cache with probability probCache, or produced by the random value generator with probability 1-probCache. Thus, by setting the parameter probCache to zero, we can create a perfect random generator, as it is the case for variable randIndex in Fig. 3. On the other hand, we can completely disable the random generator and select values exclusively from a predefined list similar to the case of variable i-funcs.

When a value is taken from the sample cache, lambda is used as the parameter of an exponential distribution function to generate the index of the cache entry storing the value to return. Note that high lambda values correspond to a high probability of generating low indexes, and thus retrieving more recent data. When the value of lambda is small, cache reads are almost uniformly distributed among all entries. Thus, it is possible to control the level of locality in the sample cache by varying lambda. In our context, this mechanism can be used to control the degree of register dependence between the generated instructions. For instance, in the example in Fig. 3, there is a dependence between field d of r-type-inst and field s of i-type-inst through destIndex. Moreover, the probability distribution of the returned values allows to efficiently control the strength of the dependence between instructions which operate on the same register and, therefore, the "stress" on the control and forwarding logic of such architectures. The constructs of sample cache and parametric fields proved to be sufficient for representing any instruction in the ISAs that we evaluated, due to the natural breakdown of the instruction formats into fields which have a fixed number of legal values and fields which can assume any value within a range. This is especially true of any RISC ISAs, because of their structured approach to instruction formats. For this family of ISAs, a large number of instructions can be compactly represented with a simple template file, using just a positional field notation, hence minimizing setup time and verification effort.

V. Activity Monitors and Feedback

This section presents the concept of activity monitors and describe how they are used to close the feedback loop. We also show how switching activity observed in the DUT can be related to transitions in the Markov model, and how multiple verification goals can be pursued simultaneously using this concept. Finally, we present an approach for the automatic extraction of additional relevant activity signals which allow to increase the quality of the tests generated.

A. Activity Monitors

In designing StressTest, we made an assumption that many bugs arise from complex interactions between instructions, which create high activity at the interface between design units and control blocks. Thus, we are more likely to find bugs by generating patterns that cause a high level of activity in those signals, rather than by simulating random input sequences. The activity monitors are responsible for identifying interesting input sequences which occurred by chance, and for reinforcing the appropriate transition edges, so that those sequences may occur more often. Fig. 5(a) illustrates the approach on 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. In particular, they monitor a set of "pressure points," that is,

user-specified relevant nodes within the design. A monitor for a single bit pressure point simply tracks its switching activity by counting the number of transitions occurred at the node due to the past stimulus. A monitor for a vector signal computes the sum of the single bit transitions and scales it by the vector's width. We found it useful to have the possibility to control a monitor through an enabled signal. For instance, in our experiments, the activity monitor for the memory interface would only be enabled if a memory transaction was executed during a cycle. Our activity monitors let us accurately recreate a range of real-world typical "stresses," such as high physical register file pressure, recurring dependent instructions in the register renamer, or high cache miss rates.

One challenging aspect of implementing the activity monitors was in maintaining the association between a particular Markov model transition and the resulting activity event. The challenge exists because input stimuli are generated many clock cycles before the corresponding activity can be observed. To maintain this binding, we use a small cache where we store a map of recently generated instructions and transitions leading to them. When sampling an observed activity at a pressure point, we also track the opcode of the instruction in the corresponding pipeline stage. The opcode is used to match an $\langle \text{instruction}, \text{transition} \rangle$ pair from the small cache. Once a transition is retrieved, the proper probability adjustments are applied. If more than one match is found, all the matches are adjusted. We found that multiple matches occur with less than 3% frequency, making the inaccuracy negligible.

Fig. 5 shows an example with two activity monitors and their impact on a Markov model. In the experiment, a Markov model for an Alpha ISA is used to generate stimuli for a testbed microarchitecture. Two activity monitors are engaged: a memory access monitor, which encourages frequent memory accesses, and a register file monitor, to push for the generation of tests with many accesses to the register file using diverse data values. In the initial Markov model [Fig. 5(a)], each transition is equally probable. However, due to the adjustment dictated by the activity monitors, after 8000 cycles of operation, a few transitions have much higher probability compared to the others. Fig. 5(b) shows the few high probability transitions that are left. As shown, the Markov model quickly morphs into a graph which generates many memory accesses due to the edges pointing toward the load/store generation node. Moreover, the use of instructions with immediate operands increases the range of values written to registers (immediate fields have random values), thereby reinforcing the monitor on the register file. Shift and multiply operations contribute to this monitor, too, by generating high variations in the computed output values. Finally, branches and jumps are less frequent since they cause little excitement in the register file.

The flexible construct of activity monitors enables the StressTest to aim for multiple verification goals simultaneously. For instance, the example above targets both registers and memory activity. When targeting multiple goals as in the example, StressTest computes a "score" (see Section IV-A) which is a weighted sum of the activity reported by the individual monitors. While this approach is often fruitful in discovering complex bugs due to unforseen components interactions (as we experienced in our experiments), it could theoretically lead to inadequate exercising of individual coverage targets. If this latter scenario occurred, StressTest could be easily adopted to focus on one verification goal at a time by using dynamic weights in computing the activity monitors' score. This flexibility enables the application of StressTest in a wide range of contexts. For instance, if a cycle-accurate description of the DUT is available, activity monitors can be used to target performance bugs. In this case, a bug can be found by observing the difference in performance of the golden model and DUT. Other examples include generating frequent collisions among packets routing through a network switch and checking correctness of the switch operation at high utilization points, or "stressing" a pipeline recovery mechanism with frequent mispredicted branches. In general, the "pressure points" to monitor during simulation should be selected based on the role they play in the operation and state of the DUT. For example, in our experimental evaluation, we selected three pressure points located at the register file interface, memory access control logic, and program counter control logic. The three corresponding activity monitors would tune the system to generate a wide variety of tests.

B. *Depth-Driven Activity Monitors*

We found experimentally that a small number of key activity points are sometime insufficient in directing the verification process toward interesting scenarios. The reason lies in the coarse "scores" collected when the system is run with few monitors showing bipolar behavior (very low or very high activity). The result is that, often, the model would "saturate" and produce very similar stimuli corresponding to the highest scores possible without exploring the surrounding areas of relatively high activity. To correct these situations, we expand the pool of user-selected pressure points with additional circuit nodes selected automatically by StressTest. The additional pressure points are nodes that directly influence the activity of the user-selected pressure points. For instance, if a pressure point is produced in the circuits's netlist as the logic AND of two other signals, then StressTest would add activity monitors for the two signals as well. It is obvious that generating high activity for the two additional signals would enforce StressTest's ability to stress the user-selected pressure point.

Another context where this technique is useful is an assertion-based verification methodology. Selecting the output of an assertion, or a checker, as a pressure point, would not provide any relevant feedback to StressTest's stimulus generator. In fact, the output of the checker would only transition once at the end of the simulation when a design error is uncovered. In this scenario, the use of additional pressure points in the logic cone of influence of the assertion is critical for the StressTest to be effective. In general, increasing the number of sampling points produces activity scores with fine-grain resolution, which reach smoothly the goal of interest. The additional pressure points are identified by the StressTest by analyzing either the circuit structure or the behavioral register transfer level (RTL) description of the DUT, which ever is available. We call "depth-1 signals" those signals which are direct inputs to the block whose output

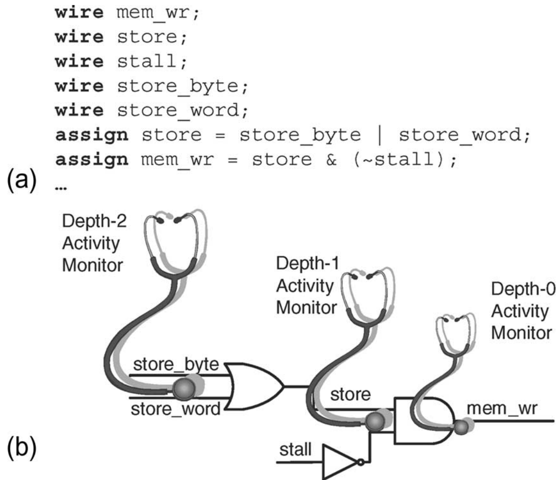

Fig. 6. Depth-driven activity monitors. StressTest complements the set of user-selected pressure points with additional nodes to refine the granularity of the measured activity scores. In the example, signal mem_wr is selected by the user; signal store and stall are identified as a depth-1 signals, based on the assign statement in the RTL description. Similarly, store_byte and store_word are depth-2 signals.

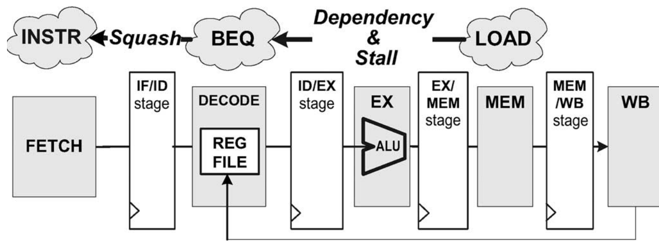

Fig. 7. Pipeline snapshot for case study Schematic of the five-stage pipeline used for the case study. The bubbles show the instructions exposing the design error: A branch instruction dependent on a load is stalled in Decode stage, and when the forwarded value shows that the branch is taken the instruction in Fetch should be squashed.

correspond to a user-selected pressure point. Here, a block could be a simple logic gate, a logic/arithmetic expression, or a control statement, depending on which DUT description is available. By expanding this construction recursively, "depth-2 signals" are signals in the cone of influence of depth-1 signals, etc. Activity monitors connected to these StressTest-selected pressure points have a lower impact in the activity score (see Section IV-A). More precisely, their weight is inversely proportional to their depth. Fig. 6 shows an example of how these additional pressure points are identified and connected to activity monitors. In the example, signal mem_wr is a user-selected pressure point, two additional monitors are created by analyzing the RTL description of the DUT.

VI. Case Study

In this section, we present a simple case study of how StressTest is used to detect an error related to a specific instruction dependence. The design under consideration is a five-stage MIPS pipeline similar to the one described in [11], whose simplified structure is reported in Fig. 7. For this design, branches are resolved in the decode stage and always predicted as not taken. When a branch instruction follows a load, it is possible to have a race condition if the load's destination matches the branch's condition register. In this case, the correct execution of both instructions requires to stall the pipeline for one cycle after load, so that the proper value can be read from the memory before it is tested for the branch. When the condition is finally evaluated, the branch is in the decode stage and the instruction at the fetch stage might need to be squashed if the branch was mispredicted (i.e., it is taken). A schematic of the specific situation is presented in Fig. 7. Note that both squashing and stalling should occur in the same clock cycle. However, due to an implementation error, the priority of the control signals (stall and squash) was incorrectly encoded and a stall would prevent the existence of a simultaneous squash. This encoding error allows the instruction following the branch to be executed causing an error. Given the very specific setup required to expose the problem, it would be unlikely to generate a test for it in a purely random verification environment. In fact, we would need to: 1) generate the proper sequence of instructions; 2) have a dependence between load and branch; and 3) have the branch be taken. A best case probability of all these conditions is computed as follows.

1) If we were to group instructions in four classes—loads, branches, noops, and all others—then the probability of generating the required sequence is: $p_{\text{seq}} = 1/4 * 1/4 * 3/4$ (note that the last instruction can be anything but a *noop*). Any other grouping would lead to an even lower probability for the desired sequence.
2) The probability to generate a dependence between the load and the branch is $1/N_{\text{regs}}$. The architecture in [11] has 32 registers; hence, $p_{\text{depend}} = 1/32$.
3) The probability of the branch being taken requires the two branch operands to have the same value. If the comparison is between two distinct registers, then this probability is $(1/2)^{32}$, which is negligible (each register stores 32-bit values). However, if the register operands of the instruction are the same, then the branch is always taken. Now, the probability of generating a branch where the operands are the same register is $p_{\text{taken}} = 1/32$.

By putting all together, the probability of generating the desired snippet of assembly program is

$$P = p_{\text{seq}} * p_{\text{depend}} * p_{\text{taken}} = 4.57763672 * 10^{-5}.$$

Hence, we can calculate the average number of cycles of random simulation before the sequence is generated as the expected value of a geometric distribution

$$\text{Cycles}_{\text{avg}} = \frac{1-P}{P} = 21844.$$

On the other hand, in StressTest, information passing between load and branch instructions can be setup by proper dependence variables. For example, if lambda $= 2$, and cacheSize $= 5$, the average distance between dependent instructions is 1.2, which means that virtually every pair of instructions depend on each other, exactly what we need in this case. Even if we offset that by setting probCache $= 0.5$, hence there is 50% probability of not exploiting locality, the probability of having a dependence between a load and a branch is still significantly higher than 1/32. Similarly, if the same dependence variable is used for both

operands of the branch, the probability of them being equal is much higher than 1/32. Finally, since the Markov model stores paths leading to favorable activities, by using a memory interface activity monitor and a branch logic one, the probability of having a load/branch/!noop sequence is much higher than that of a random approach. We found experimentally that, with the setup described, the StressTest was capable to expose the design error after only 126 simulation cycles using a range of initial condition seeds. In addition, note that in using this setup, the StressTest can generate longer dependence intervals and, thus, discover further potential bugs in the control logic. To conclude, the setup presented was simplified for presentation purposes; a full design scenario would penalize random simulator even further, while StressTest could still use all of its tuning features to narrow down on critical scenarios.

VII. Experimental Results

In this section, we first introduce our experimental evaluation framework and the test designs we used for the analysis. We then provide insights on a range of aspects in StressTest: the impact of the various parameters involved in dependence variables and a study of the convergence of the initial stimulus generator to a stable Markov model. Section VII-D evaluates the performance of our proposed technique against an open-loop random instruction generator, comparing both coverage of bugs and number of simulated instructions required to expose those bugs. Finally, the last part of this section reports on the impact of depth-driven activity monitors on the speed of convergence of the system on a range of design errors.

A. Experimental Testbeds

To evaluate the performance of StressTest, we conducted a series of simulations on two processor core designs described in Verilog RTL. The two systems have both five pipeline stages. The first is based on the MIPS-Lite ISA, and branches are resolved in the ID stage. The second design runs an Alpha ISA with branches resolved in the EX stage and has two-cycle store instructions. We also built single-cycle golden models (that is, functional descriptions) for both systems to evaluate the correctness of the designs. The DUT and the golden model were connected to independent data memories and interfaced to StressTest through the instruction bus. We connected the two implementations, DUT and golden model, through a small Verilog testbench interface. We also implemented the stimulus generator (template file parser and Markov model data structure) and the activity monitors (analyzing the activity on the pressure points) in a C++ program. Sampling of a switching activity at the pressure points was accomplished through Vera's binding constructs [8], which work as a glue between the Verilog description and StressTest's C implementation. We selected a range of pressure points to connect to the activity monitors, including key register file interface signals, memory system interface, branch/jump resolution logic, and pipeline stalling and flushing logic. The selection was driven by the wide variety of potential design errors that we were hoping to uncover in the testbed designs. To target more specific bugs,

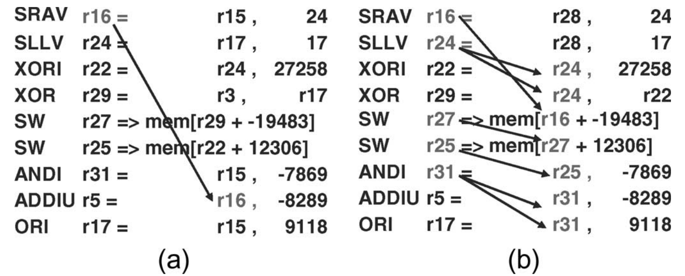

Fig. 8. Results of two sample runs with different values of cacheSize and lambda. The arrows represent dependences among instructions in a sequence. Note how larger values of lambda generate shorter dependence intervals. A larger cache size allows for more complex dependence patterns. a) CacheSize $= 20$, lambda $= 0.1$. b) CacheSize $= 10$, lambda $= 5.0$.

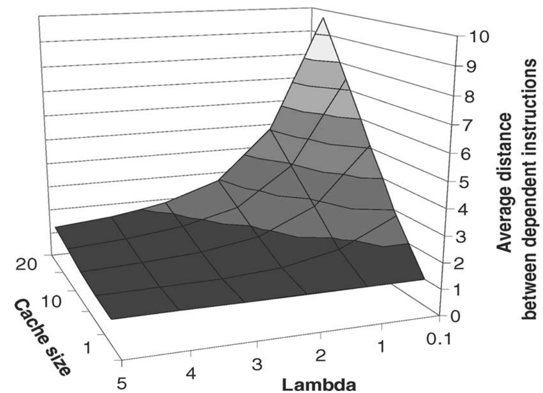

Fig. 9. Average distance between dependent instructions. Higher values of cacheSize and lower values of lambda result in larger intervals between dependences.

users would choose activity monitors at very similar points. The activity feedback from the DUT was observed using Vera and then passed to C functions for adjusting the Markov model.

B. Dependence Variables: Parametric Evaluation

The first set of experiments evaluates the impact of the parameters involved in a dependence variable. To this end, we set up an experiment on our Alpha pipeline testbench: The template file includes only one dependence variable, which is used to create a dependence between the source and destination operands in sequences of instructions. We fixed probCache $= 1$, and we used a range of cache sizes and lambda values for the dependence variable. For each simulation produced with this setup, we recorded the average distance between dependent instructions, that is the average interval between the insertion of a value into the locality cache and the retrieval of that value. In addition, we disabled activity monitors for this experiment, so that the sequence of instruction generated would be unchanged across different runs, as long as we started with the same random seed. Portions of sequences generated in two of such runs are shown in Fig. 8. It can be noted that the run, shown in part (b), has many more dependent instructions very closely spaced. This is due to the smaller value of cacheSize and larger value of lambda. The average dependence distance is plotted as a function of lambda and cacheSize in Fig. 9. The results of this analysis allowed us to better select the configuration of dependence variables for the experiments presented in the

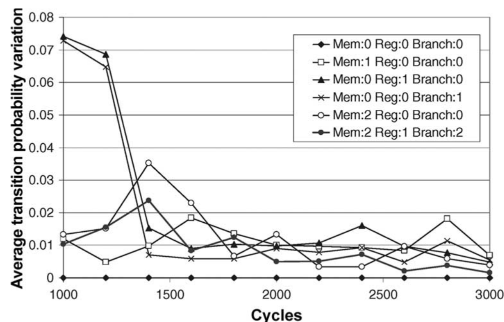

Fig. 10. Impact of activity monitor's weights on the Markov model stability. The diagram shows that the Markov model consistently converges to a stable form over time (simulation cycles). Each trend line has been generated using different weights for three distinct monitors: Memory accesses (Mem), register file (Reg), and branching (Branch). A value zero in the legend means that the corresponding monitor is suppressed. Note that the Markov model stabilizes in all cases and that higher weights result in faster stabilization.

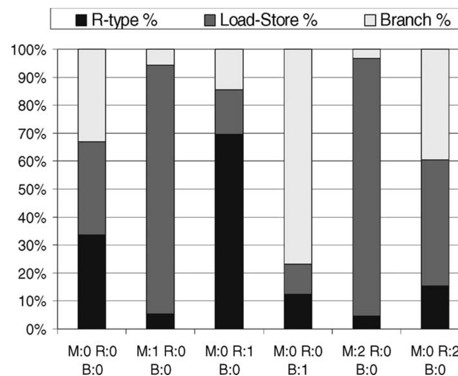

Fig. 11. Impact of the monitor's weight on stimuli generated. The graph reports the distribution of the stimuli among three types of instructions: Register, memory, and branch. The experiment is set up with three activity monitors, a memory (M), register access (R), and branching activity (B). Each bar considers a distinct setup with different weights associated with the monitors. The analysis shows that a higher weight results in a larger fraction of the stimuli generating activity for it.

following sections. Specifically, knowing that our testbed designs are five-stage pipelines with 32 registers, we were only interested in generating dependences between pairs that were four instructions apart or less. A lambda value of two provides a majority of dependence intervals within our range of interest. Also, we set the locality cache size to 20, so that we could have a broad set of values available. Larger values of lambda are more suitable for longer pipelines.

C. Stability of the Activity Monitors

The objective of our second set of experiments is to evaluate the change of the Markov model over time during simulation. More specifically, we study if the model converges to a stable shape (that is, if the probabilities associated with the transitions stabilize) in the long run, or if it keeps oscillating. In addition, we evaluate the impact of simultaneous multiple activity monitors on the stability of the model. We generate stimuli for a RISC pipeline for 3000 cycles and observe the probabilities marking the edges of the Markov model every 200 cycles. After the simulation, we calculate how much, on average, the values associated with the transition edges changed during each 200-cycle interval. The Markov model used for our experiments includes only three vertices, each representing a different class of instructions, namely, memory operations (loads and stores), register-to-register instructions, and branches. We also used three distinct activity monitors, practically corresponding to each of these instruction types: a memory accesses monitor, a register file one, and a program counter control monitor. We ran several simulations varying the weights assigned to each monitor, effectively changing the impact of the activity observed. The results of this experiment are shown in Figs. 10 and 11. Fig. 10 shows that the model stabilizes for all the scenarios (a range of different weights associated with the activity moni tors) that we explored within 2000 cycles (probability variation is below 0.02 on average). This consistent stabilization of the Markov model has both advantages and drawbacks. The advantage is that a particular "path" is set up within 2000 simulation cycles, and from then on, sequences of instructions with similar structure are generated repeatedly. However, since the various operands of the instructions are generated via dependence variables, different dependence combinations are still explored, which is crucial for thorough testing. On the other hand, the Markov model would not transition to another stable point any longer, because the activity feedback forces it to remain stable. Therefore, after a while, the simulation starts losing diversity. As expected, higher weights cause the Markov model to converge faster, since activity scores are higher.

During this set of experiments, we also monitored the number of each type of instruction generated during a simulation run. Ideally, a high weight on a memory access monitor should lead to generating more load/store instructions. A high-weight register file monitor should provoke many register-type instructions, and the PC control logic monitor should generate many branch instructions. Fig. 11 shows the relative distribution of each type of instruction generated during simulation. When the weights are all zero (as in the leftmost bar), the model does not receive any feedback and instructions are roughly 1/3 of each type. It is also evident from Fig. 11 that the relative frequency of occurrence of each instruction is related to the weight associated with a particular activity monitor reinforcing the corresponding type of instruction. Note also that when the weight of the memory interface activity monitor is increased from one to two, the number of loads and stores increases, because of the stronger reinforcement connected to the corresponding Markov model edges.

D. StressTest Coverage Density and Performance

We now examine the coverage density and performance of StressTest compared to other approaches based on random simulation. The techniques we compare are: plain random simulation, simple constrained generation, Open-Loop, and StressTest. For each approach evaluated, we quantify the effort (in simulation cycles) required to expose bugs and analyze how many bugs the technique is capable to expose. The testbenches used for these experiments are the DLX and Alpha processors described in Section VII-A. The set of tests for the DLX design consists of 30 distinct versions of the DLX core, each containing a different bug. Bugs vary from simple

(such as incorrect operation for a given arithmetic opcode) to complex ones, involving forwarding logic and interactions through memory. Ten of the simplest bugs for DLX were taken from a testsuite that is part of an advanced hardware verification course at the University of Michigan, while the others were handcrafted for the experiment. For each of the buggy variants and techniques we considered, we performed 25 runs using distinct random seeds, and we evaluated the average effort and standard deviation. In each run we simulated for a maximum of 75 000 cycles, or less, if the bug was exposed sooner. We selected the number of distinct seeds and the length of the simulation runs to ensure stabilization of the Markov model and to permit each experiment to complete in less than 3 h. The set of tests for the Alpha design used ten buggy variants including moderate and very complex bugs, mostly very specific corner cases in the pipeline's forwarding logic. The four techniques we compared in the analysis are the following.

1) **Random** utilizes only a static evenly distributed Markov model of the ISA for the instruction generation and does not collect feedback from the DUT. It also uses several dependence variables, but without caching. Random represents a capable open-loop testing solution.
2) **Simple** relies on a feedback-adjusted Markov model, based on activity feedback from the DUT. However, it still does not use caching for the dependence variables.
3) **Open-Loop** is an open-loop setup which uses dependence variables with caching, but does not have activity feedback, so the Markov model has evenly distributed edges across the entire simulation.
4) **StressTest** is the full-fledged implementation using both a feedback-adjusted Markov model and dependence variables with caching. Variables are used to transfer destination register indexes to source register fields and to share arithmetic immediate values and memory and branch offsets among the instructions in the same test.

The tests' setup uses an initial Markov model with 7 and 11 vertices for DLX and Alpha, respectively. Each vertex represents a particular class of instructions. No single vertex included multiple instructions, since the ISAs did not impose any sequencing constraints on the input, i.e., branches did not require a noop instruction to follow. The dependence variables were set with cacheSize = 20 and lambda = 2.0, values which we derived from the analysis described in Section VII-B. We also set probCache = 0.66, allowing random values to be freshly generated in 30% of the uses of dependence variables, to introduce variation in the generated test programs. Variables that store static information, such as opcodes, had probCache = 1.0 and lambda = 0.1, so to obtain a uniform random distribution among the values in the cache (which was initialized with all the legal opcodes in the ISA). We used three activity monitors located at the memory control logic, the register-file interface, and the PC control logic. We used an analysis similar to that of Section VII-C to select a set of weights for the monitors, and finalized them to 2, 1, and 2, respectively. In setting up the experiments for each of the four random testing techniques, we exposed bugs by running in lockstep with a golden model and flagging any discrepancy in the committed results by comparing register file, program counter, and memory writes.

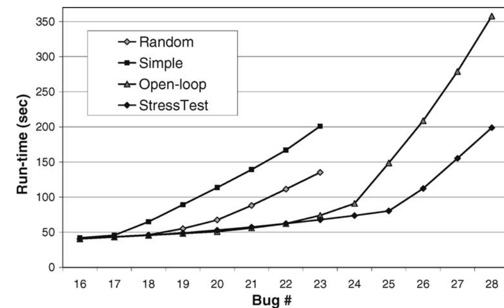

Fig. 12. Effort versus bugs covered for the DLX processor. The diagram compares four random testing-based techniques in their effectiveness at uncovering design errors. StressTest and Open-loop can expose more bugs than Simple and Random, and StressTest can do faster than all other techniques.

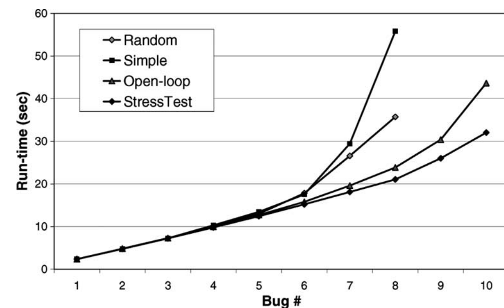

Fig. 13. Effort versus bugs covered for the Alpha processor. The diagram compares four random testing-based techniques in their effectiveness at uncovering design errors. StressTest can expose more bugs in less time than all other solutions.

Figs. 12 and 13 show the results of the four random test generation approaches applied to the DLX and Alpha processor pipelines. For each of them, the graph illustrates the cumulative effort (in total runtime) versus the total number of bugs detected. To distinguish between easy-to-find bugs from harder ones, we have sorted them in ascending order of total number of instructions required to locate a bug. As a result, the bugs on the left part of the graph were easier to locate than the bugs on the right. When a technique was incapable of finding some of the bugs (for instance, random), the curve stops short. In addition, we are only showing the 15 hardest bugs in Fig. 12, since the performances were virtually indistinguishable for simpler bugs. As shown in Fig. 12, StressTest and Open-Loop achieve better coverage than Random for the DLX processor, detecting five extra bugs. StressTest is also far more efficient than Open-Loop at detecting all the bugs, requiring approximately half the time. Interestingly, Simple appears to be the worst approach, despite of its use of activity feedback. We believe that this is due to the inability of generating interesting correlations between instructions due to the lack of caching in dependence variables.

The experiment led to exposing three hidden bugs in the forwarding logic of the DLX pipeline, which initially was assumed to be correct. Although this design was a subject of verification projects for several years, these bugs were unknown, until we exposed them with the StressTest.

1) Forwarding through Reg.0. Register 0 in a DLX architecture should always retain the value zero. When forwarding through this register, the second instruction should always receive a zero value, no matter what the

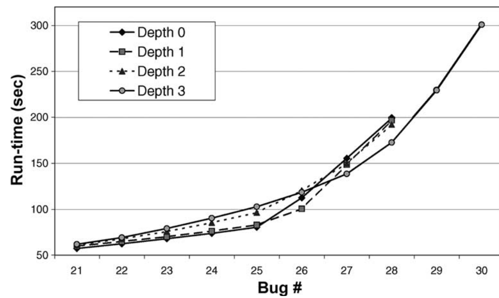

Fig. 14. Effort versus bugs covered for StressTest with depth-driven activity monitors (DLX processor). The diagram compares a range of depths for the activity monitors and shows that depth-driven activity monitors allow StressTest to find complex bugs faster.

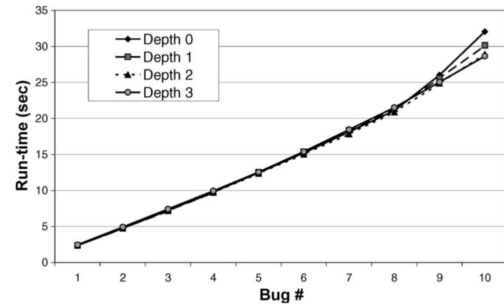

Fig. 15. Effort versus bugs covered for StressTest with depth-driven activity monitors (Alpha processor). The diagram compares a range of depths for the activity monitors and shows that depth-driven activity monitors allow StressTest to find complex bugs faster.

first instruction computed. However, the original design allowed bypassing, and the value forwarded to the second instruction would be the result of the first instruction's operation. Most experiments with Random and Simple could not locate this bug because the generated operand fields had a very small chance of creating the dependences through register 0 necessary to expose the bug.

2) Stalling logic. Because of the timing between memory accesses and branch resolutions, it is possible in DLX to create a scenario where the pipeline is stalled due to memory contention while its front-end is being flushed due to a mispredicted branch. We found that our initial pipeline design ignored pipeline flushes in this context, allowing the instruction following the mispredicted branch to proceed to execution. An analysis of this situation is discussed in the case study of Sections VI.
3) Forwarding through unused register. Some MIPS instructions use only one source register, which is ignored in the execution and memory stages. However, when a dependence through this register existed, the forwarding logic would still trigger and forward the bogus value to a following instruction. Dependence variables played a key role in discovering this bug.

The features which led to find these bugs were: 1) simulating the DUT in lockstep with the golden model to check correctness; 2) generating instructions using templates; and 3) passing information between instructions using dependence variables. Without these techniques, the bugs would have been extremely hard to find and would have required significant user effort in directing the test toward them. Fig. 13 shows the results of the same experiment on ten variants of the Alpha pipeline, each including a different bug. Again, we computed average cumulative effort and sorted the bugs from easiest to hardest. The analysis confirms once again that the advanced features of StressTest are critical in exposing the more complex bugs.

E. Depth-Driven Activity Monitors

We also investigated the impact of the depth-driven activity monitors on the performance of StressTest. We used the same 30 variants of DLX cores and ten of Alpha pipelines and compared the full-fledged StressTest used for the previous section against StressTest with depth-driven activity monitors of depths 1, 2, and 3. The experiment setup was the same as the one in Section VII-D. Figs. 14 and 15 show the results for DLX and Alpha pipelines, respectively. Note that in both cases, deeper monitors perform better for harder bugs, but perform slightly worse for medium-difficulty bugs. For example, in Fig. 14, the performance of StressTest up to bug 25 is better than other approaches, but it is worse for all other harder bugs (26 and up). The reason lies in the amount of additional information which is accrued by the depth-driven monitors: Easier bugs are usually exposed by simpler scenarios which can be reached by watching only a handful of critical signals, and additional information is just "distracting" the system. However, in harder bug scenarios, any available additional information is helpful in narrowing the bug scenario. Finally, Table I reports the standard deviation from the mean number of simulation cycles (to locate bugs) that was encountered for each of the experiments. We performed the comparison using the last bug index which was exposed by all the techniques. Table I shows that StressTest finds bugs more consistently than Random and other techniques which rely mostly on chance. Additionally, the table illustrates that the deviation is reduced with increasing activity monitors' depths, implying that observing more signals leads the StressTest to narrow design errors more precisely and consistently.

TABLE I
DEVIATION FROM THE MEAN (CYCLES)

	DLX	Alpha
	max up to bug 24	max up to bug 8
Random	38323	7876
Open-loop	19788	4211
StressTest	5043	1089
	max up to bug 28	max up to bug 10
StressTest	27200	5228
Depth_1	27198	3516
Depth_2	24289	3091
Depth_2	12873	2295

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented a novel approach to constraint random validation. The approach, implemented in a tool called StressTest, is a closed-loop technique based on a Markov model which generates instruction sequences based on templates. These templates are designed by verification engineers to resemble legal directed tests. Our template language is

particularly expressive, in that, it supports the generation of a wide range of input types with varied dependence and locality characteristics and can be used in the verification of processor cores or other digital circuits. Moreover, the verification engineer needs to identify the key activity signals in the design, i.e., signals that are indicators of "stressful" activity or are suspected to be indicators of performance or design bugs. A closed-loop feedback engine adjusts the Markov model continuously during simulation based on the activity observed at the activity points, to produce effective and efficient tests. For more effective feedback, StressTest automatically extracts the cone of logic influencing the user-selected activity points and monitors them as well, smoothly guiding the simulation toward interesting scenarios. Experimental evaluation found that the StressTest is capable of finding more bugs in fewer simulation cycles than open-loop random simulation or less-sophisticated closed-loop test generation techniques.

Looking ahead, we are extending this paper in a number of directions. An important issue that we are investigating is in overcoming the limitation of Markov models of generating input based only on short past history. We plan to create mechanisms to store more information about the stimuli generated in the past inside StressTest by adding path-dependent transition policies or grouping sequences of favorable stimuli into additional vertices in the Markov model. We also plan to introduce a meaningful measure of coverage to StressTest that would depend on the quality and variance of activities seen throughout the simulation. This would allow us to compare the tool to similar software developed in industry, for example Genesys-Pro and X-Gen. In addition, we are expanding the language to support more effective specification of data values, along with the instructions that access them. Finally, we are exploring the application of the StressTest infrastructure to other domains. In particular, we are working to deploy techniques similar to StressTest to communication protocols and hardware with multiple parallel interfaces.

REFERENCES

[1] A. Adir *et al.*, "Genesys-pro: Innovations in test program generation for functional processor verification," *IEEE Des. Test Comput.*, vol. 21, no. 2, pp. 84–93, Mar./Apr. 2004.

[2] A. Allan, D. Edenfeld, J. William, H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, "2001 technology roadmap for semiconductors," *Computer*, vol. 35, no. 2, pp. 42–53, Jan. 2002.

[3] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov, "Industrial experience with test generation languages for processor verification," in *Proc. DAC*, Jun. 2004, pp. 36–40.

[4] B. Bentley, "Validating the Intel Pentium 4 microprocessor," in *Proc. DAC*, 2001, pp. 224–228.

[5] B. Bentley and R. Gray, "Validating the Intel Pentium 4 processor," *Intel Technol. J.*, vol. 5, no. 1, pp. 1–8, 2001.

[6] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed. Norwell, MA: Kluwer, 2003.

[7] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proc. DAC*, Jun. 2003, pp. 286–291.

[8] F. I. Haque, K. A. Khan, and J. Michelson, *The Art of Verification With Vera*. Fremont, California: Verification Central, 2001.

[9] Y. Hollander, M. Morley, and A. Noy, "The e language: A fresh separation of concerns," in *Proc. Technol. Object-Oriented Languages and Syst.*, Mar. 2001, vol. TOOLS-38, pp. 41–50.

[10] J. M. Ludden *et al.*, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM J. Res. Develop.*, vol. 46, no. 1, pp. 53–76, Jan. 2002.

[11] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1997.

[12] R. Emek *et al.*, "X-gen: A random test-case generator for systems and socs," in *Proc. Int. Workshop HLDVT*, 2002, pp. 145–150.

[13] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin, "System-level validation of the Intel Pentium M processor," *Intel Technol. J.*, vol. 7, no. 2, pp. 38–43, May 2003.

[14] G. Spirakis, "Opportunities and challenges in building silicon products in 65 nm and beyond," in *Proc. DATE*, 2004, pp. 2–3.

[15] 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," in *Proc. ICCD*, 2001, pp. 82–88.

[16] 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 *Proc. DAC*, 1998, pp. 638–644.

[17] J. Yuan, C. Pixley, and A. Aziz, *Constraint-Based Verification*. New York: Springer-Verlag, 2006.

Ilya Wagner (S'06) received the B.S. and M.S. degrees in computer engineering from University of Michigan, Ann Arbor, in 2004 and 2006, respectively, where he is currently working toward the Ph.D. degree.

He is currently working with the Advanced Computer Architecture Lab of the Computer Science and Engineering Department, University of Michigan. He is also the President of the Michigan Mars Rover Project, a volunteer student group with mission to develop a manned vehicle for planetary exploration. His research interests include hardware verification and hardware reliability.

Valeria Bertacco (M'95) received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1998 and 2003, respectively.

She is an Assistant Professor of electrical engineering and computer science (EECS) with the University of Michigan, Ann Arbor. Her research interests are in the areas of formal and semiformal design verification with emphasis on full design validation and digital system reliability. She joined the faculty with Michigan after being with Synopsys for four years, as a Lead Developer of Vera and Magellan, two popular verification tools. She has been leading the effort for the development of the verification section in the International Technology Roadmap for Semiconductors report since 2004.

Dr. Bertacco has served in several program committees, including ICCAD, FMCAD, and HLDVT, and as a chair of the verification committee for DATE.

Todd Austin (M'88) received the Ph.D. degree in computer science from University of Wisconsin, Madison, in 1996.

He is an Associate Professor of electrical engineering and computer science with University of Michigan, Ann Arbor. His research interests include computer architecture, compilers, computer system verification, and performance analysis tools and techniques. Prior to joining academia, he was a Senior Computer Architect with Intel's Microcomputer Research Labs, a product-oriented research laboratory in Hillsboro, OR.