A Novel Approach for Implementing Microarchitectural Verification Plans in Processor Designs*

Yoav Katz[1], Michal Rimon[2], and Avi Ziv[1]

[1] IBM Research - Haifa, Israel,
{katz, aziv}@il.ibm.com
[2] IBM Server and Technology Group, Haifa, Israel,
michalr@il.ibm.com

Abstract. The ever-growing microarchitecture complexity of processors creates a widening gap between the verification plan and the test generation technologies used in its implementation. This gap impacts the cost and quality of the verification process. To overcome this, we introduce a novel test generation platform for processor verification. This approach is based on a scenario description language that is close to the microarchitecture verification plan, and uses new test generation algorithms and a microarchitectural model to support this higher level of abstraction. Initial results on a high end industrial design show our approach reduces the effort of implementing a microarchitectural verification plan and improves the quality of verification.

1 Introduction

The goal of functional verification of processors is to establish the conformance of a processor design to its specification. Today's state of the art verification methodologies is based on a highly automated process that includes stimuli generation, checking, and coverage collection—combined with islands of manual labor [1]. Verification begins with the creation of a verification plan. The plan defines the aspects of the architecture and microarchitecture to be verified and the methods that will perform the verification. Test-case generators play a central role in such automated verification environments. The stimuli generated by these tools need to trigger architecture and micro-architecture events defined by the verification plan and ensure that all the dark corners of the verified design are exercised and the bugs hidden in them are exposed.

The input to a test-case generator is a test-template, which describes at a high level the desired characteristics of the generated test-cases. Given a test-template as input, the test-case generator generates a large set of architecturally valid test-cases that satisfy the template request and fill in the remaining details in a pseudo-random way.

* Please treat as confidential until publication; if declined for publication, please return to the authors as confidential information.

Existing processor-level test-case generators (such as [2,3]) provide a rich language for specifying requests at the instruction-level and a powerful instruction-based solving scheme for generating test-cases that satisfy the instruction level requests. This generation scheme calls for generation of instructions in execution order, one instruction at a time. The generation is interleaved with execution on a software reference model (ISS). This generation scheme has many advantages. First, it breaks the generation problem into a set of smaller, manageable sub-problems. In addition, it allows the generation engine to use the current processor state when generating the next instruction. Many tools formulate the generation of each instruction as a Constraint Satisfaction Problem (CSP) and thus achieve a high level of randomness and user controllability [4].

Advanced microarchitecture techniques such as out-of-order execution, on-chip caching and multi-threading, exploit the growth of available transistor count to deliver improved performance. As processor microarchitecture complexity increases, there is a growing need to thoroughly exercise the microarchitecture and reach all its corner cases. Advances in the verification methodologies and test-generation tools led to new features that target the microarchitecture. For example, tools embed testing knowledge [2] to increase the probability of generating interesting microarchitectural events (e.g., creating register dependency between instructions to trigger pipeline forwarding). The tools also include elaborate user control in the test-template to help the test-case reach specific microarchitectural events, and address the challenges of multithreaded and multiprocessing designs [5, 6].

Nevertheless, we observe a growing gap between the goals of the verification plan, which now targets events deep inside the processor, and the available test generation tools. This impacts both the resulting verification quality and the effort required to complete the verification process. One cause of this gap is the limited support for specifying and generating interactions between instructions. Specifically, users have to invest significant effort in creating the test-templates to generate the required intra-instruction dependencies and adapt them to the specific microarchitecture.

Another outcome of this methodology is that verification know-how as to the best ways to address microarchitecture verification is embedded in the test-templates, but not in the tools. Therefore, applying this knowledge in new test-templates requires significant effort. Moreover, less experienced verification engineers may be unaware of this knowledge and will not apply it in subsequent verification efforts.

There are other approaches for addressing the complexity of modern microarchitectures.. One approach calls for a test generator that is fully aware of all the microarchitectural implementation details. Armed with this knowledge and a strong solution engine, the test generator can generate test-cases that reach complex microarchitectural events [7,8]. The main problem with this approach is that creating and maintaining an accurate description of the microarchitecture can be impractical.

Coverage Driven Generation (CDG) is another way to addressing the difficulty of generating stimuli that targets complex microarchitectural events [9]. In this paradigm, machine learning techniques, such as Genetic Algorithms [10], Bayesian networks [9], Markov models [11] and inductive logic programming (ILP) [12], are used to learn the relation between test-templates and coverage points and modify the test-templates to improve coverage. While there is much research in this area [13], there are few successful applications of CDG in real industrial designs.

Automatic ways to embed microarchitectural testing knowledge into existing test generators were explored by Katz et al. [14]. In this approach, information is collected from simulation traces and automatically converted into instruction-level testing knowledge using machine learning classification algorithms.

In this paper we introduce Test Plan Automation (TPA), a novel test generation approach for processor verification. The approach is based on formulating a scenario description language that is close to the microarchitecture verification plan and using new test generation algorithms and a microarchitectural model to support this higher level of abstraction. Initial results show our approach reduces the effort of implementing a microarchitectural verification plan and improves the quality of verification.

The rest of this paper is organized as follows: In Section 2 , we present the concept and main components of our proposed method. We then describe each of these components in-depth in Sections 3-6. Section 7 describes the experimental results and we conclude in Section 8.

2 Solution Concept

The main goal of TPA is to improve the stimuli generation aspects of the implementation of the microarchitectural verification plan. This goal is achieved in two ways. First, TPA raises the level of abstraction of the test-template language and brings it closer to the verification plan while relying on a microarchitectural model to provide specific details on microarchitecture behavior. In addition, TPA closes the gap between the test-template and the generated test-cases using, new stream solving generation algorithms and scenario-level testing knowledge. These are depicted in Figure 1.

The test-template language used in TPA is designed to support the main ingredient of the verification plan, namely scenarios. The basic building blocks of the language are basic scenarios that target simple events that involve a single microarchitectural mechanism. A basic scenario is expressed as a set of instructions and the required constraints between them. An example of such scenario is two instructions that access the same cache line to create a cache hit. The language provides means, such as scenario combinations, to create more complex scenarios from the basic scenarios. For example, a cache hit and a cache miss scenarios can be combined to create a scenario that that hits on the L1 cache and misses on the L2 cache.

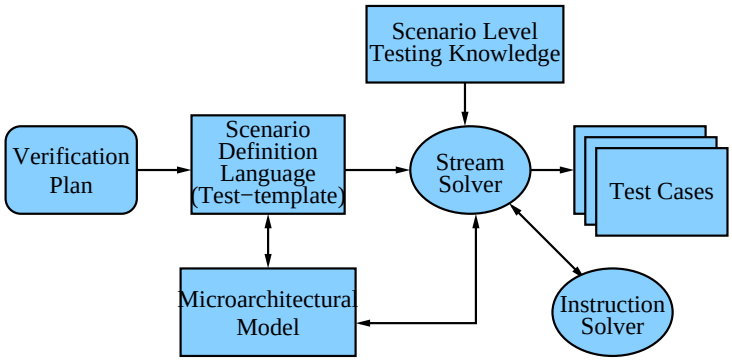

Fig. 1. TPA main components

Many of the parameters in the scenarios TPA needs to generate come from the microarchitectural mechanisms they operate on and many of the events TPA targets are relevant to several mechanisms. For example, cache hit events are relevant to all the caches in the system. To allow reuse of the scenarios between mechanisms, TPA uses a microarchitectural model that contains the important parameters of these mechanisms. When a scenario is generated, this information is used for filling in scenario details to create a specific scenario that targets the requested event in a specific mechanism.

TPA includes a new test-case generation scheme that is able to effectively satisfy constraints between instructions [15]. It formulates an abstract constraint satisfaction problem (CSP) that captures the essence of the requested scenario. This abstract CSP is solved incrementally and the abstract CSP solution is interleaved with single instruction generation.

To improve the quality of the test-cases it generates, TPA extends the notion of testing knowledge from the instruction-level to the scenario-level. Testing knowledge is the embodiment of expert verification knowledge in the tool such that the tool biases the stimuli toward interesting verification events without the need for explicit direction by the verification engineer. Scenario level testing knowledge automatically elaborates and modifies the original scenario to reach variants of the targeted event or other related events.

3 Microarchitectural Model

TPA is a tool for generating microarchitectural scenarios and thus, information about the microarchitecture is required to reach the needed events. To facilitate maximal reuse of scenarios, we separate the scenario description from the microarchitectural information and use a microarchitectural model that contains all the needed microarchitectural information. TPA does not attempt to provide a fully accurate model that guarantees that microarchitectural events are

reached by the scenarios. Instead, TPA aims to significantly increase the probability of reaching these events while minimizing the cost of model development and maintenance.

TPA captures the commonalities between microarchitecture mechanisms within the design and among different designs by forming an ontology of microarchitectural mechanisms. It defines an inheritance hierarchy of mechanism types, the properties that exist for each type and the basic behaviors that pertain to it. Figure 2 shows a graphic representation of part of this model that describes microarchitectural buffers. The type *Buffer* defines a set of properties which are shared among all microarchitectural buffers, this includes common properties such as *numEntries*, and a special set of properties that denote the type of instructions that read, write, and remove entries in the buffer. Inheriting from Buffer is *RandomAccessBuffer*, in which entries can be accessed in any order. This type specifies the conditions for four basic collision scenarios that apply to it: read-after-write (RAW), write-after-read (WAR), read-after-read (RAR) and write-after-write (WAW). A *MemoryRandomAccessBuffer* is a random access buffer that keeps memory data. It inherits from *RandomAccessBuffer* and adds an additional property *inputAddress* to specify whether access to this memory buffer is calculated based on virtual or real address values. Cache mechanisms are special cases of *MemoryRandomAccessBuffer*, and therefore they are defined as a subtype of it. The figure uses a lighter color for the actual design mechanisms that are defined as instances in the model. For example, the L1DataCache, L2Cache are defined as instances of *CacheMechanism* whereas the Load-Miss-Queue and *StoreReorderQueue* are defined as instances of *MemoryRandomAccessBuffer*.

The ontology helps maximize the reuse of scenarios. For example, scenarios that target a 'buffer full' event can be applied to any mechanism derived from *Buffer*, ranging from the store reorder queue (SRQ) to caches to register rename buffers. Localizing all the mechanism properties in a single location simplifies the overall maintenance effort of the verification process and encourages structure and rigor.

4 Scenario Input Language

TPA provides a high-level scenario description language. It has constructs for defining scenarios as a set of instructions and the constraints between them. In addition, given a collection of predefined scenarios the language has constructs for defining new scenarios that instantiate them in several combination options.

4.1 Scenario Definition

A scenario definition starts with a declaration of the instructions that participate in the scenario and the mechanisms to which the scenario applies. Each instruction declaration statement may specify a single instruction or a set of instructions. In the latter case, the user needs to specify lower and upper bounds on the number of instructions in the set. In addition, the declaration can restrict

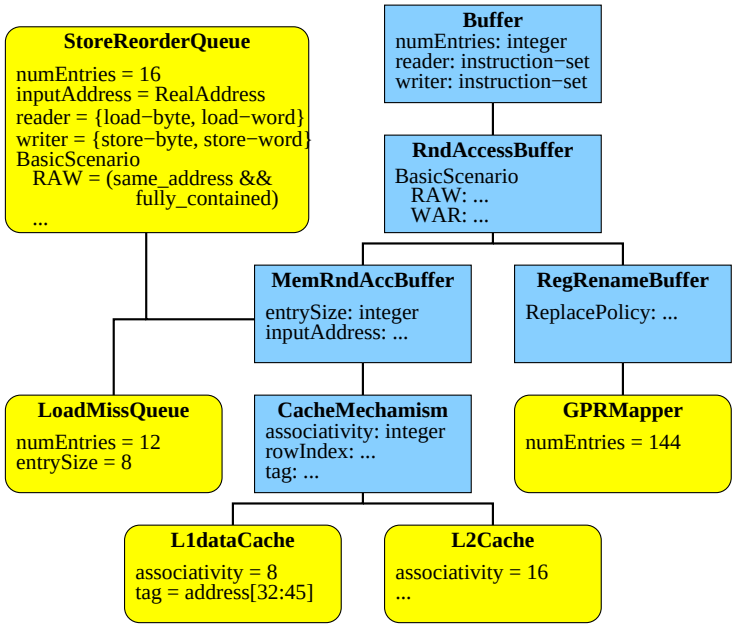

Fig. 2. Microarchitectural model ontology

the instructions to a specified type. A mechanism declaration statement specifies a mechanism type or a specific mechanism instance. If the declaration specifies a mechanism type, each scenario instantiation can be restricted to a derived type or a particular instance of the specified mechanism type.

Consider a cache-replace event; caches are arranged into rows, where each row can contain multiple cache lines, depending on the cache associativity. Each memory address is mapped to a specific row which is calculated based on some bits in the address. Within the row, cache lines are identified by tags, which are formed by other bits in the address. A cache-replace event occurs when all the entries in the row are used, and a new address with a new tag is mapped to the same row. In this case, one of the existing cache lines needs to be evicted. The following is a high level description of a scenario that targets a cache-replace event:

1. Generate at least $n+1$ instructions that access memory, where n is the cache associativity
2. All instructions should access the same row in the cache
3. At least $n+1$ instructions should have a different tag

Figure 3 shows the TPA definition of the cache-replace scenario. The scenario can be applied to any mechanism M1 of type *CacheMechanism*. The instruction declaration of the scenario states that the scenario requires a set of instructions,

with a size larger than the *associativity* of the cache. The scenario puts an upper limit on the number of instructions. Each instantiation of the scenario will generate a random number of instructions within the specified limits. All the instructions are of type *M1.Writer*. This type is defined in the mechanism, and includes all the instructions that can write to the cache (e,g., loads and stores).

```
ScenarioDefinition Cache-Replace
Mechanisms:
        M1          type=CacheMechanism;
Instructions:
        accessors   type=M1.Writer
                    size=[M1.associativity+1,2*M1.associativity]
Constraints:
        SomeDiff
                    mechanism=M1
                    instructions=accessors
                    lowerLimit=M1.associativity+1
                    property=tag
        AllSame
                    mechanism=M1
                    instructions=accessors
                    property=row
```

Fig. 3. Cache-replace scenario description

In addition, each scenario definition has to include a declarative description of the constraints between its instructions. We distinguish between two types of constraints: constraints that control the interactions between instructions and constraints that control the placement of instructions in the test.

Constraints that control the interaction between instructions are divided into two groups: *Property constraints* request that a property value be the same/different for all the instructions in the specified set and *Mechanism behavior constraints* target a basic mechanism behavior and are parameterized according to properties of the mechanism.

Constraints that control the placement of instructions in the test are divided into three groups. *Order constraints* specify a required partial order between two specific instructions in the generated test-case. Unlike traditional test generators, TPA does not assume that the order of appearance in the scenario implies any order in the resulting test. *Distance constraints* specify how many instructions are allowed between any two specified instructions. TPA fills the space between two instructions with "non-scenario" instructions. These instructions may belong to a different scenario or may be selected by testing knowledge. *Thread constraints* specify for any set of instructions whether they should be generated on the same or on different threads.

In the cache-replace example, two property constraints enforce the scenario restrictions on the instructions' cache row and tag properties. When the scenario is instantiated on a specific cache mechanism, the mechanism is accessed to obtain the row and tag calculation methods that apply to it.

Figure 4 shows a scenario for targeting a read-after-write collision event in a buffer using the *MemoryCollision* constraint. The constraint operates on pairs of instructions and a mechanism of type *MemoryRandomAccessBuffer*. It enforces collision conditions on the memory accesses of the instructions according to a set of parameters provided by the mechanism. For example, when this scenario is applied to the *StoreReorderQueue* shown at the bottom left of Figure 2, the mechanism parameters specify that instr1 that writes to the buffer is a store instruction and instr2 that reads from the buffer is a load instruction. In addition, the mechanism provides the *MemoryCollison* constraints the exact nature of the collision: same address and fully contained, meaning that the load and store instructions access the same memory location and the data of the load is contained in the data of the store.

```
ScenarioDefinition Read-After-Write
Mechanisms:
        M1          type=MemoryRandomAccessBuffer;
Instructions:
        instr1      type=M1.writer
        instr2      type=M1.reader
Constraints:
        Order(instr1, instr2)
        SameThread(instr1, instr2)
        MemoryCollision
                    mechanism=M1
                    instructions=(instr1 instr2)
                    collisionType=RAW
```

Fig. 4. Read-After-Write scenario description

Note that the scenario can be applied as is to any other instance of *MemoryRandomAccessBuffer* such as the *LoadMissQueue*, resulting in a totally different sequence of instructions.

4.2 Scenario Instantiation

Given a scenario definition, each instantiation of the scenario can request that the scenario be applied to a desired subtype of the declared mechanism type or to a specific instance. Figure 5 shows several possible invocations of the Read-After-Write scenario. In the first invocation the user requests an instantiation of the Read-After-Write scenario to any arbitrary design mechanism. In this case, the user request is combined with the restrictions specified in the scenario definition

and the generated test-cases will target any mechanism that is defined in the microarchitectural model as an instance of *MemoryRandomAccessBuffer*. In the subsequent invocations, the user requests that the Read-After-Write collision occur on one of the cache mechanisms, or specifically on the store reorder queue.

```
Read-After-Write ()
Read-After-Write (CacheMechansim)
Read-After-Write (StoreReorderQueue)
```

Fig. 5. Possible Read-After-Write scenario instantiations

4.3 Scenario Combinations

Scenario combinations are important because they can cause several events to occur in a small time window by having the same instructions take part in multiple scenarios, or stress a specific mechanism by instantiating multiple scenarios for that mechanism. TPA supports the definition of scenarios that instantiate previously defined scenarios. When a scenario is instantiated by another scenario, the selection of instructions and mechanisms to use has to satisfy restrictions expressed by both scenarios.

Consider the combined scenario depicted in Figure 6, which creates two different types of events on two mechanisms in a small time window: a cache replace on some cache and a read-after-write collision on some internal buffer. Here, the cache-replace scenario determines the set of instructions for the scenario and the read-after-write scenario operates on two random instructions that participate in the cache scenario.

```
ScenarioDefinition LSU_Stress
Mechanisms:
        cache       type=CacheMechanism
        buffer      type=MemoryRandomAccessBuffer
Instructions:
        instrSet
Constraints:
        Cache-Replace(M1=cache, accessors=instrSet)
        Read-After-Write(M1=buffer,
                instr1=instrSet[random],
                instr2=instrSet[random])
```

Fig. 6. Combining scenarios

5 Generation Scheme

TPA generates a scenario in two main steps. First, the scenario definitions are parsed and several high level decisions are made. These decisions include the selection of mechanism instances that were not completely specified and the selection of instruction set sizes. Once these decisions are made, the number of the instructions and the relevant constraints are known and TPA creates a constraint graph that represents this particular scenario instantiation. The nodes in the graph are instructions and the arcs represent scenario constraints between sets of instructions. In the second part of the generation process, the constraint graph is passed to a scenario solver for generating instruction streams that satisfy the user request. The challenge lies in having the test generator effectively generate test-cases that consist of sequences of instructions satisfying these constraints.

A test generation approach that generates instruction by instruction is not suitable for this problem because of its inability to consider constraints emanating from instructions later in the sequence when the current instruction is solved. This would cause the generator to make early decisions that may lead to generation failure of dependent instructions later in the stream. Trying to formulate and solve the entire scenario as a single CSP is not a feasible approach as the size of the resulting CSP would make this problem intractable.

To address this, TPA implements an abstraction-refinement approach to scenario generation [15]. It formulates an abstract constraint satisfaction problem that captures the essence of the requested scenario and interacts with an instruction-based test generator for single instruction generation.

The abstract CSP contains CSP variables that determine for each instruction its identity (mnemonic), identity of the thread for which it will be generated, and the location in the program order of that thread (timestamp). In addition the stream constraints add the relevant CSP variables to all the participating instructions. For example, in the CSP that is generated by the combined scenario in Figure 6, the *MemoryCollision* constraint which implements the read-after-write scenario adds variables to represent the real address and length of the memory access of each instruction, while the property constraints that implement the cache replace scenario add variables that represent the cache tag and row of the address.

The abstract CSP propagates constraints between all instructions, including constraints that influence earlier instructions based on restrictions from later instructions. When constraint propagation subsides, the instruction with lowest timestamp value is selected as the next instruction to be generated. The restrictions imposed by the stream constraints on the instruction are provided as input to a single instruction generator. This generator generates the specific instruction, taking into account all the instruction-level constraints necessary for generating an architecturally valid instruction. Once the first instruction is generated, all decisions that were made and are relevant to the rest of the scenario are propagated back to the abstract problem and the process continues.

Since the thread and location of the each instruction in program order are CSP variables they can be randomly selected. Hence instructions can be generated in many orders and interleavings in the final test-case. These instructions could have originated from the same scenario or from different scenarios that were combined.

6 Scenario Testing Knowledge

Testing knowledge is a way of embedding the knowledge and expertise of the verification engineer in a random stimuli generator that utilizes it to bias the generator towards interesting events. The raised level of abstraction in TPA opens the door for new, scenario-level, testing knowledge that can be used to improve the quality of the generated test-cases.

One area in which testing knowledge plays a major role is creating an interesting microarchitectural state for the requested scenario to operate in. This is done in TPA in two main ways: selecting an interesting order and placement for the instructions in the scenario and adding background instructions to vary the microarchitectural state. For example, TPA may choose to place two instructions involved in a collision close to each other to increase the probability of them fetching together. In addition, it may insert a background instruction that causes the first instruction in the collision to stall, so that the instructions are executed out-of-order.

Another important type of scenario level testing knowledge used in TPA is scenario mutations. The goal of mutations is to reach simulation events that are not the original intent of the scenario but are related to it. The tool supports several types of mutations such as microarchitectural model mutations that change the behavior of the mechanisms that the scenario applies to and mutations that execute parts of the scenario in a speculative path.

In addition to the scenario-level testing knowledge, TPA also takes advantage of instruction level testing knowledge provided to it by the single instruction generator. Users can control the application of both instruction-level and scenario-level testing knowledge as part of the scenario description and thus convey their own judgment as to what testing knowledge is more relevant to a given scenario at a given stage of the verification process.

7 Experimental Results

TPA implements the scenario-based generation approach described in the previous sections. TPA utilizes the instruction solving capabilities of Genesys-Pro, a leading commercial instruction-based test generator. We demonstrate the advantages of scenario-based generations of TPA over the instruction-based Genesys-Pro by comparing the two tools in their ability to cover the Store Reorder Queue (SRQ) microarchitectural feature of a high-end Power processor. The SRQ is a buffer found in the Load Store Unit (LSU) of processors. It keeps the data of

each store instruction internally in the processor until the store instruction completes. This prevents wrong updates to the caches (and the rest of the system) when the store instruction does not complete for any reason and helps maintain the ordering rules between stores. One of the roles of the SRQ is to provide data to newer load instructions, thus avoiding stalling the processor until the store instruction completes. Therefore, read-after-write (RAW) collisions in the SRQ (also called load-hits-store and abbreviated to LHS in the rest of the section) are an important item in the verification plan of the processor.

The verification plan for LHS calls for test-cases that create all interesting read-after-write collisions in the SRQ, such as out-of-order collisions, simultaneous accesses to the buffer, and more. The implementation of the verification plan is monitored using a coverage model defined by the design team, whose goal is to ensure that all interesting events in the SRQ occur.

We compare the ease of creating test-templates for TPA and Genesys-Pro that implement the LHS item in the verification plan and the quality of the test-cases generated from these test-templates by both tools. Table 1 summarizes the first part of the comparison. In Genesys-Pro, each requested type of collision needs to be encoded specifically in the test-template. This encoding includes properties of the collision that originate from the SRQ mechanism. As a result, 13 test-templates, each handling a different type of collision or near collision of interest, are needed to cover this single item in the verification plan. While these test-templates are carried over between generations of the same architecture, adapting them to each generation takes effort because the test-templates need to be adapted to the microarchitecture in many places. The reliance on architectural and microarchitectural features of the design makes it virtually impossible to reuse these test-templates in different architectures.

Table 1. Comparison of ease in creating test-templates

	Genesys-Pro	TPA
Number of test-template	13	3
Encoding SRQ behavior	Test-templates	Model
Reusability across designs	Needs effort	Easy
Reusability across architectures	Impossible	Easy
Combinations with other scenarios	Hard	Easy

In TPA, on the other hand, it is easy to stipulate specific collisions in a test-template and fit it to a specific mechanism in the microarchitectural model. As a result, only a small number of test-templates are needed to implement the LHS item in the verification plan. In this comparison, we used three test-templates: 1) a simple test-template, shown in Figure 4, that creates many instances of basic RAW collisions in the SRQ; 2) a test-template that combines these collisions with other scenarios, such as group formation and cache scenarios (similar to the test-template in Figure 6); and 3) a test-templates that includes mutations of the basic scenario and the mechanism to create interesting near-collisions. It

is important to note that the three test-templates are needed only to illustrate the benefits of various features of TPA. For actual verification purposes, the last template suffices.

The simplicity of the TPA test-templates and the fact that most of the relevant information for the collisions comes from the architectural and microarchitectural models, makes reuse of this test-templates across design and architectures easy. In fact, we used the same test-templates to generate test-cases for several Power and zArchitecture designs.

The second part of the comparison evaluates the quality of the tests generated by both tools. Here we compared the ability of the tools to hit the LHS coverage events defined by the design team. A summary of the results is shown in Table 2.

Table 2. Comparison of the quality of generated tests

	Genesys-Pro	TPA			
		Simple	Comb	Mutation	Total
Test-cases	769	295	475	511	1281
Cycles	20M	11M	10M	10M	31M
LHS events covered	46	41	49	50	51
Other LSU events covered	1519	1063	1587	1523	1715
Generation time per instruction	1.50	1.28	1.45	1.52	1.43

The first two lines in the table provide information on the number of test-cases and simulation cycles used. We believe that the simulation cycles count, and not the number of test-cases, is a fairer base for comparison, so our goal was to have twice as many cycles for all the Genesys-Pro test-cases combined than cycles for test-cases from each of TPA test-templates.

Comparison of the LHS coverage results shows that the simple TPA test-template reaches lower coverage than Genesys-Pro. This can be explained by the fact that the simple test-template does not try to create near-collisions. Each of the two other TPA test-templates, which combine the basic LHS scenario with other scenarios and use mutations to create near-collisions, achieve better coverage than the 13 Genesys-Pro templates. This indicates better test-case quality. Another evidence for the superior quality of TPA test-cases is the coverage of other LSU events (that are not targeted by any of the templates) by the TPA test-cases. To show that the higher quality of the generated test-cases is not caused by increased generation time, the last row in Table 2 compares the average generation time per instruction for the compared test-templates. This time is calculated by dividing the total generation time of a test-case by the number of scenario instructions it contains. Therefore, for the TPA test-templates, this time includes the time needed to construct and solve the stream CSP as well as the time needed to generate each of the instructions. The row shows that despite using a more sophisticated generation scheme, the generation time per instruction in TPA is similar or lower. This can be explained by the planning

done in the TPA generation scheme, which reduces the number of instruction generation failures.

8 Conclusions

The growing complexity of microarchitectures creates a widening gap between the verification plan and test generator input languages used to implement it. This impacts the cost and quality of the verification process. In this paper, we proposed a novel method of test generation. Our method is based on a high-level scenario description language that is close to the microarchitecture verification plan, and a new test generation algorithm and microarchitectural model to support this higher level of abstraction. Experimental results show that the proposed method is indeed capable of achieving test-cases with higher coverage, lower test-template development costs, and comparable generation time, when evaluated against the existing state-of-the-art test generation solution.

Future development directions of the technology include extension of the scenario-based testing knowledge to other areas such as multithreading, integration of automatic methods for populating the microarchitecture model, and full scale deployment in the verification of current high-end processor designs.

References

1. Wile, B., Goss, J.C., Roesner, W.: Comprehensive Functional Verification - The Complete Industry Cycle. Elsevier (2005)
2. Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., Ziv, A.: Genesys-Pro: Innovations in test program generation for functional processor verification. IEEE Design and Test of Computers **21**(2) (2004) 84–93
3. Hennenhoefer, E., Typaldos, M.: The evolution of processor test generation technology. (http://www.obsidiansoft.com/pdf/evolution.pdf)
4. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine **28**(3) (2007) 13–30
5. Ludden, J., Rimon, M., Hickerson, B.G., Adir, A.: Advances in simultaneous multithreading testcase generation methods. In: Proceedings of the 6th Haifa Verification Conference. LNCS 6504, Springer-Verlag (2010) 146–160
6. Burns, D.: Pre-silicon validation of hyper-threading technology. Intel Technology Journal **6**(1) (2002)
7. Adir, A., Bin, E., Ziv, A.: Piparazzi: A test generator for micro-architecture flow verification. In: Proceedings of the High-Level Design Validation and Test Workshop. (2003) 23–28
8. Mishra, P., Dutt, N.: Specification-driven directed test generation for validation of pipelined processors. ACM Trans. Design Autom. Electr. Syst. **13**(3) (2008)
9. Fine, S., Ziv, A.: Coverage directed test generation for functional verification using Bayesian networks. In: Proceedings of the 40th Design Automation Conference. (2003) 286–291
10. Squillero, G.: MicroGP—an evolutionary assembly program generator. Genetic Programming and Evolvable Machines **6**(3) (2005) 247–263

11. Wagner, I., Bertacco, V., Austin, T.: Microprocessor verification via feedback-adjusted Markov models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**(6) (2007) 1126–1138
12. Eder, K., Flach, P., Hsueh, H.W.: Towards automating simulation-based design verification using ILP. In Muggleton, S., Otero, R., Tamaddoni-Nezhad, A., eds.: Inductive Logic Programming. Volume 4455 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 154–168
13. Ioannides, C., Barrett, G., Eder, K.: Feedback-based coverage directed test generation: An industrial evaluation. In: Proceedings of the 6th Haifa Verification Conference. LNCS 6504, Springer-Verlag (2010) 112–128
14. Katz, Y., Rimon, M., Ziv, A., Shaked, G.: Learning microarchitectural behaviors to improve stimuli generation quality. In: Proceedings of the 48th Design Automation Conference. (2011) 848–853
15. Katz, Y., Rimon, M., Ziv, A.: Generating instruction streams using abstract CSP. In: Proceedings of the 2012 Design, Automation and Test in Europe Conference. (2012) 15–20