H-0215 November 23, 2003
Computer Science

IBM Research Report

Quality Improvement Methods for System-Level Stimuli Generation

Roy Emek, Itai Jaeger, Yoav Katz, Yehuda Naveh
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Quality Improvement Methods for System-level Stimuli Generation

ABSTRACT

Functional verification is known to be a major part of the hardware design effort. System verification is aimed at validating the integration of several previously verified components. As such, it deals with complex designs, and invariably suffers from tight schedules and scarce resources. We present a set of methods, collectively known as *testing knowledge*, aimed at increasing the quality of automatically generated system-level test-cases. These methods are based on the commonality of some basic architectural concepts. Testing knowledge reduces the time and effort required to achieve high coverage of the verified design. Towards the end of the paper, we compare the coverage achieved with and without the usage of testing knowledge, and describe related hardware bugs.

1. INTRODUCTION

Functional verification is widely acknowledged as the bottleneck in the hardware design cycle [3]. Simulation is the main functional verification vehicle for large and complex designs, and therefore stimuli generation plays a central role in this field.

During the last few years, complex hardware designs have shifted from custom ASICs towards System on a Chip (SoC) based designs, which include ready made components (IP, cores). The verification of such systems requires new tools and methodologies that are up to the new challenges raised by the characteristics of systems and SoCs. At the heart of these challenges stands the requirement to verify the integration of several previously designed components in a relatively short time.

In this paper, we present a set of methods aimed to improve the quality of automatically generated test-cases for system-level verification.

Coverage is the prime measurement for the quality of a set of test-cases. Simply stated, the idea in coverage [10, 7] is to create, in a systematic fashion, a large and comprehensive list of tasks and check that each task is covered in the testing phase. Ultimately, higher coverage means higher chances of exposing a bug. Coverage can help monitor the quality of testing and direct test generators to create tests that cover areas that have not been adequately tested. We may measure what coverage is achieved by a given set of test-cases. We may also measure the additional coverage achieved by these test-cases, given that a previously used pool of test-cases already achieved some coverage of the verified design. Disregarding the issue of coverage, a simplistic measurement of the quality of a test-case bucket is the number bugs found in the verified design.

In this paper, we explore a set of generic methods useful for a variety of hardware designs. We claim that these methods increase the coverage for many typical coverage models, and aim at areas which are, because of their inherent complexity, bug prone. We use the term *Testing Knowledge* (TK) to refer to generic methods for increasing the quality of test-cases.

The idea of a testing knowledge was previously implemented in test-case generators oriented at the verification of processors [2, 6, 12, 1][1]. Testing knowledge for the verification of address translation mechanism was also presented. We are not aware, however, of a significant effort to develop a set of testing knowledge mechanisms for the system level.

General purpose verification environments, such as *Specman* [13], *Vera* [8], and *TestBuilder* [5] provide means for implementing testing knowledge in the form of non-mandatory (or 'soft') constraints. However, soft constraints alone do not contain any semantic knowledge of the verified design. In [4, 14], the authors deal with constructing a constraint solver for stimuli-generation. Such solvers provide random solutions, and at the same time support both hard (mandatory) and soft (non-mandatory) constraints. As such, they provide means for implementing testing knowledge once the users, the verification engineers, come up with ways for increasing stimuli quality. None of these papers, however, provide actual testing knowledge mechanisms.

The rest of this paper is structured as follows: Section 2 describes the concept of testing knowledge, provides motivation, and explains why the idea of testing knowledge is especially advantageous in system verification. Sections 3 through 6 describe a set of system oriented testing knowledge mechanisms. Section 7 provides two examples of hardware bugs found using the described TK mechanisms, and examines the coverage achieved by two comparable tools—one that uses TK, and one that does not. Finally, Section 8 concludes the paper.

2. MOTIVATION: TESTING KNOWLEDGE AND SYSTEM VERIFICATION

2.1 Testing Knowledge

The stimuli generation layer of a verification environment can be roughly partitioned into two (see Figure 1): knowledge about the

[1]In the first reference, the term 'testing knowledge' is used in a slightly different manner than here.

specification of the verified system, and knowledge about the way it should be verified. The former defines the set of *valid* tests that can be injected to the Design Under Verification (DUV). The latter, which implements the verification plan, is a mixture of two types of information: the first determines which tests will actually be generated. It is expressed through 'test-templates' of the form "100 transactions of type *A*, and 50 transactions of types *B*". The second type aims at increasing the quality of *all* the generated test-cases, and contains statements of the type "by default, the ratio between valid and erroneous transactions should be 4:1"—this is what we refer to as TK.

Given a test-template as input, a random test-case generator produces a set of different test-cases, all satisfying the requirements specified in the test-template. In most cases, generating high-quality, complex scenarios, require complex test templates. Developing these test-templates is a costly effort: this creates an incentive to move some of the burden from the test-templates to the testing-knowledge. By doing so, we allow all the generated tests, and not just a small fraction of them, to benefit from the intelligence implemented in a complex test-template.

It should be noted that the separation between test-templates and testing knowledge is not absolute. For testing knowledge to be used to its full extent, the user—the test-template writer—should have some control over what types of testing knowledge are used in conjunction with a specific test-template. Following the previous examples for testing knowledge and for a test-template, a user may combine both aspects, for example, by constructing the test-template "100 transactions of type *A*, and 50 transactions of types *B*, with a ratio of 3:2 between valid and erroneous transactions".

- Knowledge required for stimuli generation
  - System specification: defines test validity
  - Verification plan implementation
    - Test templates
    - Testing knowledge

Figure 1: High-level classification of knowledge in stimuli generation

In principal, random stimuli generation environments produce test-cases which are uniformly selected from the domain of valid test-cases. Testing knowledge biases the random selection of test-cases towards areas that have higher chances of increasing coverage or exposing hardware bugs (See Figure 2). This concept is therefore closely related to non-uniform distribution functions, and the ideas described in Sections 3 through 6 can be seen as instances of such functions.

2.2 System Verification

We define a 'system' as set of *components* connected using some sort of interconnect, capable of performing a given set of *transactions*. Components may include processors and other processing elements, caches, various types of memories, bridges, interrupt controllers, DMA engines, etc. In many cases, a system contains multiple instances of a certain type of component: for example, a system with symmetric multi-processing would contain several processors. Examples of transactions include Memory Mapped I/O (MMIO), Direct Memory Access (DMA), and simple `store` or `load` instructions from a processor to a memory.

System verification deals, in essence, with the validation of the integration of several previously verified components (cores, IP). Inherently, it deals with large designs: verification methodologies that are apt to the unit or component level are not necessarily suitable for the system level. A related factor that is also crucial to verification is the intricacy of the specification of the system. As the aim of the verification effort is to show that a design implements its specification, complex specifications require special attention and affect the verification process. Other than the size of the design and the complexity of the specification, the main challenge related to system verification is limited resources, and specifically short schedules. Verifying the entire system can often start only after all its components were brought to a certain level of stability, which, in many cases, leaves only a small time window for the system verification effort itself.

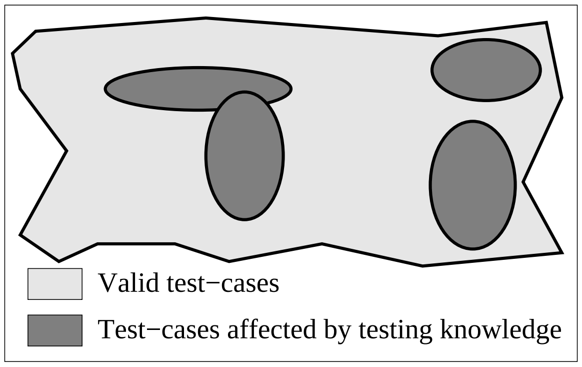

Figure 2: Testing knowledge as non-uniform distribution. Each dark area represents a class of 'interesting' cases, targeted by a testing knowledge mechanism.

The main direction proposed as a possible solution to these challenges is reuse. Checkers (e.g., assertions) written for lower level interfaces and components can be reused at the system level. The same is true for Bus Functional Models (BFMs): a BFM written for the verification of a PCI/X core, for example, can be reused to inject input to the system as a whole. Standard, or commonly used, interfaces call for verification IP, which is another form of reuse. For example, a BFM for an AMBA bus can be purchased from a third party company, and then used for the verification of multiple systems (even across different organizations).

Testing knowledge embodies another type of reuse. Here, we gain knowledge about prone-to-bug areas in one system, and reuse this knowledge for the verification of others. An even stronger form of reuse may be achieved if the stimuli generator itself allows the integration of general testing knowledge for a specific domain (e.g., systems or SoCs). TK mechanisms can then be configured differently for different systems. This allows the verification engineer to avoid implementing a piece of testing knowledge for each new design.

The ability to use general ideas of testing knowledge, and apply them to a wide variety of systems, is based on the fact that architectural concepts often appear in similar forms in multiple systems. For example, the concepts (and corresponding units) of address translation, caches, complex interconnect, and multiple instances of processing elements appear in many systems. Testing knowledge that aims at the verification of systems containing such units can be then used multiple times.

3. BASICS: WEIGHTED RANDOM CHOICE

The most basic form of testing knowledge, supported by all existing verification environments, is weighted random choice. A given property *P* of the stimuli can randomly accept a value from a domain (set) D_P of valid values. A uniform distribution would result in an equal probability of $\frac{1}{|D_P|}$ for all the values in D_P. In many cases, however, we would like to attach different probabilities to different values in the domain of P. For example, generating `read` transactions 80% of the time, and `write` transactions 20% of the time.

Weighted random choice is *unary*: it modifies the distribution of a single property of the stimuli, independently of other properties. Weight functions may vary from a simple two-value step function to, e.g., the Poisson distribution. Non uniform distribution functions can also apply to the relationships between multiple properties. For example, given a system whose interface contains two properties, a and b, we may require that the ratio between the cases of $a > b$ and $a \leq b$ would be 5:1.

It should be noted that distribution functions can be attached to *virtual* properties of the stimuli, as well as to *real* ones. According to this terminology, real properties are syntactically present on the interfaces of the DUV, while virtual properties are not. For an example of a virtual property that is important from a verification point-of-view, consider the processor instruction `load` $R_t \leftarrow [R_a + R_b]$. Here, the contents of the memory cell at the address given by the sum of registers R_a and R_b is transferred into register R_t. In this case, the indices of the three registers t, a, and b are real properties, while the address of the memory cell, the value $R_a + R_b$, is a virtual property.

In addition to the transactions that comprise the stimuli, weighted random choice can also apply to system initialization. Here, a set of registers, and potentially other resources, is randomly initialized. The initialization values are then used later in the test-case. A simple form of testing knowledge may bias the random selection of values for each register (or resource) separately. A more advanced form is to create dependencies between different registers (or register fields) of the same component, and an even more advanced form is to create dependencies between resources from different components. The precise nature of these dependencies is related to the function of the DUV.

4. RESOURCE CONTENTION

System-level bugs are often related to scenarios in which several consumers contend for the usage of a single resource. It is therefore beneficial to create a TK mechanism that causes multiple 'agents' in the system to access a single resource, preferably during a limited time interval. Note that the existence of several consumers is closely related to systems: it is derived from the existence of multiple components.

One way to cause contention for resources is though an *address collision* mechanism. In many systems the concept of a system-address is the main way to identify resources. In these systems, it is typically beneficial to cause multiple accesses to the same addresses. This is especially true in systems with a complex cache hierarchy: Completely random accesses would practically only cause cache misses, while address collisions would also cause cache hits, cast-outs, and potentially various cache related corner cases.

An address collision mechanism can be implemented using a queue that contains pairs of the form $(address, length)$ ($length$ is the number of bytes accesses by a transaction). Following every transaction that uses an address, we push a new pair into the queue, and potentially remove a pair if the queue is full. With some probability, the address and length properties of a new transaction are generated to form a collision with one of the pairs in the queue.

Consider an example with a queue of size 3. At some point of the generation process, the queue contains the pairs

$$[(0x1000, 0x10), (0x1100, 0x20), (0x1200, 0x30)]$$

At that point, we generate a new transaction, and randomly decide it will not be biased by the address collision mechanism. The address and length of the new transaction are selected to be $(1300, 0x8)$. Following this transaction, the state of the queue is

$$[(1300, 0x8), (0x1000, 0x10), (0x1100, 0x20)]$$

We then choose to generate a transaction that is affected by the address collision mechanism, for example $(10F8, 0x20)$. This transaction collides with the last pair in the queue.

The usage of a queue as a data structure for holding candidate pairs for collision serves two purposes. First, it allows for a performance efficient generation process. Saving *all* the previous accesses may result in a huge data structure, which in turn slows down the generation. Second, it aims at collisions not only on a certain resource (e.g., a cache line), but also in time.

The basic address collision mechanism described here can be expanded in various ways. In Section 6, we will show how this mechanism can be combined with topology and configuration based TK mechanisms. Another enhancement would be to provide more information to the algorithm that causes collisions for a newly generated transaction. For example, for a DUV that includes an L2 cache, we may aim for transactions that collide on the same L2 cache-line, and not necessarily on the exact same address. This can be further enhanced to collision mechanisms for various address-based resources in the system (e.g., cache congruence classes, translation pages, etc.).

An example for a different enhancement is the usage of an address collision mechanism for the `larx` and `stcx` instructions of the PowerPC [11] architecture. These two types of instructions provide a means for implementing semaphores and other synchronization mechanisms in an MP system: `larx` creates a reservation for a single processor on a certain location in memory, and `stcx` provides an atomic test-and-set operator. Most PowerPC implementations contains special purpose hardware for supporting these instructions, and specifically for supporting their combined usage. To improve the verification of this hardware, we may add a testing knowledge mechanism that increases the probability of collisions specifically between a `stcx` instruction and the `larx` instructions preceding it.

Finally, the address collision mechanism can be easily expanded to create contention on resources that are not identified by address and length. Here, the queue of $(address, length)$ pairs is replaced by a queue of other types of resource identifiers.

5. TESTING KNOWLEDGE FOR ADDRESS TRANSLATION

Most modern systems support several address spaces. First, complex processors typically distinguish between a 'virtual' address space, as seen by applications, and the 'real', or 'physical' address space, as seen by the operating system. The hardware provides mechanisms for translating between these two address spaces. Second, there are often address translation mechanisms between the system address space (as viewed on the system, or processor bus) and I/O address space (as seen, for example, on a PCI bus). Finally, high-speed interconnect technologies, such as InfiniBand [9], also support complex address translation mechanisms.

One type of testing knowledge for address translation is a natural follow-on to the resource contention mechanism described in Section 4. With large translation tables, the probability of randomly using the same translation entry twice within a single test is relatively low. However, it is beneficial to verify the system under scenarios that reuse the same entry multiple times. We would therefore like to bias the test towards such scenarios. This is especially important when the system, or the translating component, contains a cache for translation entries, usually known as 'translation look-aside buffer' (TLB).

A more complex scenario related to the TLB is translation entry invalidate. Here, the translation table is modified during the test, to validate that the relevant component continues to function correctly. Other types of testing knowledge for address translation are mentioned other papers.

A different type of testing knowledge related to address translation deals with the *placement* of accesses relative to virtual pages or segments. Most translation mechanisms partition the address space into a set of pages, or segments. Some architectures support a single page (segment) size, while others support multiple sizes. The placement testing knowledge is aimed to better stimulate address translation related hardware. With some probability, this mechanism places an access in a manner that would cause one of three placement events (see Figure 3):

- *Vicinity*: an access is placed near the beginning of a page (segment), or near its end.
- *Boundary*: an access is placed at the very beginning of a page (segment), or at the very end of it.
- *Crossing*: an access starts in a certain page (segment), and ends in another.

Each of these three events may activate a control-logic that relates to a certain corner-case in the DUV. For a specific example, see sub-section 7.3.

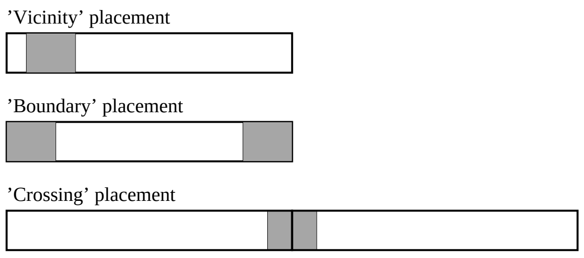

Figure 3: Placement testing knowledge

6. TOPOLOGY AND CONFIGURATION BASED TESTING KNOWLEDGE

The testing knowledge mechanisms presented in Sections 3 and 4 are suitable for systems, but some of them can also be used for the verification of a single unit or component. This section deals with mechanisms that are only applicable for systems containing multiple components, and in most cases multiple instances of a single component type.

Consider the system depicted in Figure 4. This system is comprised of four nodes, connected together with some form of interconnect. Each node contains two processing elements (e.g., a processor), and a memory. According to the specification of this system, each processing element may access the memory of its own node, as well as the the memory of any of the other nodes.

The first type of testing knowledge mechanism we describe in this section is based on the understanding of such topologies. From the point of view of the processing elements, accessing the local memory and the remote memory requires the same operation (e.g., a regular `load` or `store` instruction in the case of a processor), only to a different address space. However, from the point of view of the hardware, accesses to different memories stimulate different control mechanisms. Note, that the address space of each of the four memories is not necessarily fixed, and can depend, for example, on a configuration register. An example for a configuration based testing knowledge mechanism is therefore: *the ratio between accesses to local memories and accesses to remote memories should be 3:1.* In the general case, this kind of TK mechanism requires knowledge of the topology and the configuration of the verified system. Based on this knowledge, it can then bias between different values for properties of the stimuli.

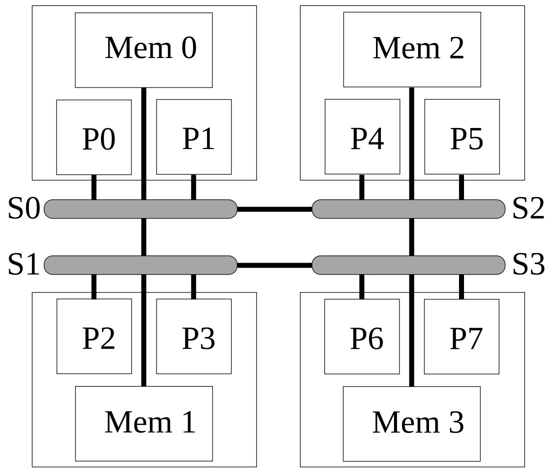

Figure 4: A sample system: four nodes, of two processing elements and a memory each, connected with some form of interconnect.

A second type of configuration based TK is *actor choice pattern* (ACP). The system shown in Figure 4 contains eight processing elements and four memories. Disregarding the previous testing knowledge mechanisms, the probability of generating a transaction between any pair of a processing element and a memory is $\frac{1}{4\times8} = \frac{1}{32}$. ACP aims at generating more 'interesting' patters for transactions that involve several components (actors). The mechanism creates a non-uniform distribution function, and uses this function when choosing actors for newly generated transactions. The distribution function for a transaction that involves *n* actors can be represented as an *n*-dimensional matrix. For example, the uniform distribution matrix for a processor - memory transaction for the system shown in Figure 4 is shown in Table 1 (a). An example for a non-uniform distribution matrix is shown in Table 1 (b).

Sparse matrices like the one shown in Table 1 (b) cause the traffic in a certain test-case to concentrate in some parts of the interconnect. The basic TK behind ACP is that for each test-case, a sparse probability matrix is created. This probability matrix can be completely random, or, it can take into account issues like the topology of the DUV, or its configuration. Over a large number of test-cases, we will then form different patterns of stress on different parts of the interconnect. The effectiveness of ACP is driven from the fact that it increases the quality of interconnect testing, with almost no knowledge of the structure of the system, and through an easy-to-use and easy-to-implement, mechanism.

Path aware testing knowledge goes one step further. For the system shown in Figure 4, it takes into account not only knowledge of

	P0	P1	P2	P3	P4	P5	P6	P7
Mem0	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$
Mem1	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$
Mem2	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$
Mem3	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$

(a) Uniform probability matrix

	P0	P1	P2	P3	P4	P5	P6	P7
Mem0			$\frac{1}{16}$	$\frac{1}{16}$				$\frac{1}{8}$
Mem1			$\frac{1}{16}$	$\frac{1}{16}$			$\frac{1}{8}$	
Mem2	$\frac{1}{16}$	$\frac{1}{16}$				$\frac{1}{8}$		
Mem3	$\frac{1}{16}$	$\frac{1}{16}$			$\frac{1}{8}$			

(b) Non-uniform probability matrix

Table 1: Probability matrices

which memory resides with which processing element in the same node, but also knowledge of the path between any two nodes (and any two components). For example, to stress the switch S_1 it may bias the transactions generated in a test-case towards ones that involve S_1. This stress can be viewed as another form of system-level resource contention.

In the general case, *path aware* testing knowledge creates test-cases in which large portion of the transactions go through a single component, or a set of components, in the interconnect between transactions initiators (e.g., processors) and targets (e.g., memories).

Lastly, knowledge about the topology and configuration of the system can be integrated into the resource contention testing knowledge mechanism presented in Section 4. To do that, we partition the set of consumers (e.g., processors) in the system into several groups. This partition can be completely random, it can be based on the topology of the system (e.g., processors from the same node would be in the same group), or it can be based on a mixture of randomness and knowledge of the topology. The queue of $(address, length)$ pairs described in Section 4 is expanded to include triplets of the form $(address, length, consumer)$. When a collision is generated, each consumer would collide only with other consumers from its own group.

7. COVERAGE AND SAMPLE HARDWARE BUGS

7.1 Coverage

Table 2 compares the coverage achieved by two comparable tools, used simultaneously for the verification of the same design—a high-end MP system—and implementing the same verification plan. The first line shows the number of simulation cycles consumed for tests generated by each of the two tools (normalized to 100, the real numbers are in the range of billions of cycles). Tool 1, which did not use any testing knowledge, generated about five times as many cycles as Tool 2, that did[2]. Because of the use of testing knowledge, the number of test-templates (shown in the second line) required to implement the verification plan was much smaller in Tool 2 than in Tool 1. The rest of the table shows the coverage achieved by the two tools on a set of coverage models for activities on various buses. The TK based tool reached better coverage on all of the coverage models. The results show the percentage of covered events as part of the complete cross product model. The percentages out of the architecturally legal events are not available. Coverage results for other models, not shown here, are similar.

[2]The names of the tools are removed for review anonymity.

Category	**Tool 2** With TK	**Tool 1** No TK
Simulation cycles (normalized)	100	479
No. of test-templates	737	7168
Coverage model 1	40.57%	37.10%
Coverage model 2	43.84%	26.88%
Coverage model 3	74.28%	68.30%
Coverage model 4	61.14%	59.17%

Table 2: Comparison between the coverage achieved by two tools: one using testing knowledge, and one does not.

We describe two hardware bugs found by test-cases that were significantly influenced by some of the testing knowledge mechanisms described above. In the two cases we describe below, the bug would not have been found, or would have been found later in the verification process, if it wasn't for the relevant testing knowledge mechanism.

7.2 Address Collision

This bug was found in a system containing several processors, as well as a complex I/O sub-system (Figure 5). External I/O devices can connect to the system through a PCI/X bus. These devices then communicate with the rest of the system through a series of bridges and controllers. To improve performance, one of the bridges in that series maintains a cache of areas in memory read by PCI/X devices. When a processor or another I/O device writes to a cached area, a 'kill cache-line' transaction is sent down from the controller residing between the bridge and the system bus.

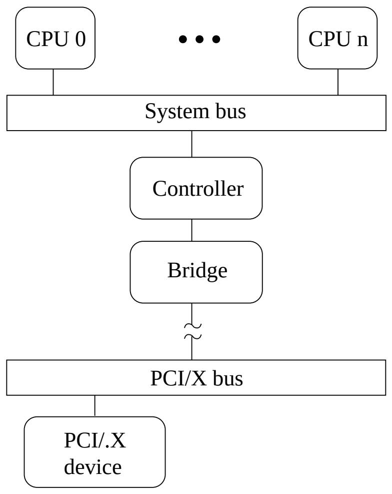

Figure 5: Address collision related bug: system sketch

For performance reasons, the hardware is trying to avoid a situation in which every write on the system bus is converted to a 'kill' by the controller, and sent down to the bridge. For that purpose, the controller maintains a list of 4K pages accessed by the bridge. A 'kill' is sent from the controller to the bridge only when an address in one of those pages is written to.

In the flawed design, this list of cached pages within the controller was not updated correctly. The bug was found because of the address collision testing knowledge mechanism. Without any further input from the verification engineer, the test-case generator produced collisions between read accesses from a PCI/X device, and write accesses from a processor. This triggered the 'kill' messages that exposed the bug.

7.3 Address Placement

The other bug was found in a state-of-the-art, high-end SoC design. One of the components in this system is responsible, as part of a wider operation, to perform a series of memory accesses, each of a potentially different size (number of bytes). The physical address of each access is the result of a translation process, that accepts a virtual address as input.

The virtual addresses are split into two—high order bits, and low order bits—and the two parts are given through two different means. A bug was found in a case in which an access is placed on the upper boundary of a translation page. To illustrate this point, consider a page size of 0x1000. An access of size 0x8, starting at virtual address 0xFF8, would, under some circumstances, cause the design to behave incorrectly[3].

This bug was found by the address placement testing knowledge mechanism, that aims at 'boundary' events of the type described above.

8. CONCLUSIONS

We defined testing knowledge as a general term that relates to methods whose purpose is to increase the quality of automatically generated stimuli. We then explained why the idea of testing knowledge is particularly suitable for system-level verification: it improves the quality of the verification, with a relatively low cost, in a place were time and resources are scarce, and in which verification is a significant challenge.

We then presented a set of testing knowledge mechanisms oriented at the system level. These include mechanisms that aim at increasing resource contention, methods targeted at address translation mechanisms, and methods that are based on understanding of the topology and configuration of the system.

We compared the coverage gained by two environments used for stimuli generation for the same design: one using testing knowledge, and another that does not. The test-case generator that used testing knowledge achieved better coverage, with significantly less simulation and human resources. We also described two sample bugs that were exposed by the usage of system-level testing knowledge.

The testing knowledge mechanisms and methods described in this paper were implemented in XXX, a random test-case generator used for the verification of several systems in YYY[4].

9. REFERENCES

[1] Allon Adir and Gil Shurek. Generating concurrent test-programs with collisions for multi-processor verification. In *IEEE International High Level Design Validation and Test Workshop*, Cannes, France, October 2002.

[2] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *32nd Design Automation Conference (DAC95)*, pages 279 – 285, 1995.

[3] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[4] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using constraint satisfaction formulations and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, August 2002.

[5] Cadence. Testbuilder. http://www.testbuilder.net.

[6] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test program generator. In *Design Automation & Test in Europe (DATE99)*, pages 434–441, Munich, March 1999.

[7] Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, and Avi Ziv. User defined coverage - a tool supported methodology for design verification. In *Design Automation Conference*, pages 158–163, 1998.

[8] Faisal Haque, Jonathan Michelson, and Khizar Khan. *The Art of Verification with Vera*. Verification Central, 2001.

[9] Infiniband Trade Association. Infiniband architecture specification, October 2000. Volume 1, Version 1.0.

[10] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.

[11] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann, 1994.

[12] Obsidian software. *Raven(TM) Software User's Manual*, May 2001. http://www.obsidiansoftware.com/files/manual.pdf.

[13] Verisity Design. Spec-based verification. http://www.-verisity.com/resources/whitepaper/specbased.html.

[14] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. Modeling design constraints and biasing in simulation using BDDs. In *International Conference on Computer-Aided Design (ICCAD '99)*, pages 584–590, Washington - Brussels - Tokyo, November 1999. IEEE.

[3]The actual bug was more complex, but we are prevented from providing more details due to intellectual property considerations

[4]The names of the tool and the organization are left out for anonymity.