Automated Micro-architectural Test Generation for Validation of Modern Processors

Heon-Mo Koo
hkoo@cise.ufl.edu

Prabhat Mishra
prabhat@cise.ufl.edu

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.

***Abstract*— Design complexity of todays microprocessors is increasing at an alarming rate to cope up with the required performance improvement by adopting complicated micro-architectural features such as deep pipelines, dynamic scheduling, out-of-order and superscalar execution, and dynamic speculation. Since verification complexity is directly proportional to the design complexity, considerable amount of time and resources are spent on design validation. In the current industrial practice, billions of random test programs generated at instruction set architecture (ISA) level are used during simulation-based validation. 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.**

I. 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, functional validation has become one of the major bottlenecks in modern processor design: up to 70% of the design development time and resources are spent on functional verification. Simulation is the most widely used form of microprocessor validation. A major challenge in simulation-based validation is how to reduce the overall validation time and resources.

In the current industrial practice [1], [12], 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 IV, 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, Read After Write (RAW), 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 random or biased-random tests, the directed tests can reduce overall validation effort significantly 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 typically hand-written by experts, which is time consuming and error prone. As an automated approach, Model Checking can be used as a test generation engine. The negated version of a desired property and the processor model are applied to a model checker to produce a counterexample automatically that contains a sequence of instructions (a test program) from an initial state to a failure state. This test program can be used to exercise the desired property. However, this naive approach is unsuitable for a real processor model due to the state explosion problem during model checking. There is a need for automated and directed test generation techniques.

To address these challenges, we present a directed test generation technique at micro-architectural level for functional validation of microprocessors. Figure 1 shows the overall flow of the proposed test generation process. The input specification contains both the structure (micro-architectural details) and the behavior (instruction set) of the processor. From the specification, a micro-architectural model of the processor is formally specified in Model Checking language. Properties can be automatically generated from the specification based on a functional fault model such as pipeline interaction coverage. The processor model is decomposed into functional units and the properties are decomposed accordingly to alleviate the state explosion. Model checker generates partial counterexamples for individual units and they are merged together to form a counterexample for the entire processor. Once a test program is generated, RTL simulation is performed to determine if the test detects the fault. Another undetected fault is selected from

[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

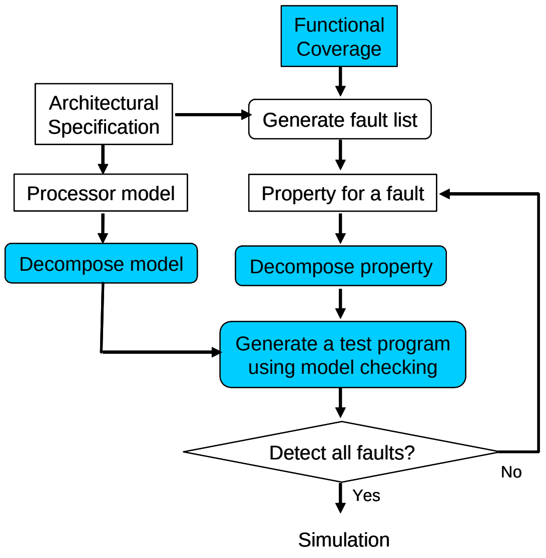

Fig. 1. Automated Test Program Generation Methodology

the fault list and the process continues until all the faults are detected.

The main contribution of this work is to provide 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. II presents related work addressing test generation in the context of micro-architectural validation of pipelined processors. III describes modeling of an industrial processor and our test generation methodology, followed by a case study in IV. Finally, V concludes the paper.

II. Relate Work

As a recent industrial micro-architectural test generation technique, Piparazzi [2] has been developed at IBM where a model of micro-architecture and the user's specification are converted into a Constraint Satisfaction Problem (CSP) and the dedicated CSP solver is used to construct an actual test program. Their technique requires additional confirmation of the conversion and construction procedures compared to using formal methods in test generation.

Several methodologies have been developed for validation of pipelined processors using finite state machine (FSM) models [3], [7], [8], [11] where FSM coverage based on reachable states and state transitions 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. [13] 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 [4], has been successfully used in software and hardware validation as a test generation engine [5], [10]. 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 [9] have proposed 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 [6] 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.

III. 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 [14]. For each interaction S_i, the

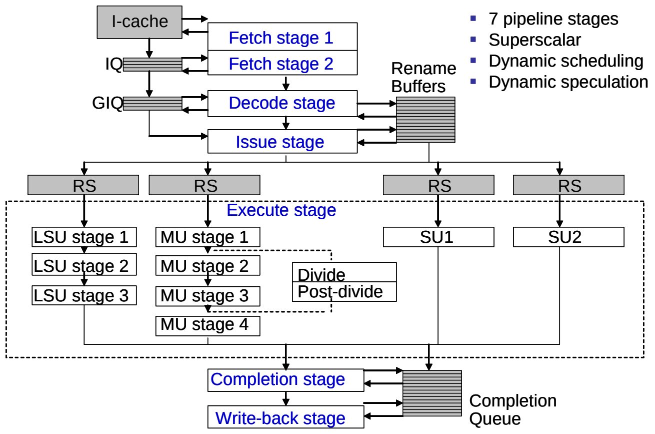

Fig. 2. Instruction Pipeline Flow of e500 processor that is based on the Power Architecture Technology

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.

A. 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 [15] 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 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 example, a requirement of Issue stage is that instructions can be issued out-of-order from only the bottom two entries and a rule is that an instruction cannot be issued if the reservation station for its unit currently holds a non-executing instruction. 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.

B. 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 propositional temporal logic LTL (Linear Temporal Logic) [4]. Each property consists of temporal operators (G, F, X, U) and Boolean connectives (∧, ∨, ¬, and →). 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 node *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:

$$\begin{array}{ll} \neg X(p) = X(\neg p) & \neg G(p) = F(\neg p) \\ \neg F(p) = G(\neg p) & \neg pRq = \neg pU\neg q \end{array}$$

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.

$$\begin{array}{ll} G(p \wedge q) = G(p) \wedge G(q) & F(p \vee q) = F(p) \vee F(q) \\ X(p \vee q) = X(p) \vee X(q) & X(p \wedge q) = X(p) \wedge X(q) \end{array}$$

TABLE I
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
7	Renaming for WAW (no stall)	8
8	Stall at Decode stage due to IQ full	14
9	Stall at Decode stage due to CQ full, then released queue full at the next clock cycle	34
10	CQ full, then full again	35
11	CQ full, then empty, and then full again	95
12	Only one instruction Completion (one or two instructions can retire per cycle)	12

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.

C. Test Generation using Decompositional 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 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.

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

IV. EXPERIMENTS

For evaluation of overall validation efforts to activate micro-architectural corner cases, we applied our directed test methodology on a commercial superscalar PowerPC processor at Freescale Inc. We performed various test generation experiments for validating the pipeline interactions and corner cases. Table I 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 except for the case 11 where test generation took approximately one minute. It is obvious that test generation time is proportional to the length of the generated test programs. Each of our test cases took less than 100 clock cycles on an average whereas the existing random tests in the company took approximately 100,000 clock cycles. Clearly, our approach can reduce the validation effort by several orders-of-magnitude.

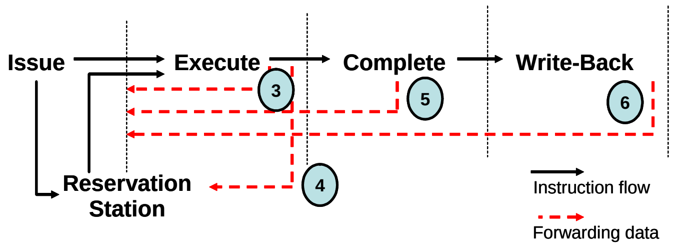

Fig. 3. Four Different Data Forwarding Mechanisms

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.

V. 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. Furthermore, this work can be an excellent complement to the current RTPG validation methodology without modification of the existing validation flow.

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. We also plan integrate our work in a performance analysis tool.

References

[1] 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 & Test*, 21(2):84–93, 2004.

[2] A. Adir, E. Bin, O. Peled, and A. Ziv. Piparazzi: A test program generator for micro-architecture flow verification. In *Proc. of High-Level Design Validation and Test Workshop (HLDVT)*, pages 23–28, 2003.

[3] D. Campenhout, T. Mudge, and J. Hayes. High-level test generation for design verification of pipelined microprocessors. In *Proc. of Design Automation Conference (DAC)*, pages 185–188, 1999.

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

[5] 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.

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

[7] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test pattern generation for pipelined processors. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 580–583, 1994.

[8] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Proc. of Design Automation Conference (DAC)*, pages 816–821, 2001.

[9] H.-M. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 1240–1245, 2006.

[10] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 182–187, 2004.

[11] 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.

[12] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, L. McConville, and T. Swanson. Verification of the cell broadband engine processor. In *Proc. of Design Automation Conference (DAC)*, pages 338–343, 2006.

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

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

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