Depth-Driven Verification of Simultaneous Interfaces

Ilya Wagner   Valeria Bertacco   Todd Austin
Advanced Computer Architecture Lab
University of Michigan
Ann-Arbor, MI 48109
e-mail: {iwagner, valeria, austin}@umich.edu

Abstract— The verification of modern computing systems has grown to dominate the cost of system design, often with limited success as designs continue to be released with latent bugs. This trend is accelerated with the advent of highly integrated system-on-a-chip (SoC) designs, which feature multiple complex subcomponents connected by simultaneously active interfaces.

In this paper, we introduce a closed-loop feedback technique targeting the verification of multiple components connected by parallel interfaces. We utilize an environment with hierarchical Markov models, where top-level submodels specify overarching simulation goals of the system, while lower-level submodels specify the detailed component-level input generation. Test accuracy is improved through the use of depth-driven random test generation. The approach allows users to specify correctness properties and key activity nodes in the design to be exercises. We examine three non-trivial designs, two microprocessors and a chip-multiprocessor router switch, and we demonstrate that our technique finds many more bugs than constrained-random test generation technique and reduces the simulation effort in half, compared to previous Markov-model based solutions.

I. Introduction

Systems-on-a-chip (SoC) are becoming predominant in application-specific domains, such as portable media devices, network processors, and so on. The development of an SoC entails the integration of multiple heterogeneous integrated circuit (IC) blocks, with a mix of components developed in-house and acquired through third-party IP, producing extremely complex IC systems in a relative short time. Unfortunately, the verification of SoC has become a bottleneck in the design process, as it commonly absorbs 70% of the design costs. In fact, the International Technology Roadmap for Semiconductors (ITRS) has determined that the most important productivity challenge to be overcome, in order to maintain the current growth trends in SoC development, is precisely verification [4].

Two aspects of SoC make their functional verification particularly challenging: first, the integration density offered by state-of-the-art fabrication technologies is impressive: a typical SOC could contain hundreds of millions of transistors. The result is a large complex stateful design that does not lend itself to existing verification techniques, either because of the small fraction of state space that simulation-based approaches can explore, or because the sheer complexity is beyond the grasp of current formal and semi-formal technology. Second, SOCs feature many heterogeneous autonomous components connected by multiple simultaneously active interfaces, often abiding to complex communication protocols. The inability of exploring all the possible interactions among these components further underscores the challenge of SoC verification.

A variety of techniques have been explored to assist the designer in locating design bugs on systems such as this. Simulation-based random test generation is a long-standing approach used to locate design errors [5, 9, 10, 8, 12]. However, due to the huge state spaces of even simple devices, it is impossible to achieve sufficient confidence in the correctness of a design by just using random generation techniques. Formal and semi-formal solutions can help by providing powerful mechanisms to achieve high coverage in verification, but can only be deployed on the smallest components of a design. The need for solutions that can address both the high-coverage and design size requirements has been recently voiced by Intel, which predicted that by 2007 designs will deploy 100M transistors and require 2000 person-years of effort for verification [13]. The work presented here attempts a step in this direction by proposing a novel solution which targets both high-coverage and scalability in the verification of complex systems with multiple parallel interfaces.

A. Contributions of This Work

In this paper, we introduce a novel closed-loop test generation technique targeting the construction of high-quality tests for large complex designs with multiple, simultaneously active interfaces. Our solution generates constrained, automatically-biased tests by modeling the interfaces' communication protocols through abstract Markov models, and by sampling the system's reactions through signal sensors placed at relevant internal nodes of the designs. Specifically, the work makes two novel contributions compared to previous efforts in this field:

1. The complications of modeling the input protocols and the interactions among multiple simultaneous interfaces are mitigated through the use of a **hierarchical modeling environment**. The environment dictates the specification of a high-level Markov model (through the use of simple templates) concerned with describing the overarching testing goals. Subsequently, this high-level model spawns off lower-level Markov models – threads – that specify component-specific input generation.
2. To address complexity challenges, our constrained random test generation technique introduces a **depth-driven test quality evaluation** technique. In this context, the

user hints to the system the critical regions to cover through the specification of key signals within the design. The test generator then tunes its focus on input stimuli that exercise the logic closer to these key signals (where ”closeness” is defined in terms of logic depth from the key signal, hence the name).

In addition, we implement our proposed test generation infrastructure in a tool called **IQTest** (Interface Quality Tester), and demonstrate that it is capable of finding more bugs than constrained random testing techniques, and with less simulation effort than previous Markov-model based solutions.

The remainder of this paper is organized as follows. Section II highlights previous related work. Section III overviews the high-level architecture of *IQTest*, while section IV provides details on the novel aspects of the work. Rationale and details of the implementation of *IQTest* are also discussed. Section V presents experiments that compare IQTest to previous solutions and naive random testing. We examine three non-trivial designs, two microprocessors and a chip-multiprocessor network router switch. Finally, Section VI gives conclusions and suggests future directions.

II. Prior Work

Random test generation has long been a focus of industry and the academic research community [6]. The key aspects of systems that differentiate the verification solutions are whether or not simulation is pure random or directed, how constraints on the random tests are specified, and the mechanics of the underlying random test generation engine.

Recent advances in random test generation have focused on generating feedback on previous tests to influence the generation of future tests. Tools such 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 GUIs 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 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 [3]. Most tools employ a coverage-directed generation process, but use sophisticated techniques for representing relationships between coverage and input generation through Markov-models (as in this work) [11, 14] or with Bayesian networks and computer learning [7].

StressTest was recently proposed as a technique to implement closed-loop feedback directed random testing [14]. The tool is based on an abstract representation of the input model, specified using a template based specification language. In addition, users specify “activity monitors” which represent signals in the design that correspond to coverage concerns, so that the underlying simulation engine directs simulation towards the excitation of the activity monitors. Our solution, implemented in *IQTest*, borrows the specification language and Markov-model test generation from [14]. However, we make a number of significant advances over that work. In particular, we extend the template language to include support for hierarchical specifications, which simplifies the modeling of simultaneous interfaces. Moreover, we provide a highly responsive and accurate link between activity monitors and model response, through the use of a depth-driven activity analyzer.

III. An Overview of IQTest

Figure 1 illustrates the high-level architecture of IQTest’s random testing infrastructure. There are four primary inputs to the system (colored in gray): 1) the design under test (DUT), 2) a known-correct golden model, 3) the input template specification, and 4) the activity signals. Based on the input template, IQTest generates an input sequence which is fed to both the DUT and the golden model. The outputs of the two models are compared and, if any discrepancy is noted, a bug is detected.

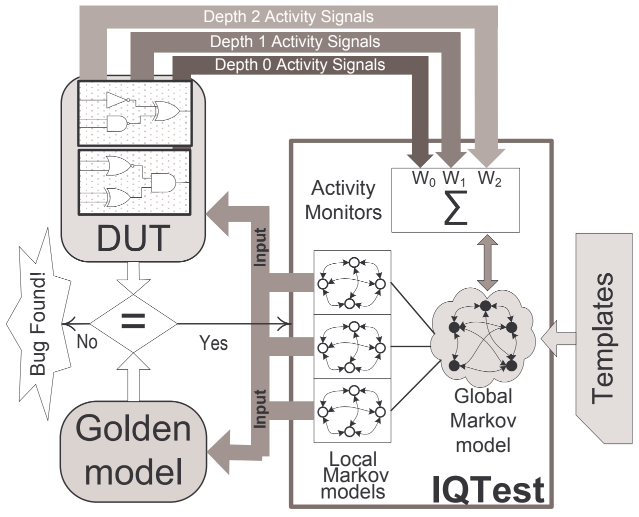


Fig. 1. IQTest Structure. A template describing the legal DUT stimuli is used by IQTest to generate input sequences. In addition, an activity analyzer closes the feedback loop by scoring the quality of the stimuli. Bugs are detected by validation against a golden model.

Markov models and Templates. The Markov models are in charge of the stimuli generation in IQTest. In this context, a Markov model is a graph where each vertex describes a legal input sequence for the design. For instance, if the input interface was a processor’s instruction bus, a vertex could describe a single instruction, or a program segment. Or, if the input interface was a bus protocol, than a vertex could describe a possible transaction. The vertices in the graph are connected by edges labeled with the probability of performing a graph transition through them: at the beginning of the test generation, all the outgoing edges from a vertex have equal probability. Probabilities are adjusted over time so as to bias the stimuli towards the interesting ”activity”, as specified by the ”activity signals”.

Moreover, the input sequence described by each vertex incorporates random aspects, based on the template specification. As discussed in more detail in Section IV, templates are text files providing a model for the input sequences at each vertex of the Markov graph. For more details on the basics of templates and the related Markov model construction see [14]. To address the specific needs of SoC verification, we devised a novel technique to structure the Markov models hierarchically, so that a system with multiple, simultaneously active interfaces could be stimulated by IQTest through all its input channels asynchronously and in parallel.

Activity Signals and Analyzer. The activity signals are internal nodes of a design under test selected by a user because they are representative of critical activity in a component. Examples of such signals are collision indicators between different ports of a network switch (which can be used to check the correctness of the switch at high utilization), or a branch misprediction signal (used to verify a pipeline recovery mechanism). The activity analyzer gathers the switching activity of these signals and steers the test generation by indicating to IQTest which transactions in the Markov models generate the highest activity, and thus are most relevant in exercising critical circuitry. During simulation, the edges of the Markov models are adjusted continuously so that high-activity transactions are associated to higher probabilities.

Activity signals may be any design's internal node deemed relevant by the user, or checkers (that is, properties that we are trying to falsify). In the latter case, IQTest focuses on activating the output signal of the checker: the detection of switching activity at such a node corresponds to having triggered the checker. Note that, particularly in the case of checkers, the activity observed at the node would be non-existent during the whole simulation, until when the checker is fired, thus reducing IQTest to a mere constrained random test generation with no adaptive biasing. To solve this problem, IQTest introduces a novel depth-driven activity feedback solution. The approach works as follows: from the signals selected as activity sensors, we derive an additional set of auxiliary signals, whose values closely affect the value of the activity sensors. The activity monitor then estimates the change to apply to the Markov models' edges based on a weighted sum of the activity in the extended set of signals. The weights are heavier for the activity sensors and lighter for the auxiliary signals. We call the approach "depth-driven" because the weights are inversely proportional to the logical depth of a given auxiliary signal from the sensor. We find that this approach reaches significantly more bugs than simple constrained random simulation.

IV. Architectural Insights

In this section we focus on the two main contributions of this paper, namely:

- The ability to specify the communication protocols of the DUT hierarchically. The architecture of *IQTest* and the template language we defined facilitates the simulation and verification of simultaneous interfaces.
- The activity analyzer, our technique to produce an accurate analysis of the design response to a stimulus transaction. At each simulation step we consider the signal transitions at multiple nodes in the design and evaluate the quality of the last transaction by weighing this data based on the circuit depth from the critical node

A. Hierarchical Specification of the Stimulus Generator

For the specification of the input protocol at each interface of the design, we deploy a hierarchical Markov model that generates valid legal input sequences based on the template specifications. In previous work proposing random input generation based on a Markov model [14], the model was used to partition the set of inputs of a microprocessor core and generate instruction sequences based on activity feedback obtained from the design. However, it is often the case that a design has multiple parallel input interfaces, unlike pipelines which can be viewed as having only one stimuli entry point. A good example of such a design is a network switch or a crossbar that has multiple ports. In such situations, it is critical to be able to generate stimuli at each individual port that are time- and data-independent. Therefore, it is often more desirable to generate multiple input streams in parallel and observe the interactions between them. Since often the hardest bugs in an SoC design are found when multiple input requests are competing for the same hardware resources, we deemed crucial to allow the designer to create sequences of input stimuli where the traffic at each interface is independent, so that it becomes easier to produce relevant input sequences. Finally, a methodology that allows for hierarchical specification of the stimuli leads to a description of the input transactions that is simpler and easier to understand.

To cope with this problem we devised a new approach that employs a multi-level Markov model. Each of the parallel inputs of the circuit is assigned to an individual Markov model for generating valid input stimuli according to the interface specifications. However, some information between these models can be shared to increase the competition for the resources and intensify the pressure on the design. To allow this information passing between the models, we use a global variable space that is accessible from any model. Note that the models can still generate valid input sequences independently of their peers through local variables.

In addition to the individual Markov models assigned to input ports of the design, a global model is used to encode possible scenarios of simultaneous stimuli generation. For example, the model can determine the number of ports of the design activated simultaneously and values of the controlling inputs to the design. The objective of the global Markov model is to coordinate local models assigned to individual design ports. Note that individual models supply sequences of stimuli with dependencies between them, while the global model orchestrates them to exert simultaneous pressure on the design. The activity feedback from individual inputs of the design in this framework is used to reinforce transitions in local models. Moreover, combined activity measures from different points in the circuit are used for adjusting edges in the global Markov model.

```
/ Global variable space /
global {
  dest(probCache=0.7,cacheSize=8,lambda=.5,
                                    minVal=1,maxVal=15);
  srcW(probCache=1,cacheSize=20,lambda=3)={'b0000, 'b1111};
}
 / Global Markov model/
none : TopModel(global) {
  rand-send-one(probCache=1)={'b1000,'b0100,'b0010,'b0001};
  command[3:0] : { portD, portC, portB, portA };
  vertex(send_one_pkt) {  command = 'bCCCC;
                          field(C)=$rand-send-one.read(); }
  vertex(send_all) { command = 'b1111; } };
 / Local Markov models descriptions/
switchPort {
  using global::dest;
  using global::srcW;
  vertex (message) {  input='bDDDDSSSS;
                      field(D)=$dest.read();
                      field(S)=$srcW.read();
                      $dest.write(field(D)); } };
burstingPort {
  Bsrc  (probCache=0,minVal=0,maxVal=15);
  vertex (message){  input='b0100SSSS;
                     field(S)=$Bsrc.read();
                     $Bsrc.write(field(S)); } };
/ Local Markov models binding /
dut.port_a : switchPort(portA) ;
dut.port_b : burstingPort(portB) ;
dut.port_c : switchPort(portC) ;
dut.port_d : switchPort(portD) ;
```

Fig. 2. Example Template file. Shown are a global Markov model along with two local models bound to each port of a network switch.

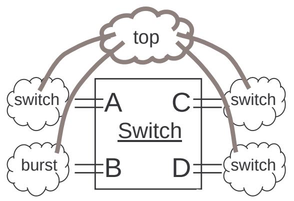


Fig. 3. Switch and hierarchical Markov model for the template example.

An example of a hierarchical template file shown in Figure 2 indicates how hierarchical Markov models can be used to model different interfaces of a DUT. See [14] for details of Markov model and variable specification. In the case shown in the example, the DUT is a network switch with four simultaneous interfaces *port_a* to *port_d*. The two input-generating models *switchPort* and *burstingPort* produce different kinds of data packets, while the global top level Markov model creates different scenarios and orchestrates the work of the packet generators. The messages described here consist of a packet with 8 bits of data.

The top portion of the template contains the global variable space with two random variables, *dest* and *srcW*. They are visible to any of the local models that declare using them via a *using* clause, as shown in model *switchPort*. The global Markov model either runs a scenario where input is sent to only one port or where it is sent to all ports simultaneously, by signaling to the low level models with command bits. Model *switchPort* produces packets with source and destination fields generated by accessing the global variables, while model *burstingPort* produces all messages to destination 0100.

B. Depth-Driven Activity Monitoring

One important assumption that was made in prior work related to Markov-model based testing is that a handful of key activity signals is sufficient to guide the simulation towards areas of interest and expose hard-to-find bugs. However, we found this approach to be somewhat coarse. In other words, the scores reported by the activity monitors manifest bipolar behavior. Therefore the system was likely, after just a handful of cycles, to strongly reinforce input sequences leading to high activities and almost eradicate the possibility of generating inputs that lead to low activity rates.

In our improved test generation methodology, a different approach was taken. We have created a tool that traverses the hierarchy of the design and extracts additional signals that influence the behavior of the selected activity points. Preference was given to control signals that were relatively easy to identify from a register-transition level design description. The signals that directly influence the primary activity points were assigned depth one, the signals that influence them were assigned depth two, and so on. The activity analyzer incorporates these secondary signals using a weight proportional to depth of the signal exercised (Figure 4-left). Note that the selection is done automatically and does not require any additional user effort.

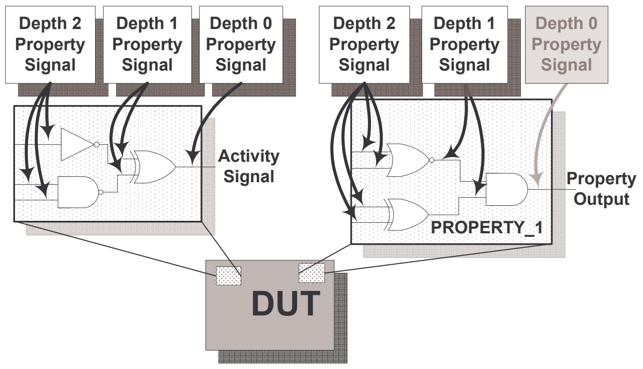


Fig. 4. Depth-driven activity monitoring selects auxiliary signals to monitor based on their logic depth from the property output or activity signal

In the context of a verification methodology that attempts to falsify properties embedded in a design, we deploy a specialized adapted technique for the evaluation of the activity analyzers. Often, a design includes several checker modules that track signals vital to system operation, and are triggered when a property is violated. Typically, these checkers are derived from the design specification or embedded by the developer. The analysis of the outputs of these checkers is the critical aspect of validating a design's correctness, however, since these properties are never asserted until the corresponding bug is exposed, the guidance provided to the input generator is poor. The input signals to the property expression, on the other hand, change frequently and can be selected automatically from the description of the checker (Figure 4-right). As it is shown in our results, this depth-directed activity analysis technique is able to achieve greater bug coverage with less effort, compared to approaches that observe only a handful of key points.

V. Experimental Evaluation

In this section, we introduce our experimental evaluation framework and the designs we tested. We then compare the performance of our proposed technique against an open-loop random instruction generator and a recently proposed Markov-model based test generator, comparing both coverage of bugs and number of simulated instructions required to expose them.

A. Experimental Framework

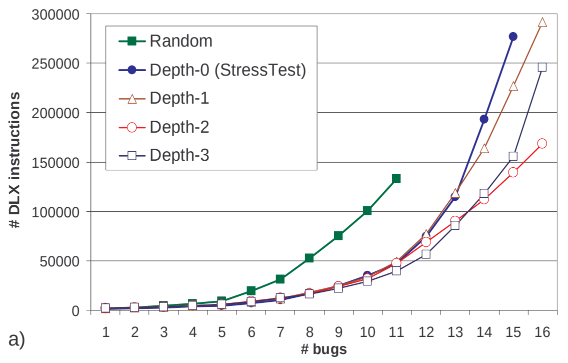
Evaluation of the random test generators was performed on three designs: two microprocessors and a chip-multiprocessor router switch. The two microprocessor designs that we tested were the DLX pipeline (MIPS-Lite ISA) and a DEC Alpha processor core. For both designs we wrote a behavioral golden model, against which the correctness of the design was compared. Also we created 30 cores for the DLX and 10 for Alpha, each containing one bug. The simplest bugs included incorrect opcode interpretation and erroneous ALU operations, while the hardest ones could only be exposed through complex interactions between instructions. The templates for these experiments were derived directly from ISA of the processors, with one vertex corresponding to one instruction type. The amount of effort needed to create the templates was under 8 man-hours. The chip-multiprocessor router switch was included because it features simultaneous parallel interfaces. The switch was initially designed for testing performance and traffic patterns of different routing algorithms. In all experiments, we used a version of the switch utilizing an adaptive cut-through minimal-path routing algorithm for two-dimensional mesh networks. The design consists of five input ports with three virtual channels each, five output ports, and crossbar logic.

For the switch experiments, the generator was a hierarchical Markov model with the top level model specifying the number of packets to send simultaneously. The top model also specified the state of the back pressure signals in the network surrounding the switch. The local Markov models were used to generate valid packets that were likely to have similar destinations, thereby exerting high pressure on the switch. The amount of effort to write templates for both global and local Markov models for the switch was around 16 man-hours. To guide the test generation during switch verification, we utilized checkers written in Verilog. We derived ten distinct checkers from the high level description of the switch routing algorithm and buffer functionality. Each property module had a single-bit output, which depended only on particular signals in the design. The *Depth-0* experiment only monitored the output bits of the properties, while IQTest monitored the inputs to the properties, *i.e. Depth-1* signals, as well.

Incidentally, we were able to find three actual design bugs during verification of the switch: one in the buffer control logic and two in the routing logic of the crossbar. All of these bugs were hard corner cases of the switch's behavior, for example, in one bug several internal counters were incorrectly handled during a buffer-overflow situation, however from the error could be seen several tens of cycles later.

B. Results and Analysis

Since our verification mechanism uses a random generator in its kernel, each buggy design in both experiments was run 25 times with different random seeds and afterwards we calculated average effort and coverage. In the first test, the maximum allowed time to search for a bug was limited to 75000 cycles for both DLX and Alpha. For performance evaluation each bug was checked by an open-loop constrained random generator (*Random*). We also compared IQTest against *StressTest*, which is based on a closed-loop Markov model structure but it observes only a handful of crucial control signals in the pipeline. *IQTest* itself was implemented with different depths of the observed signals, labeled correspondingly (*Depth-1*, *Depth-2*, and *Depth-3*).



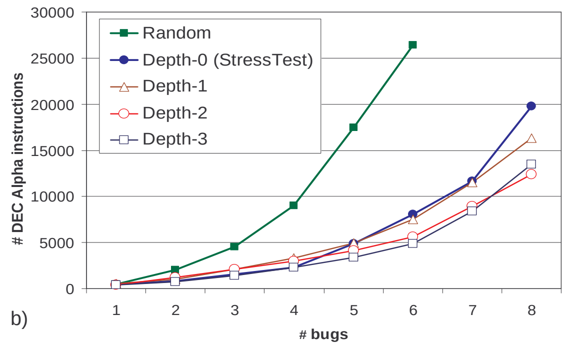


Fig. 5. Effort vs. Bug coverage for DLX (a) and Alpha (b) cores

Figure 5 shows the effort in terms of number of instructions produced, of these three techniques vs. the number of bugs discovered. Here we present the results for the 16 hard bugs in DLX and 8 hard bugs in the Alpha design. Since the easy bugs were found by all techniques in just several hundred instructions, for these bugs there is no clear differences between the approaches. As can be seen in both graphs, the curve for *Random* stops short of the rightmost edge, that is, the hardest bugs, while both *StressTest* and *IQTest* were able to find all the bugs in the Alpha pipeline. Note that *StressTest* was unable to find the most complex bug in the DLX pipeline. In general, we note that a deeper activity analysis is better for harder bugs.

The results of the switch verification experiment are presented in Figure 6. The effort in this case is measured in number of packet transactions required to find a bug. Again only the hard to find bugs are reported, since all three systems performed equally well in discovering easy bugs. The three bugs

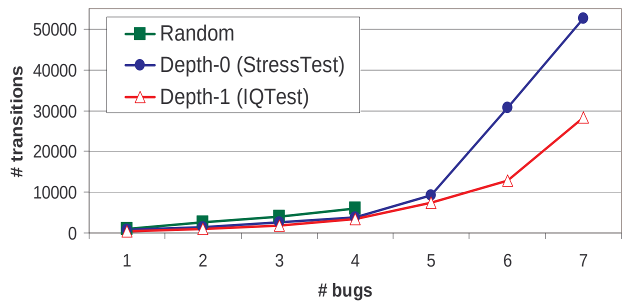


Fig. 6. Effort vs. Bug coverage for the switch design

on the far right of the graph are the three original bugs found during the verification of the switch. Once again, the *Random* configuration was unable to discover the last three bugs and requires significant effort for covering easier bugs. That explains why the bugs were not discovered during the design of the switch where only random testing was used. As can be seen in the graph, the approach of monitoring just the output bits of the property modules performs relatively well for intermediate bugs. We believe that is partially because our approach enabled information sharing between local Markov models. Moreover, this system explores more distinct input sequences than *Random*. This is because observing property outputs produces low activity scores that stimulate negative reinforcement in the Markov model, thus continuously steering the generation away from past sequences. The third approach produces positive and negative reinforcement and therefore significantly reduce the effort needed to uncover the bugs.

Finally, in Table I, we compare the wall-clock performance of *IQTest* vs *Random* (we run on a 1 GHz UltraSPARC IIIi machines with 1GB of RAM). Since *Random* discovers only a fraction of the bugs that IQTest finds, we used the best achievements of *Random* as a reference. As shown in the table, IQTest performs always better than Random. And after Random runs out of steam, IQTest keeps finding more complex bugs.

VI. Conclusions

In this paper, we introduced a random simulation technique targeting SoC devices, with stateful components connected by parallel interfaces. The approach is based on a Markov-model driven random simulation, with two novel enhancements: To produce effective random vectors for multiple components with parallel interfaces, we utilized a hierarchical modeling environment where ”global” models specify overarching simulation goals of the system, while the ”local” models specify the specific component-level input generation. To address the statefulness of SoC devices, we utilize a depth-driven random simulation search engine. The search engine allows users to specify correctness properties and key activity points in the design that are of particular concern. A closed-loop feedback system then tailors the hierarchical Markov-models into one that stresses the points of concern in the design.

We found that the hierarchical specification and depth-driven random test generation are quite effective in reducing the amount of effort required to expose bugs in complex designs, both in terms of the amount of human effort to craft the verification scripts and the amount of simulation required to reach the bugs. We examined three non-trivial designs: two microprocessors and a chip-multiprocessor router switch. We demonstrate that hierarchical specification of input models yields a compact precise representation. Additionally, we showed that depth-driven random simulation finds more bugs more quickly than simple constrained random simulation. Moreover, we found that depth-driven simulation cuts the simulation effort in half, compared to a recently published Markov-model based verification technique.

TABLE I - Time comparison of random and IQTest

	DLX	Alpha	Switch
	up to bug 25	up to bug 8	up to bug 7
Random	135.5s	35.7s	39.1s
Depth-0	N/A	N/A	32.1s
Depth-1	70.1s	21.3s	31.6s
Depth-2	75.7s	21.0s	N/A
Depth-3	79.1s	21.5s	N/A

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. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[4] A. Allan et al. 2001 technology roadmap for semiconductors. *IEEE Computer*, pages 42–53, Jan. 2002.

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

[8] I.Silas et al. System-level validation of the Intel Pentium M processor. *Intel Technology Journal*, 07:38–43, May 2003.

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

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

[11] S. Tasiran et al. 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 et al. 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.

[13] G. Spirakis. Opportunities and challenges in building silicon products in 65nm and beyond. In *Design and Test in Europe (DATE-2004)*, 2004.

[14] I. Wagner, V. Bertacco, and T. Austin. Stresstest: An automatic approach to test generation via activity monitors. In *DAC, Proceedings of Design Automation Conference*, 2005.