Directed Micro-architectural Test Generation for an Industrial Processor: A Case Study

Heon-Mo Koo Prabhat Mishra
Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.
{*hkoo, prabhat*}*@cise.ufl.edu*

Jayanta Bhadra Magdy Abadir
Freescale Semiconductor Inc.
7700 West Parmer Lane Austin, TX 78727, USA
{*Jayanta.Bhadra, M.Abadir*}*@freescale.com*

Abstract

Simulation-based validation of the current industrial processors typically use huge number of test programs generated at instruction set architecture (ISA) level. However, architectural test generation techniques have limitations in terms of exercising intricate micro-architectural artifacts. Therefore, it is necessary to use micro-architectural details during test generation. Furthermore, there is a lack of automated techniques for directed test generation targeting micro-architectural faults. To address these challenges, we present a directed test generation technique at micro-architectural level for functional validation of microprocessors. A processor model is described in a temporal specification language at micro-architecture level. The desired behaviors of micro-architecture mechanisms are expressed as temporal logic properties. We use decompositional model checking for systematic test generation. Our experiments using a processor based on the Power Architecture^{TM} *Technology*[1] *shows very promising results in terms of test generation time as well as test program length.*

1 Introduction

Performance improvement of modern processors is accompanied with high design complexity by adopting complicated micro-architectural mechanisms such as deeply pipelined superscalar, dynamic scheduling, and dynamic speculation. Since verification complexity is directly proportional to the design complexity, considerable amount of time and resources are spent on design verification.

In the current industrial practice [11], random and biased-random test generation techniques at architecture (ISA) level are most widely used for simulation-based validation to uncover errors early in the design cycle as well as to perform simulation for the entire processor design. However, as demonstrated in Section 4, architectural test generation techniques have difficulty in activating micro-architectural target artifacts and pipeline functionalities since it is not possible to generate information regarding pipeline interactions or timing details using input ISA specification. For example, it is very hard to generate an architectural test program for micro-architectural design bugs such as a pipeline interaction error (e.g., “decode stage is not stalled even if Completion Queue is full”), or a performance error (e.g., “data dependency is not resolved by forwarding path even if operand is available”). Therefore, it is necessary to use micro-architectural details during test generation.

Compared to the random or biased-random tests, the directed tests can reduce overall validation effort since shorter tests can obtain the same coverage goal. However, there is a lack of automated techniques for directed test generation targeting micro-architectural faults. As a result, directed tests are hand-written by experts. Due to manual development this process can be error prone.

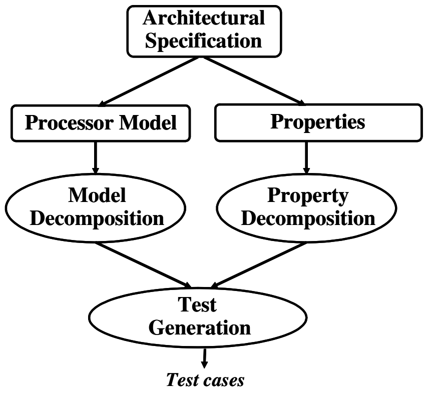

Figure 1. Test Program Generation Methodology

To address these challenges, we present a directed test generation technique at micro-architectural level for functional validation of microprocessors. Figure 1 shows our directed test generation methodology. The input specification contains both the structure (micro-architectural details) and the behavior (instruction-set) of the processor. The micro-architectural features in the processor model include pipelined and clock-accurate behaviors that enable micro-architectural test generation. Properties can be automatically generated from the input specification based on a functional fault model such as pipeline (graph) coverage [9]. Additional properties can be added based on interesting scenarios such as combined pipeline stage rules and corner cases. These properties are described in temporal

[1]The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org

logic. For automatic test generation, we use decompositional model checking where the processor model as well as the properties are decomposed and the model checking is applied on smaller partitions of the design using decomposed properties. We introduce the notion of time steps to enable decomposition of the properties into smaller ones based on their clock cycles. We have developed an efficient algorithm to merge the partial counterexamples generated by the decomposed properties to produce the global counterexample corresponding to the original property. We applied this methodology on a processor based on the Power Architecture Technology to demonstrate the usefulness of our approach.

The main contribution of this work is to establish a framework for a directed and automated micro-architectural test generation technique for validation of modern industrial processors. Since the proposed method is generic, its framework can be used for validation of any other real processors. The rest of the paper is organized as follows. Section 2 presents related work addressing test generation in the context of micro-architectural validation of pipelined processors. Section 3 describes our test generation methodology. Section 4 presents a case study and Section 5 concludes the paper.

2 Related Work

Several methodologies have been developed for validation of pipelined processors using finite state machine (FSM) models [1, 5, 6, 10] where FSM coverage is used to generate test programs. In modern processor designs, complicated micro-architectural mechanisms include interactions among many pipeline stages and buffers that lead the FSM-based approaches to the state-space explosion problem. To alleviate the state explosion, Utamaphethai et al. [12] have presented a FSM model partitioning technique based on micro-architectural pipeline storage buffers whose entries store data and status. However, it suffers from targeting complete micro-architectural features because test programs are generated by design errors from each buffer, not for combined buffers.

An alternative formal method, model checking [2], has been successfully used in software and hardware validation as a test generation engine [3, 9]. The negated version of a desired property along with the processor model is applied to the model checker. The model checker automatically produce a counterexample that contains a sequence of instructions (a test program) from an initial state to a failure state. However, this naive approach is unsuitable for a real processor model due to the state explosion problem during model checking.

Koo and Mishra [7, 8] have proposed a processor/property decomposition technique to reduce the search space during counterexample generation as well as an algorithm for merging the partial counterexamples to generate architectural test programs. Their test generation technique is built on a relatively simple MIPS processor [4] with no renaming buffer, reordering buffer, or reservation station. They use pipeline path-level model partitioning to generate a test program for data forwarding, but it causes deprivation of memory during model checking when applying to the industrial processors due to high complexity of even single pipeline path. In addition, they mainly focus on the data path rather than the control path. While a data (opcode and operands) is located in a single pipeline stage, control signals (functional unit status and buffer status) may spread across multiple pipeline stages and buffers which make model partitioning and counterexample merging more difficult. Therefore, it is necessary to improve the decomposition and merging algorithms for application to the complex industrial processors.

3 Directed Micro-architectural Test Generation

Today's test generation techniques and verification methods are very efficient to find bugs at the unit level. Hard-to-find bugs arise often from the interactions among many pipeline stages and buffers of a modern processor design. We primarily focus on such micro-architectural interface among functional units in a pipelined processor.

```
Algorithm 1: Test Generation
Inputs: i) Processor model $M$
        ii) Set of interactions $S$ from fault model and corner cases
Outputs: Test programs
Begin
   TestPrograms = $\phi$
   for each interaction $S_i$ in the set $S$
      $P_i$ = CreateProperty($S_i$)
      $\overline{P_i}$ = Negate($P_i$)
      $test_i$ = DecompositionalMC($M$, $\overline{P_i}$)
      TestPrograms = TestPrograms $\cup$ $test_i$
   endfor
   return TestPrograms
End
```

Algorithm 1 describes our test generation procedure. This algorithm takes the processor model M and desired pipeline interactions S as inputs and generates test programs. The processor model is described in a temporal specification language such as SMV [13]. For each interaction S_i, the algorithm produces one test program $test_i$. S_i is composed of a set of instruction and control functionalities at pipeline units and their relations and it is converted to a temporal logic property P_i. The negation of P_i is an interaction fault. The processor model M and the fault $\overline{P_i}$ are applied to decompositional model checking framework to generate a test program. The algorithm iterates over all the interaction faults in the fault model and corner cases.

3.1 Micro-architectural Modeling

Figure 2 shows a functional block diagram of the four-wide superscalar e500 processor that is based on the Power Architecture Technology [14] with the seven pipeline stages. Pipeline buffers are highlighted in grey. We have developed a processor model based on the micro-architectural structure, the instruction behavior, and the rules in each pipeline stage that determine when instructions can move to the next stage and when

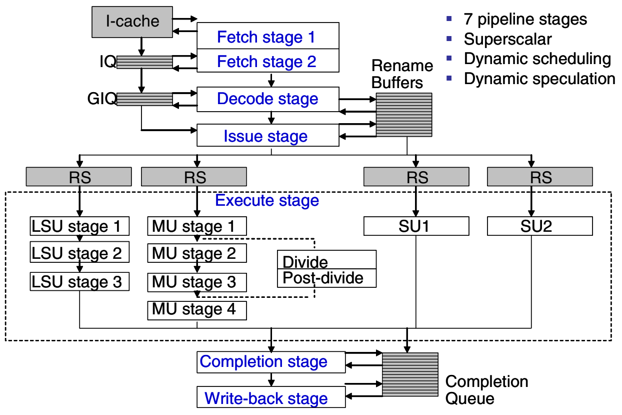

Figure 2. Instruction Pipeline Flow of e500 processor that is based on the Power Architecture Technology

they cannot. The micro-architectural features in the processor model include pipelined and clock-accurate behaviors such as multiple issue for instruction parallelism, out-of-order execution and in-order-completion for dynamic scheduling, register renaming for removing false data dependency, reservation stations for avoiding stalls at Fetch and Decode pipeline stages, and data forwarding for early resolution of RAW data dependency. By representing them in a model checking language, we can achieve the automatic test generation goal.

In order to use model checking as a test generator, the processor model needs to be verified beforehand. Since it is infeasible to verify the entire model as a single unit due to the state explosion during model checking, we have partitioned the entire processor model into multiple modules based on the functional units shown as rectangles in Figure 2. Each partitioned module has been verified using the requirements and rules described in the specification. For verification of module interface, we integrated neighboring modules and verified their interface. These modules are basic units in processor model decomposition for test generation.

3.2 Property Generation and Decomposition

We generate a property for each pipeline interaction from the specification. Since interactions at a given cycle are semantically explicit and our processor model is organized in structure-oriented modules, the interactions can be converted into properties. The generated properties are expressed in LTL (Linear Temporal Logic) [2]. Each property consists of temporal operators (G, F, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$, and $\rightarrow$). Most pipeline interactions can be converted in the form of a property $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities p_i over n modules using logical *AND* operator. The atomic proposition p_i is a functional activity at a module i such as operation execution, stall, exception or NOP. The property is true if $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ becomes true at any time step.

Since we are interested in counterexample generation, we need to generate the negation of the property first. The negation of the properties can be expressed as:

$$\neg X(p) = X(\neg p) \qquad \neg G(p) = F(\neg p)$$
$$\neg F(p) = G(\neg p) \qquad \neg pRq = \neg pU\neg q$$

For example, the negation of the interaction property is $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ that becomes true if any of p_1, p_2, $\ldots$, *or* p_n is not true over all time steps. In the remainder of this section, we describe how to decompose these properties (already negated) for efficient model checking. There are various combinations of temporal operators and Boolean connectives where decompositions are not possible e.g., $F(p \wedge q) \neq F(p) \wedge F(q)$ and $G(p \vee q) \neq G(p) \vee G(q)$. In certain situations, such as pUq, $F(p \rightarrow F(q))$, or $F(p \rightarrow G(q))$, decompositions are not beneficial compared to traditional model checking. The following combinations allow simple property decompositions.

$$G(p \wedge q) = G(p) \wedge G(q) \qquad F(p \vee q) = F(p) \vee F(q)$$
$$X(p \vee q) = X(p) \vee X(q) \qquad X(p \wedge q) = X(p) \wedge X(q)$$

Introducing the notion of clock (time step) in the property allows more decompositions for counterexample generation as shown below[2]. Note that the left and right hand side of the decomposition are not logically equivalent but they produce functionally equivalent counterexamples.

$$G((clk \neq t_s) \vee (p \vee q)) \approx G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$$

Although we only use a few decomposition scenarios, it is important to note that these scenarios are sufficient for generating the properties where interactions are considered. In addition to these interaction properties, we created many micro-architectural properties based on real experiences of industrial designers.

3.3 Test Generation using Model Checking

The basic idea of DecompositionalMC( ) in Algorithm 1 is to apply the decomposed properties (sub-properties) to appropriate modules and compose their responses to construct the final test program. Model checker is used to generate partial counterexamples for the partitioned modules. Integration of these partial counterexamples is a challenge due to the fact that the relationships among decomposed modules and sub-properties are not preserved at whole design level in general. We propose clock-based integration of partial counterexamples.

For example, if two sub-properties are applied at the same clock cycle ($clk = t_s$) to two modules sharing a parent module, then two counterexamples are generated and merged into the output property of the parent module for generating the counterexamples at the previous clock cycle ($clk = t_s - 1$). In Figure 2, four reservation station (RS) modules share the parent module Issue. Counterexamples generated from multiple RS at the cycle k are merged for creating the output property of Issue stage. The negated version of this property is applied to

[2]The clk variable is used to count time steps, and t_s is a specific time step.

Table 1. Test Cases and Code Length

	Test Cases	Test Code Length
1	Instruction dual issue	15
2	Renaming *src1* operand	12
3	Read operand from forwarding path (RAW)	9
4	Reservation station reads operand from forwarding path (RAW)	7
5	Read operand from renaming reg. (RAW)	10
6	Read operand from GPR (RAW)	11

the model checker along with Issue module to generate a counterexample at the cycle $k-1$ that is used to produce the output properties of Decode, GIQ, and Rename buffer. Merging partial counterexamples continues until we obtain the primary input assignments for all the sub-properties. These assignments contain fetched instruction data from I-cache and they are converted into assembly instruction sequences.

4 Experiments

We applied our methodology on a superscalar processor based on the Power Architecture Technology. We performed various test generation experiments for validating the pipeline interactions and corner cases. Table 1 shows a subset of the directed test cases that we generated and their corresponding length in terms of number of instruction sequences. For example, test programs for case 3 through 6 exercise operand read from four different resources as shown in Figure 3, which can be generated at micro-architecture level but very difficult at ISA level. In terms of efficiency, only several seconds were spent on test generation.

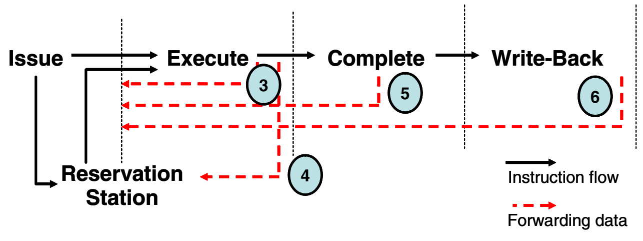

Figure 3. Four Different Data Forwarding Mechanisms

To validate these test cases, we converted the test programs into the input format of RTL simulation and monitored instructions in pipeline stages at every clock cycle during simulation to ensure that the generated test program activates the actual micro-architectural fault.

5 Conclusions

Architectural test generation techniques have limitations to achieve micro-architectural coverage goal. This paper presented a directed test generation technique based on decomposition of both processor model and properties for validation of performance as well as functionality of the modern microprocessors. Our experimental results using e500 processor that is based on the Power Architecture Technology demonstrate the efficiency of our method by generating complicated micro-architectural tests. Since the proposed technique is generic, its framework can be used for validation of industrial-strength processors. Our future work includes extension of the processor model for dynamic speculation and other features. Since the number of interactions (directed tests) can be still extremely large, we plan to develop a test compaction technique to reduce the number of test programs.

References

[1] D. Campenhout, T. Mudge, and J. Hayes. High-level test generation for design verification of pipelined microprocessors. *DAC*, pages 185–188, 1999.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[3] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 146–162, 1999.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[5] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test pattern generation for pipelined processors. *ICCAD*, pages 580–583, 1994.

[6] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. *DAC*, pages 816–821, 2001.

[7] H.-M. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. *DATE*, pages 1240–1245, 2006.

[8] H.-M. Koo and P. Mishra. Test generation using SAT-based bounded model checking for validation of pipelined processors. *GLSVLSI*, 2006.

[9] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. *DATE*, pages 182–187, 2004.

[10] J. Shen and J. A. Abraham. An RTL abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing: Theory and Applications*, 16(1-2):67–81, 2000.

[11] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, L. McConville, and T. Swanson. Verification of the cell broadband engine processor. *DAC*, pages 338–343, 2006.

[12] N. Utamaphethai, R. D. S. Blanton, and J. P. Shen. Effectiveness of microarchitecture test program generation. *IEEE Design & Test*, 17(4):38–49, 2000.

[13] www-cad.eecs.berkeley.edu/ kenmcmil/smv. *Cadence SMV*.

[14] www.freescale.com/files/32bit/doc/refmanual/e500CORERMAD.pdf. *Freescale PowerPc e500 core family reference manual*.