Generating Instruction Streams Using Abstract CSP

Yoav Katz, Michal Rimon, Avi Ziv
IBM Research - Haifa, Israel
Email: {katz, michalr, aziv}@il.ibm.com

***Abstract*—One of the challenges that processor level stimuli generators are facing is the need to generate stimuli that exercise microarchitectural mechanisms deep inside the verified processor. These scenarios require specific relations between the instructions participating in them. We present a new approach for processor-level scenario generation. The approach is based on creating an abstract constraint satisfaction problem, which captures the essence of the requested scenario. The generation of stimuli is done by interleaving between progress in the solution of the abstract CSP and generation of instructions. Compared with existing solutions of scenario generation, this approach yields improved coverage and reduced generation fail rate.**

I. Introduction

Modern processors use a combination of architectural and microarchitectural innovations to improve their performance and power consumption [1]. This significantly increases the complexity of processor design and its verification. Generating processor-level stimuli is one of the main challenges in the verification of processors. The need to thoroughly exercise the microarchitecture and reach all its corners led to the development of sophisticated stimuli generators that are based on techniques such as machine learning [2], constraints satisfaction [3], [4], and formal methods [5].

One of the challenges that processor-level stimuli generators are facing is the need to generate stimuli that exercise microarchitectural mechanisms deep inside the verified processor. These scenarios require specific relationships between the instructions participating in them. For example, creating an interesting collision in a memory buffer requires two instructions to access the same memory location. These scenario requirements, in turn, propagate constraints between the instructions. Specifically, they can propagate constraints from a late instruction to an early one.

Many processor-level stimuli generators, such as Genesys-pro [6], generate stimuli in execution order, one instruction at a time. This generation scheme has many advantages. It breaks the generation problem into a set of smaller, manageable sub-problems. It also allows the generation engine to use the current processor state when generating the next instruction. For example, when generating an add instruction, the generator can select registers that contain certain values to generate an interesting result, such as an overflow. The disadvantage of this generation scheme is the difficulty in propagating the constraints from late instructions to early ones. This is because it is difficult to automatically analyze the requirements of late instructions and propagate them to the early ones. A common solution to this problem is for the verification engineer to manually analyze such requirements and specify them explicitly in the test template. However, this solution requires a lot of effort and expertise.

978-3-9810801-8-6/DATE12/©2012 EDAA

A different approach, which automates this requirements propagation process, relies on abstractions. With this approach, an abstract problem that captures the essence of the required scenario is created and solved to propagate the requirements between instructions. Once the requirements are propagated, the stimuli generator can generate the instructions one at a time while incorporating the solution of the abstract problem. This approach is used in stimuli generators such as X-Gen [7]. The disadvantage of this approach is that the solution of the abstract problem may cause early decisions that can lead to infeasible instruction generation problems. The use of abstraction refinement techniques to solve difficult problems is common. In addition to the stimuli generation example above, abstraction refinement techniques are also used in formal verification to reduce the size of the problem and avoid state-space explosion [8]. Abstraction refinement is often used in artificial intelligence (AI). For example, a complex reactive planning may be solved in two phases. First, the rough plan is created using an abstract problem. Then, each step of the abstract solution is refined to get the actual plan [9], [10].

This paper proposes a new approach for processor-level stimuli generation that addresses the difficulty of generating complex scenarios. This approach uses abstract problems to capture the essence of the requested scenario. Unlike other abstraction-based approaches, our approach does not completely solve the abstract problem before generating the individual instructions. Instead, the solution of the abstract problem and generation of individual instructions are interleaved. Specifically, we progress just enough in the solution of the abstract problem to propagate the requirements from the late instructions to the first one. We then generate the first instruction and propagate its solution back to the abstract problem and continue to progress the abstract problem solution towards the second instruction. This process continues until all the instructions are generated. The interleaving between the abstract problem solution and the generation of individual instructions eliminates (or at least minimizes) the unnecessary early decisions made by the abstract problem solver and increases the chance of avoiding dead-ends.

We implemented the proposed approach on top of the existing generation engine of Genesys-Pro. The implementation automatically creates the abstract stream problem from the scenario specification and models it as a constraint-

satisfaction problem (CSP) [3]. The generation engine then alternates between progress in the stream problem using an arc consistency algorithm [11] and the generation of individual instructions. Experimental results show that this new approach can significantly improve the generation success rate even for simple scenarios when compared to basic test templates of Genesys-Pro. It also reduced the effort needed to implement high-quality test templates.

The rest of the paper is organized as follows. In Section II, we illustrate the problem we solve using a simple scenario example. Section III describes our new generation scheme. Section IV provides experimental results. We present our conclusions in Section V.

II. Simple Scenario Example

To illustrate the problem and demonstrate the advantages of our solution, we use a collision in the load miss queue (LMQ) buffer. The LMQ is a buffer that is commonly found in load store units of processors. The buffer stores cache access requests that miss in the L1 cache. These requests are stored in the LMQ until the data arrives at the cache, at which time the execution of the missing instruction is resumed. When a cache access is stored in the LMQ, additional accesses to the same memory location cause collisions in the LMQ that lead to special treatment of the colliding instructions, such as rejects and flushes. More specifically, in the implementation of the LMQ of the verified processor, data read accesses to memory that miss in the L1 cache are kept in the LMQ, while data write accesses may bypass the cache and are not stored in the LMQ. However, if they hit an address that is in the LMQ, they are flushed and the instruction that wrote to the LMQ is rejected[1].

The specific scenario we are interested in calls for a read-after-write (RAW) collision in the LMQ. The specific requirements for the collision scenario coming from the LMQ specification are the following: The scenario includes two instructions. The first instruction needs to write to the LMQ. Because data read accesses to memory that miss in the L1 cache are written to the LMQ, this instruction needs to read data from memory. The second instruction needs to read from the LMQ, but not write to it. This means that the second instruction should write data to memory. The specification of the LMQ states that an interesting RAW collision occurs when the reader from the LMQ shares the same address as the writer and its data is fully contained in the writer data. In other words, the reader and writer have the same address and the reader access length is less than or equals the writer access length. In addition to these requirements, there are several other sources for requirements that affect the generated scenario. For example, exceptions clear the pipelines and buffers of the processor. Therefore, avoiding exceptions during the execution of the scenario is needed to allow interesting collisions in the LMQ. In addition, to achieve coverage closure, we may be interested in generating the scenario when the second instruction is a `stfd` (store floating-point double) instruction.

[1]This specification of the LMQ and its behavior is not taken from any specific processor, instead it was devised from behaviors of several LMQ buffers to illustrate our new generation scheme.

Examination of the requirements for the RAW scenario reveals several constraints they impose on the instructions in the scenario. To achieve a write to the LMQ, an instruction that reads from memory is required. Similarly, to fulfill collision requirements, the second instruction needs to write to memory. The coverage requirement specifies a specific instruction for the second instruction, namely `stfd`. This, in turn, adds several more constraints to the second instruction. First, the length of the memory access of the `stfd` instruction is eight bytes. In addition, to avoid exceptions, the memory access needs to be eight bytes aligned and the memory page the instruction accesses needs to be write-enabled. These constraints affect the first instruction as well. Because the collision requires the two memory accesses to the same address, the memory access of the first instruction also needs to be eight bytes aligned. In addition, the memory access of the second instruction needs to be fully contained in the first one, so the access length of the first instruction needs to be eight bytes or longer. Finally, because the two instructions access the same page in memory, this page needs to be write-enabled (even though the first instruction only reads from memory). All this results in many constraints on both the mnemonic of the first instruction (e.g., `lbz`, which reads a single byte from memory cannot be used because of the access length) and its operands (e.g., the eight byte alignment).

The constraints that propagate from the second instruction in the scenario to the first cause difficulties for stimuli generators that generate the instructions in the test program one by one, in their execution order, such as Genesys-Pro [6]. There are several ways to address this. One solution is to put the burden of specifying all the constraints discussed above on the user. This solution calls for the user to specify these constraints in the test template. The main disadvantage of this approach is the significant effort required to create and maintain all the test templates that are needed to implement it. A different approach is to a use smaller abstract description of the required test. With this approach, the abstract description is used to propagate the constraints and guide the generation of the individual instructions in the test.

III. Stream Generation Using Abstract CSP

The main disadvantage of using abstract problems to assist in the solution of stream generation is that during the solution of the abstract problem early decisions are taken that may lead to infeasible instruction generation. For example, deciding the accessed memory in the abstract problem may lead to an address that is not accessible by the processor, thus causing instruction generation to fail. We present here a new approach to stream level generation that overcomes this disadvantage. This new approach is based on the creation of an abstract problem that captures the essence of the requested scenario. The novelty of this approach is the way the abstract problem is solved. We do not completely solve the abstract problem

before generating individual instructions. Instead, we interleave progress in the solution of the abstract problem with the generation of individual instructions. In this way, the abstract stream problem helps select the mnemonics needed to achieve the required scenario and can propagate constraints arising from the scenario between the instructions while minimizing the early interfering decisions.

Our method is based on modeling the instruction stream in the scenario as a constraint satisfaction problem (CSP) [3]. A constraint satisfaction problem consists of a finite set of variables, where each variable is associated with a domain of values, and a set of constraints. A constraint is a relation defined on some subset of the variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. A constraint network is a hyper-graph whose nodes are the variables of a CSP and whose hyper-edges ('arcs') represent constraints. Maintaining arc consistency (MAC) [11] is a commonly used family of CSP solution algorithms that filters invalid solutions using the arc-consistency procedure. A constraint (arc) is said to be consistent if, for any variable of the arc, and any value in the domain of that variable, there is a valid assignment to the other variables of the arc which satisfies the constraint. The general idea behind the MAC algorithm is to perform arc-consistency on all constraints until a fixed-point is reached, randomly reduce the domain of one (or more) of the variables, and repeat this process until a solution is reached. In our method, the random reduction of the variables' domain is done through the generation of individual instructions.

Figure 1 describes the generation process in our approach. The process comprises two main steps: construction of the abstract stream problem and the actual test generation, which combines progress in the solution of the stream problem with generation of instructions. It is important to note that the process described in the figure and the rest of the section is done automatically by the stream generator.

We begin the generation process by creating a CSP model of the requested scenario. This model includes all the instructions that participate in the scenario; for each instruction several variables that are important to the scenario are included in the CSP. Specifically, for each instruction, we add a fixed set of variables, such as its mnemonic and exception indicator to the CSP. Additional sets of variables relating to operands of the instruction are added if needed by the scenario.

Figure 2 shows the CSP model for the RAW collision in the LMQ. The collision scenario has two instructions, leading to two sets of variables in the model. The left side of each instruction contains the fixed set of variables for the instructions. The collision scenario involves a memory collision. Therefore, a memory operand with several variables, such as address and length, is added to each instruction and appears in the right side of the figure. Note that instructions can have more than one operand attached to them and each instruction can have different operands. For example, if the collision scenario also included a write-after-write (WAW) register collision between the second instruction and a new third instruction, the second instruction would have contained two operands, one memory and one register, and the third instruction would have contained only one register operand.

```
// Build the stream CSP
add all instructions to stream
for each instruction
   add variables and relevant operands
   add intra-instruction constraints
add scenario and ordering constraints

// Generate the test
do
   reach fixed-point in the stream CSP
   select instruction with minimal timestamp
      set its timestamp
      randomly select a mnemonic
   call Genesys-Pro instruction generation engine
      build CSP of selected instruction
      connect the stream CSP to it
      generate the instruction by solving its CSP
      propogate values from instruction CSP to stream CSP
until all instructions are generated
```

Fig. 1. Stream generation process

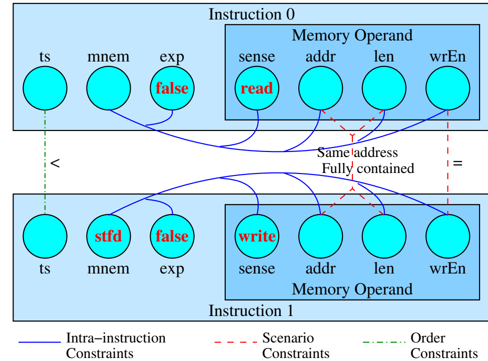

Fig. 2. Stream CSP for the RAW collision

Next, we add the constraints to the CSP model to construct the stream CSP. These constraints are shown in Figure 2 as edges in the graph. We add three types of constraints: intra-instruction constraints, scenario constraints, and order constraints. Intra-instruction constraints describe the relations between each mnemonic and the rest of the variables in the instruction. We add intra-instruction constraints for each instruction in the model and for each possible mnemonic. These constraints are automatically extracted from the architectural specification of the processor. They serve to remove infeasible mnemonics based on the values of the instruction variables and to set the value of the instruction variables when a specific mnemonic is given. For example, setting the memOp.sense value to `read` will remove all the mnemonics of instructions that do not read from memory. Figure 3 shows some of the intra-instruction constraints of the `stfd` instruction. The intra-instruction constraints appear as blue solid edges in Figure 2.

The scenario constraints are constraints that come from the requirement of the scenario. These constraints usually relate

mnem = stfd $\rightarrow$ memOp.len = 8 and
memOp.sense = write and
memOp.wrEn = 0 $\rightarrow$ exp and
memOp.addr not aligned $\rightarrow$ exp and
...

Fig. 3. Intra-instruction constraints for `stfd`

variables from two or more instructions. For example, the same-address fully-contained requirement of the LMQ RAW collision appears in the stream model as a constraint connecting the address and length variables of the two instructions. In addition to constraints that relate several instructions, the scenario requirements can also affect variables of individual instructions. For example, in our collision, the scenario requires that the first instruction reads from memory and the second one writes to it. This is reflected in the CSP model by setting the values of the sense variable in the two instructions to `read` and `write`, respectively. In Figure 2, the values of all the variables affected by the scenario requirements appear as red values in the variables.

We use a special type of scenario constraints to specify the order of instructions in the generated test. These constraints relate the timestamp (ts) variables of the instruction. For example, in our collision scenario it is required that the load instruction be executed (and therefore generated) before the store instruction. Therefore, we add a constraint specifying that the timestamp of the first instruction is smaller than that of the second instruction.

Once we have completed the construction of the stream CSP, the generation of the test can begin. The first step in the solution is to apply the arc consistency procedure on the constraints in the stream CSP until a fixed-point is reached. This fixed-point helps propagate the scenario requirements between the instructions and helps filter out mnemonics that cannot participate in the scenario. In our collision scenario, applying arc consistency to the intra-instruction constraints of `stfd` in the second instruction, as shown in Figure 3, sets the domains of most of its variables to single values. Specifically, the domain of the length operand is set to 8 because `stfd` writes 8 bytes to memory. Similarly, to avoid exceptions, the write-enabled variable is set to `true` and the address variable is set to be 8 bytes aligned. Arc consistency on the scenario constraints updates the domains of the first instruction. For example, the same-address fully-contained constraint sets the address of the first instruction to be eight bytes aligned and its length to be at least eight bytes. The intra-instruction constraints of the first instruction filter out mnemonics that cannot participate as the first instruction in the scenario. For example, `stfd` cannot be the first instruction because it does not read from memory and `lbz` cannot participate because it reads just one byte. Therefore, both instructions (as well as many other) are removed from the domain of the first mnemonic. Finally, the ordering constraints reduce the domain of the timestamp of the second instruction to be greater than or equal to 2 because its timestamp needs to be greater than

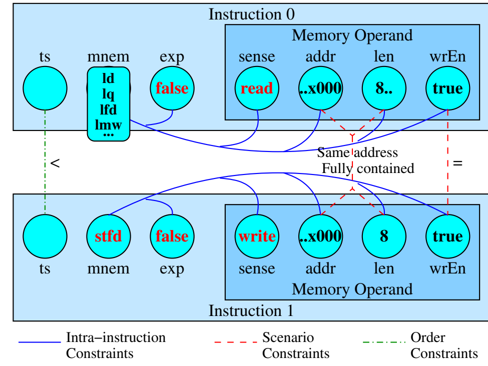

Fig. 4. Stream CSP for the RAW collision after first fixed-point

the timestamp of the first instruction.

Figure 4 shows the state of the stream CSP after the first fixed-point is reached. The figure clearly shows that the objectives of the fixed-point, namely propagating the constraints and filtering inappropriate mnemonics, are achieved in the RAW collision example.

After the fixed-point is reached, we are ready to generate the first instruction in the test. To do that, we first select the instruction with the smallest possible timestamp and sets its timestamp to the minimal value. If there are several instructions with the minimal timestamp, we select one randomly. In our collision scenario, we select the first instruction and set its timestamp to one. Next, we randomly select a mnemonic for this instruction from the domain of the mnemonics. Because of the generation scheme of Genesys-Pro, this is a random decision we cannot avoid in the abstract problem. Once the mnemonic is selected, we can call the instruction generation engine of Genesys-Pro to generate the instruction with the specific constraints that are reflected in the stream CSP. This is done by connecting the stream CSP to the CSP Genesys-Pro builds for the instruction. Figure 5 shows the combined CSP for our collision when `lmw` is the mnemonic selected for the first instruction.

After the generation of the instruction is completed, relevant values such as the actual address and length of the memory operand are copied from the instruction CSP to the stream CSP. Then, a new fixed-point is reached to propagate these values between the instructions and the generation of the next instruction can begin following the same procedure. This process continues until all the instructions are generated.

The process described in this section is a somewhat simplified version of the actual process we implemented. First, not all the variables that are used in the stream CSP are described here. For example, the memory operand has variables for real and effective addresses, not just a single address. In addition, the actual process is capable of handling multithreaded scenarios. This requires an additional thread variable for each instruction and more complex ordering constraints that include threads. All these are omitted to ensure a clear and simple description. Finally, the actual process is capable of handling larger and more complex scenarios than the simple collision

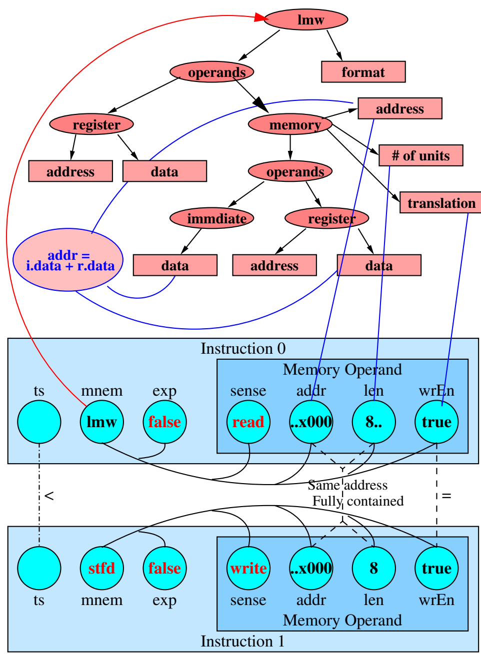

Fig. 5. Connecting the stream and an individual instruction CSPs

used in the paper. In fact, the stream CSP for real scenarios containing tens instructions is smaller and simpler to solve than an average single instructions CSP that are solved during the generation process.

IV. Experimental Results

We conducted a set of experiments to validate the ability of our stream generation scheme to generate large scenarios and improve the scenario-related generation capabilities of Genesys-Pro. We carried out the experiments using Genesys-Pro on the latest IBM Power processor.

In the first experiment we compared the generation capabilities of our approach to the current capabilities of Genesys-Pro for generating simple RAW collisions on the LMQ presented in Section II. We used the exact same scenario presented in Section II, except we removed the coverage constraint on the second instruction and allowed it to be any store instruction. To simplify the experiment and allow compact representation of its results, we focused on 10 load instructions and 10 store instructions. Specifically, the load instructions we used were `lbz, lhz, lwz, ld, lq, lfs, lfd, lfdp`, which load a fixed number of bytes from memory ranging from 1 to 16, and `lmw, lsdi`, which load a varying number of bytes from memory. For store instructions, we used the stores corresponding to these loads (i.e., `stb, sth`, etc.). In the experiment we compared the following three settings:

- The new generation scheme presented in the previous sections.
- Basic Genesys-Pro test template. We used a single test template that specifies the scenario requirements as is. That is, the test template specifies that the first instruction is a load (from the set of loads in the experiment) and the second instruction is a store whose memory operand has the same address as the memory operand of the load. The template also specified that the two instructions do not take exceptions. All the constraints that are inferred from these constraints, such as the load needing to access writable memory or that certain load store pairs are infeasible because of their access length, are not included.
- Expert Genesys-Pro test templates. Here, the test templates include the inferred constraints. Because each store instruction projects different constraints on the loads, this setting required the development of several test templates, each addressing a different set of store instructions.

TABLE I
Effort and success rate for the three generation settings

Setting	Stream	Basic	Expert
Test templates	1	1	7
Effort	10 min	10 min	3 hours
Success Rate	97%	59%	92%

Table I compares the effort needed in each of the settings and the generation success of each of them. For each setting, it shows the number of test templates needed to implement the required scenario, the effort needed to implement the test template(s), and the generation success rate. The table shows that even for a simple scenario with two instructions, the basic test template has a much lower success rate than the stream generation approach because it does not include the propagation of the scenario requirements between the instructions. The expert setting is capable of achieving a similar success rate, which is closer to the success rate of the stream generation approach. However, even for a simple scenario with a limited set of possible instructions, it requires many test templates to achieve this success rate and much more effort. This effort is required to manually propagate and infer the constraints between the instructions and iteratively test and fix the test templates. All this is not necessary in the stream generation approach, where the inference is done automatically. The success rate of the expert setting is still lower than the stream setting because of the difficulty in controlling the length of the memory access in the instruction with varying length operand (`lmw, lsdi, stmw, stsdi`) and adapting it to the length of the other instruction in the stream.

Figure 6 compares the frequency of each of the load and store instructions in each of the settings. The figure shows the results only for successfully generated tests. We observe that the distribution of the load instructions is almost uniform for the stream generation approach. This happens because the mnemonic of the load instruction is selected first. Moreover, because each load instruction has at least one store instruction it can match, the selection is uniform. Once a load instruction is selected, not all stores are possible; therefore, while the generator selects uniformly among possible instructions, the resulting distribution shows much higher preference for instructions that can access a small number of bytes. We are working on improving our generation scheme so the

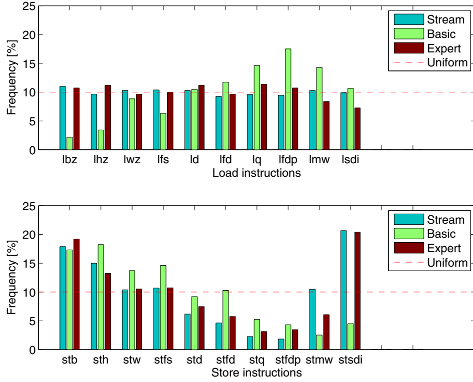

Fig. 6. Frequency of load and store instructions

distribution of instructions is more uniform in the combined instruction space. With the basic Genesys-Pro test templates, the distribution of the loads favors instructions with long access because they have a better chance of matching a random store and similarly short stores are more frequent because they have better chance of fitting within a load. The expert templates results are similar to the stream approach except for the `lmw` and `lsdi` instructions that have lower frequency. This lower frequency is caused by the difficulty to match these instructions to stores discussed above.

In the second experiment we tested how the new generation scheme scales up to more complex scenarios. We increased the number of instructions participating in the scenario, along with the number of streams and ordering constraints between the instructions. Figure 7 shows how the increase in scenario size and complexity affects the generation success rate and average generation time of the instructions. In the figure, the average generation time per instruction is the average time needed to generate a scenario instruction. This is the generation time of a test (excluding initialization and post-processing time) divided by the number of instructions in the scenario. The generation success rate is similar to the one defined for Table I. The figure shows that increasing the scenario size does have a negative effect on both the generation time and success rate, but these effects are not big. Specifically, the drop in the success rate is caused by "normal" Genesys-pro generation failure and the fact that the probability of a single generation failure increases with the size of the stream. The increase in instruction generation time is caused primarily by additional instructions that Genesys-pro inserts into the test to better manage resources, such as registers [12].

V. Conclusions

We described a new approach for generating instruction streams that implement given scenarios. This approach is based on interleaving between the progress in the solution of an abstract problem that captures the essence of the scenario and generation of concrete instructions. The proposed approach provides a means for simple description of the scenario while ensuring a high generation success rate. Experimental results show that our approach achieves much higher generation success rate even for simple scenarios over basic Genesys-Pro test templates and significant reduction in effort to create test template over expert created templates.

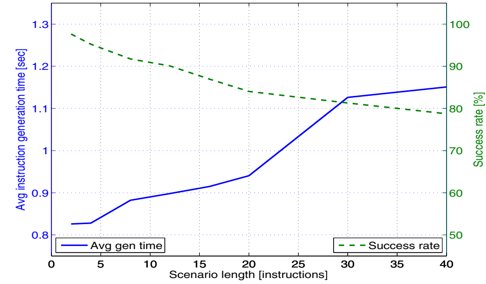

Fig. 7. Generation time and success rate as a function of scenario complexity

This work is part of a greater framework whose goal is to raise the level of abstraction in processor-level stimuli generation and bring it closer to the verification plan. Other aspects of this framework include: the definition of a language to describe scenarios that will serve as a basis for the test template language, the inclusion of microarchitectural knowledge in the generator, and the development of stream-level testing knowledge.

References

[1] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "POWER7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, no. 2, pp. 7–15, 2010.
[2] C. Ioannides, G. Barrett, and K. Eder, "Feedback-based coverage directed test generation: An industrial evaluation," in *Proceedings of the 6th Haifa Verification Conference*, ser. LNCS 6504. Springer-Verlag, 2010, pp. 112–128.
[3] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM Systems Journal*, vol. 41, no. 3, pp. 386–402, 2002.
[4] B. Gutkovich and A. Moss, "CP with architectural state lookup for functional test generation," in *Proceedings of the High-Level Design Validation and Test Workshop*, 2006, pp. 111–118.
[5] P. Mishra and N. Dutt, "Automatic functional test program generation for pipelined processors using model checking," in *Seventh Annual IEEE International Workshop on High-Level Design Validation and Test*, October 2002, pp. 99–103.
[6] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Design and Test of Computers*, vol. 21, no. 2, pp. 84–93, 2004.
[7] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-Gen: A random test-case generator for systems and SoCs," in *IEEE International High Level Design Validation and Test Workshop*, October 2002, pp. 145–150.
[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, September 2003.
[9] R. I. Brafman and C. Domshlak, "Factored planning: How, when, and when not," in *AAAI*, 2006.
[10] Y. Chen, Y. Xu, and G. Yao, "Stratified planning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2009, pp. 1665–1670.
[11] A. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.
[12] A. Adir, E. Marcus, M. Rimon, and A. Voskoboynik, "Improving test quality through resource reallocation," in *Proceedings of the High-Level Design Validation and Test Workshop*, 2001, pp. 64–69.