Piparazzi: A Test Program Generator for Micro-architecture Flow Verification

Allon Adir **Eyal Bin** **Ofer Peled** **Avi Ziv**
IBM Research Lab in Haifa, Israel
email: {adir, bin, oferp, aziv}@il.ibm.com

Abstract

Because of their complexity, modern microprocessors need new tools that generate tests for micro-architectural events. Piparazzi is a test generator, developed at IBM, that generates (architectural) test programs for micro-architectural events. Piparazzi uses a declarative model of the micro-architecture and the user's definition of the required event to create an instance of a Constraint Satisfaction Problem (CSP). It then uses a dedicated CSP solver to generate a test program that covers the specific event. We show how Piparazzi yields significant improvements in covering micro-architectural events, by describing its technology and by exhibiting experimental results. Piparazzi has already been successful in finding both functional and performance bugs that could only be discovered using an exact micro-architectural model of the processor.

Introduction

Functional verification is the bottleneck of the hardware design cycle [4], especially for large and complex designs. The investment in expert time and computer resources for functional verification is huge, and so is the cost of delivering faulty products [3]. In current industrial practice, most of the verification is done by generating a massive number of tests using architectural random test generators and by simulating the tests on the design [8, 14].

Modern microprocessors have several micro-architectural mechanisms that improve performance, but increase the complexity of the design, thereby increasing the risk of bugs. Examples of such mechanisms include multiple execution units, out-of-order execution, pipelining, and caching [12]. As a result of the complexity of these mechanisms, and the interactions between them, much of the verification plan for microprocessors consists of micro-architectural events and scenarios that the verification team would like to test during verification. There is a wide gap between the language describing micro-architectural events in the verification plan and the language used to direct architectural random test program generators. Therefore, architectural test generators cannot be used to cover these events directly.

Recently, several tools and methodologies for the verification of micro-architecture mechanisms have been described (e.g., [6, 9, 10, 13]). Most of these are based on formal methods [7] involving the creation of an abstract *Finite State Machine* (FSM) that describes the operation of the processor and translates the abstract tests to concrete architectural tests. These techniques suffer from two main difficulties: state explosion and the translation of the abstract tests to concrete architectural tests.

In this paper, we present a new approach to test generation for the verification of micro-architecture flow at the processor level, implemented in a tool named Piparazzi. Piparazzi, developed at IBM, is designed to find both architectural and micro-architectural (performance) bugs that cannot be covered by architectural test generators. As shown in Figure 1, Piparazzi's main inputs are a model of the micro-architecture and the user's specification of a required event. Piparazzi uses these to construct a CSP problem, and then solves the problem in order to produce a test program. In addition, Piparazzi produces a file detailing the micro-architectural behavior expected from the test program (i.e., the used resources and timing of events for the test instructions).

This *model-based* scheme [2], in which a generator is partitioned into a generic system-independent engine and a model that describes the verified system, has a number of significant advantages. First, generation for new processors becomes an easier task because the same generation engine can still be used, and only the model is replaced. Second, there is a structured, well-defined way to integrate new knowledge about the verified processor into the tool.

Piparazzi's input language lets the user specify micro-architectural events. Examples of such events are when two instructions at specific pipe stages generate exceptions in consecutive cycles, or when dispatching of instructions is blocked because the register file of the floating-point registers is full. Piparazzi then uses a model of the processor micro-architecture to generate an architectural (assembly) test program that causes the event requested by the user.

The model of the micro-architecture is based on a set of predefined and customizable building blocks, some describing hardware components (e.g., out-of-order queues, pipeline stages) and some describing mechanisms (e.g., flushes). In the modeling process, the modeler selects the proper building blocks that exist in the processor, customizes them, and connects them together, to create the

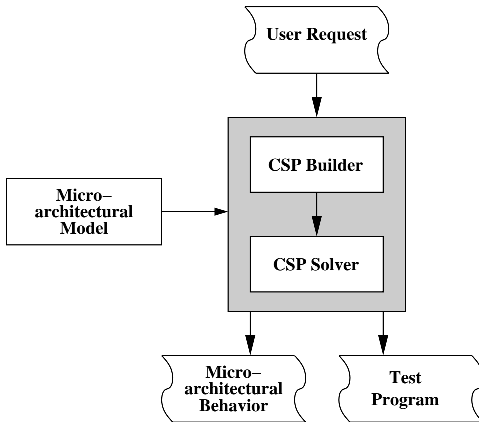

Figure 1. Piparazzi's architecture

micro-architectural model.

This modeling technique allows us generate the test program by converting the model into a *Constraint Satisfaction Problem* (CSP) and using a dedicated CSP solver [5] to construct an actual test.

Piparazzi is currently used in the verification of several processors being developed at IBM laboratories. Its flexible modeling language allows it to be used in the verification of totally different processors that implement a wide range of architectures, including processors of the PowerPC family (RISC architecture) and z-series (CISC architecture). Significant portions of the test plan rely on Piparazzi for coverage and it has already been used successfully to exhaustively cover micro-architectural models.

Motivation and Requirements

Advances in silicon technologies and the aggressive design styles used in microprocessors allow design teams to include many mechanisms that improve processor performance. The verification process should thoroughly test the operation of these mechanisms on their own, and their operation in conjunction with other mechanisms, since many of the more hard-to-find bugs involve multiple mechanisms. Therefore, the verification plan of such microprocessors often includes a long list of micro-architectural events.

Figure 2 presents an example of a typical micro-architectural event. A super-scalar microprocessor contains several floating-point execution units. Each execution unit is comprised of an out-of-order issue queue and a pipeline with six stages. The instruction type and its data can cause instructions to remain in certain pipeline stages for more than one cycle. In this case, the pipeline stage raises the corresponding stall signal (s_0 - s_5) and stalls the operation of previous stages in the pipe.

The requested event is as follows: An instruction, I_1, enters stage 1 of the pipe and raises s_1 because it needs to stay in this stage for several cycles. One cycle later, another instruction, I_3, enters stage 3 of the pipe and raises s_3. While

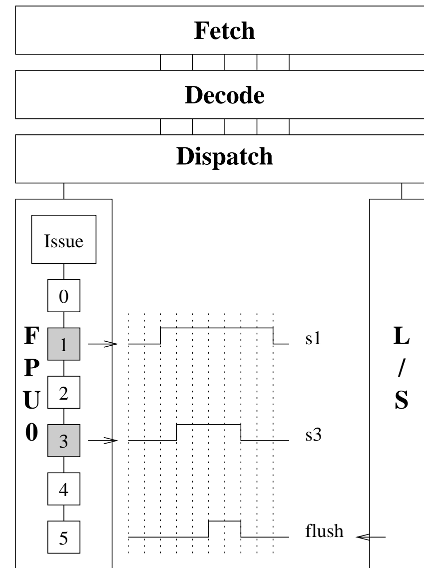

Figure 2. Micro-architectural event example

I_3 is executing in stage 3, a misalign exception generated at the L/S unit flushes I_3, but does not affect I_1. This can happen if I_1 and I_3 were issued out-of-order.

Generating a test that causes the above event is difficult. The main vehicles for functional verification of processors, namely architectural test generators, do not have enough control to provide the delicate timing required to generate such events. A labor intensive alternative is to manually analyze the event and create a corresponding test through iterative improvements and modifications.

The event above may be part of a large coverage model (i.e., a family of similar events) in the verification plan. For example, we may want to see the same event, with exceptions, coming from different units. Because the verification plan is full of many such families of events, manually writing tests that cover them, or only those that are not covered by architectural test generators, is impractical or even impossible due to the complexity of the events.

There is a need for an automatic test program generator that receives as its input a specification for *a micro-architectural event*, or family of events, and produces architectural test programs that cover the events. The input language should be comprehensive enough to specify all interesting micro-architectural events. In addition, a model-based test generator [2] should provide an infrastructure for modeling various micro-architectures. The generator should generate random tests since the state space of the tested design is huge and it is impossible to specify and generate tests that cover all of it.

Piparazzi Architecture

In the following section, we provide a more detailed description of the main components of Piparazzi, as shown in Figure 1.

Micro-architecture Model

The model of the micro-architecture lays the foundation upon which the generation engine builds the tests, based on the user requests. The model of the micro-architecture needs to be detailed and precise enough to accurately generate tests that achieve the user requested events, which involve delicate timing. On the other hand, the model must also be simple enough to allow tests to be created within a reasonable time-frame and to adapt to on-going changes in the micro-architecture.

To ease the task of modeling micro-architectures, Piparazzi provides a modeling infrastructure, realized as a set of building blocks. This allows modelers to create the micro-architectural model by customizing these building blocks and connecting them together in the proper way.

The model of the micro-architecture consists of a set of building blocks. The various types of building blocks describe the structure of the micro-architecture and the flow of instructions through it. Each building block is associated with several fixed parameters that are assigned during modeling and determine its nature and basic behavior. For example, a cache contains parameters for its size, associativity, and replacement policy; a pipeline stage contains information about instructions that are stalled in it; and a queue contains parameters about its size and issuing policy.

In addition to these fixed parameters, building blocks contain variables whose values, selected during generation, determine the behavior of the object during the execution of the generated test. For example, an instruction building block contains variables for its opcode, issue time, and the pipeline in which it is executed.

The behavior of the micro-architecture and the flow of instructions is defined by specifying constraints over variables of the same or of different building blocks in the model. For example, a constraint over instruction opcodes and pipelines indicates the pipelines in which the instruction can be executed.

The micro-architecture is described in terms of building blocks derived from three basic types: instructions, hardware components, and micro-architectural mechanisms. These types represent different aspects of the micro-architecture.

Instructions, the first building block type, are the main actors in the generation, thus they contain most of the variables in the model. These include variables that describe architectural properties of the instruction (e.g., its opcode, operands, and data), variables that describe the instruction flow (e.g., the execution pipeline), and timing variables (e.g., entry time to pipeline stages, flush time, etc.).

For every instruction in the test, Piparazzi selects values for these variables in a way that corresponds to the architectural and micro-architectural constraints. These constraints naturally depend on the type of the particular instruction. For this purpose, the model also describes the processor's instruction set. Every instruction in the set is modeled in terms of parameters that define its architectural aspects (e.g., opcode mnemonic, operands, and semantics) and its micro-architectural behavior (e.g., nominal execution time, execution pipelines, and dispatching rules).

The second building block type relates to the hardware components in the processor, such as pipelines and pipeline stages, queues, register files, and caches. Because Piparazzi uses a flow approach for the micro-architecture, most of the modeling of the hardware components is done by specifying constraints on variables of instructions that use these components. For example, the behavior of an out-of-order queue is described as constraints over the time instructions are inserted into it, the time they are ready to leave, and their departure cycle.

The third type contains micro-architectural mechanisms that are used by the processors, such as flushes. These building blocks usually consist of two parts: what activates the mechanism and how it affects the behavior of the processor. This modeling allows users to express their requests directly, in terms that relate to these mechanisms.

User Requests

The user request file defines the event, or family of events, that Piparazzi is requested to generate. The basic elements used to build the request file are *rules*. Rules are defined as constraints over one or more variables in the micro-architectural model. For example, the rule `I[1].PIPELINE = I[3].PIPELINE` states that these two instructions in the event are executed in the same pipeline. Note that instruction indexes 1 and 3 do not necessarily correspond to the program ordering of the instructions, but only serve to identify them. The language used in the request file supports arithmetic operators (e.g., +, -, *), relational operators (e.g., = and >), first order logic operators, and others.

Several rules can be combined together to describe *events*. Events are the basic generation unit for Piparazzi. Each test generated by Piparazzi attempts to cover a specific event. In addition, the request language contains other constructs, such as loops, if statements, and macros.

Figure 3a shows the definition of a parametric event that is used to define a coverage model, based on the event described in Figure 2. A coverage model can be defined as a list of events or as the cross-product of the possible values of a set of parameters used in the specification of a parametric event. Figure 3b shows the definition of a coverage model that covers events defined by the parameterized macro on the left. The coverage model defines events for all possible cases for the execution unit (`FPU0 and FPU1`), type of flush, and delay between stalls. A separate test is then generated for each event defined in the request file.

Generation Engine

The role of the generation engine is to read the model of the micro-architecture and the user-requested event, and generate a corresponding test program. The generation engine performs this task in three main stages: (1) read the

```
MacroDef fp_event (pipe, flush, delay)
    I[3].PIPELINE = I[1].PIPELINE = pipe                  // I1 and I3 go to the same pipeline pipe
    I[1].stage[1].SELF_STALL > 0                          // I1 is stalled in stage 1
    I[3].stage[3].SELF_STALL > 0                          // I3 is stalled in stage 3
    I[1].stage[1].ENTRY = I[3].stage[3].ENTRY + delay     // I1 starts stalling delay cycles after I3
    I[2].FLUSH = flush                                    // I2 generates a flush
    I[2].FLUSH_TIME = I[3].stage[3].ENTRY + 1             // I2 generates a flush one cycle after I3 stalls
    I[3].DESTINY = FLUSHED                                // I3 is flushed
    I[1].DESTINY = FINISHED                               // I1 completes normally (not flushed)
```

(a) Definition of a parametric event

```
foreach pipe in (FPU_0, FPU_1)
  foreach flush in (flush_types)
    foreach delay in (0, 1, 2)
      event
        MacroCall fp_event (pipe, flush, delay)
      end event
    end
  end
end
```

(b) Definition of a coverage model

Figure 3. Definition of a task and a coverage model

micro-architecture model and user-requested event and construct a CSP; (2) solve the CSP, and (3) produce the test program.

The generation engine first reads the model specification and converts it to a CSP. A CSP consists of a finite set of *variables*, each 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 CSP is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. Because the constraints of the user requested event are satisfied, the generated test program causes the requested event.

The CSP includes an (initially identical) instruction object for every instruction of the test. Next, the engine reads the user request file and converts the events defined in it into constraints that are added to the CSP.

Then, a dedicated CSP solver is activated and solves the CSP. That is, it assigns single values to all of the relevant variables in the CSP, such that all the constraints in the CSP are satisfied.

The solution scheme used in Piparazzi is based on the *Maintaining Arc Consistency* (MAC) [11] family of CSP solution algorithms. MAC algorithms use a procedure for achieving arc-conisistency as a filtering component. 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 that satisfies the constraint. A constraint network is said to be arc-consistent if all of its constraints are locally consistent.

We made some refinements and variations to this CSP solving technique to fit the specific characteristics of Piparazzi's CSP. Some of these characteristics are common to many CSPs that represent test programs such as the generation of random, well-distributed solutions over the solution space [8], and huge variable domains [5]. In addition, Piparazzi exploits symmetries that are typical to its CSP networks to optimize the solving process.

In the third stage, after the CSP solver has solved the CSP and assigned a single value to each variable, the generation engine translates this solution into a test program. This translation is done using the architectural variables of the instruction objects. These variables determine all the information that is needed in the test, including the opcodes of the instructions, their operands, and the data of the operands.

Usage

Piparazzi is currently used in the verification of various processors at several IBM development laboratories. These processors implement various architectures, such as PowerPC (RISC architecture) and z-series (CISC architecture). The processors modeled in Piparazzi use a wide range of micro-architectures through many different micro-architectural mechanisms.

The role of Piparazzi in the verification environment is to fill the gap in areas where the architectural level test generators are weaker. This includes scenarios that target micro-architectural resources and mechanisms, and especially events that require specific intricate timing. It is important for the processor verification plan to include coverage models of this kind and Piparazzi's language (specifically designed to express such events) can be used as a more formal way to define them. In many cases, models can be defined in a generalized and *processor independent* way and can be accumulated and reused.

Experience has shown (as demonstrated in the following section) that many coverage models can best be covered using a "hybrid" approach, which employs both Piparazzi and an architectural level test generator. The architectural level test generator is used first to quickly cover most of the tasks. Then Piparazzi is used to complete the coverage of the more difficult tasks. Piparazzi can either generate a test or declare that the task is unreachable by the design and the coverage model needs to be adjusted.

Another important task that Piparazzi performs is the comparison of the actual micro-architectural behavior reported by the design simulator, with the expected behavior as modeled and reported by Piparazzi (Figure 1). This approach has already been productive in finding several performance-related bugs, which could only be discovered using an exact micro-architectural model of the processor.

There are some limitations to the Piparazzi approach. First, the exponential complexity of the CSP solution poses an actual limit of several dozens of instructions per test in Piparazzi. This means that Piparazzi cannot directly generate tests that involve very long scenarios, such as filling large buffers. In addition, the complexity of the CSP imposed by the Piparazzi model leads to a longer generation time compared to architectural test generators. In some cases, where Piparazzi cannot automatically solve the CSP problem within a reasonable time, the user can add "hints"

to the definition of the coverage model in order to reduce the state space that needs to be explored.

Experimental Results

We carried out an experiment to demonstrate the usefulness of Piparazzi in the implementation of the micro-architectural verification plan. The experiment compares the coverage progress of one coverage model using two approaches: one that uses an architectural random test program generator; and another that combines the architectural test generators with Piparazzi (the "hybrid" approach.)

The coverage model used in the experiment contains all the possible "register forwarding" of general-purpose registers in a PowerPC processor. In register forwarding, the result of an instruction in one pipeline stage is forwarded to another instruction in another stage, without passing through the register file. The model requires the coverage of all possible combinations of pipeline stages for the source and target of the forwarding. In addition, the model requires one or two instruction "commits" to take place at exactly the cycle in which the forwarding occurs.

The architectural random test program generator we used in the experiment is Genesys-Pro [1], a state-of-the-art IBM test program generator, which employs architectural testing-knowledge to bias its random test generation. The tool was biased to generate test cases with many target-source dependencies between general-purpose registers. The fast generation rate of Genesys-Pro and its biasing capabilities enable the generation of a large number of tests that cover most of the events in a short time. On the other hand, Genesys-Pro does not have the precise knowledge about the micro-architecture of the processor. Therefore, it cannot directly cover the events in the coverage model and the probability of it covering some of the events is very low.

In the hybrid mode, we start to cover the model with an architectural test generator. When the coverage progress slows down, we activate Piparazzi with the exact specification of the uncovered tasks, and use it to generate test cases that cover these tasks (or to demonstrate that they cannot be reached.)

Figure 4 shows the coverage progress over time of both approaches. These results demonstrate that Genesys-Pro quickly covered 75% of the events in the model but then made very slow progress until reaching an 80% limit. The remaining tasks are either very hard to reach through randomly biased tests or possibly represent combinations that can never occur on the particular tested design. Piparazzi was activated after 75% of the model events were covered by Genesys-Pro. It continued to cover difficult tasks not covered by Genesys-Pro by directly generating random tests for the tasks. As the figure shows, Piparazzi was able to reach about 92% of the model's original set of tasks. During this time Piparazzi also showed that several tasks were unreachable. To complete the coverage of all the possible tasks, we began to add "hints" to the description of

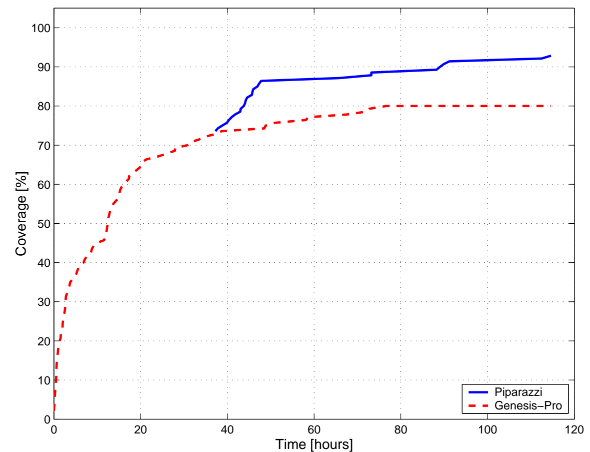

Figure 4. Coverage progress for Genesys-Pro and the hybrid approach

the events given to Piparazzi. Such hints, given in micro-architectural terms, reduce the state space searched by the CSP engine and thus increase the chance of finding a test program.

Conclusions

In this paper, we describe Piparazzi, a test program generator designed for the verification of micro-architectural mechanisms. Piparazzi consists of a declarative description of the micro-architecture, and a generation engine based on a dedicated CSP solver that converts users' requests for micro-architectural events into test programs. Piparazzi is used in the verification of several processors in IBM. Its ability to generate test programs that cover complex micro-architectural scenarios and to predict the precise micro-architectural behavior of these programs, led to the discovery of several performance bugs.

We continue to enhance the capabilities of Piparazzi and its integration in verification environments. We are working on ways to extend the modeling language with additional micro-architectural building blocks, such as branch prediction mechanisms, and simplify the modeling process. In addition, we are trying to improve the CSP-based generation engine with domain specific and general purpose techniques. Finally, we are looking at ways to integrate Piparazzi with a coverage analysis tool and architectural test generators.

References

[1] A. Adir, R. Emek, and E. Marcus. Adaptive test program generation: Planning for the unplanned. In *Proceedings of the High-Level Design Validation and Test Workshop*, pages 83–88, 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 *Proceedings of the 32nd Design Automation Conference*, pages 279–285, June 1995.

[3] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

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

[5] E. Bin, R. Emek, G. Shurek, and A. Ziv. What's between constraint satisfaction and random test program generation. *IBM Systems Jouranl*, 41(3):386–402, 2002.

[6] D. V. Campenhout, T. Mudge, and J. P. Hayes. High-level test generation for design verification of pipelined microprocessors. In *Proceedings of the 36th Design Automation Conference*, pages 185–188, June 1999.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT-Press, 1999.

[8] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proceedings of the 1999 Design, Automation and Test in Europe Conference (DATE)*, pages 434–441, March 1999.

[9] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test program generation for pipelined processors. In *Proceedings of the International Conference on Computer Aided Design*, pages 580–583, Novemnber 1994.

[10] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Proceedings of the 38th Design Automation Conference*, pages 816–821, 2001.

[11] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *A.I. Magazine*, 13(1):32–44, Spring 1992.

[12] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design : The Hardware/Software Interface*. Morgan Kaufmann, 1997.

[13] S. Ur and Y. Yadin. Micro-architecture coverage directed generation of test programs. In *Proceedings of the 36th Design Automation Conference*, pages 175–180, June 1999.

[14] J.-T. Yen and Q. R. Yin. Multiprocessing design verification methodology for Motorola MPC74XX PowerPC microprocessor. In *Proceedings of the 37th Design Automation Conference*, pages 718–723, June 2000.