Appeared in the International Conference on VLSI Design, pp. 300-305, Jan 1999

Superscalar Processor Validation at the Microarchitecture Level[1]

Noppanunt Utamaphethai, R.D. (Shawn) Blanton and John Paul Shen

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{nau,blanton,shen}@ece.cmu.edu

Abstract

We describe a rigorous ATPG-like methodology for validating the branch prediction mechanism of the PowerPC604 which can be easily generalized and made applicable to other processors. Test sequences based on finite state machine (FSM) testing are derived from small FSM-like models of the branch prediction mechanism. These sequences are translated into PowerPC instruction sequences. Simulation results show that 100% coverage of the targeted functionality is achieved using a very small number of simulation cycles. Simulation of some real programs against the same targeted functionality produces coverages that range between 34% and 75% with four orders of magnitude more cycles. We also use mutation analysis to modify some functionality of the behavioral model to further illustrate the effectiveness of our generated sequence. Simulation results show that all 54 mutants in the branch prediction functionality can be detected by measuring transition coverage.

1 Introduction

Formal verification [1, 2, 3, 4, 5] has been the focus of many current approaches to detect design errors. It provides a rigorous means to prove the correctness of a design by establishing that a mathematical relation holds between two descriptions of a system. A less rigorous but more practical approach to ensuring correctness involves design validation. In validation, one specifies "what can go wrong" in the design similar to a fault model in the automatic test pattern generation (ATPG) paradigm [8]. It relies on the simulation of generated test stimuli against the targeted design faults. Complete coverage of the universe of faults considered does not guarantee design correctness in the formal sense but is a proven process for uncovering bugs in the design.

Our long term goal is to develop a methodology for automatically deriving microarchitecture models from which test instruction sequences are generated for rigorously and systematically exercising the complex microarchitecture control behavior such as branch prediction, dynamic register renaming, pipeline interlock and others. This paper illustrates our approach for validating the branch prediction mechanism of contemporary superscalar processors. The PowerPC604 is used as our research vehicle and ATPG techniques are adapted for functional-level testing. Specific test program synthesis techniques are developed to obtain test sequences of various levels of confidence. The effectiveness of the generated sequences is shown through simulation comparisons with some real programs. We believe our validation method can be used to complement existing approaches for uncovering design errors with shorter simulation times. A technique in software testing called mutation analysis [13] is also used to illustrate the effectiveness of our generated sequences in detecting design errors.

The rest of this paper is organized as follows. Section 2 describes the branch processing details of the PowerPC604 and our FSM-like models of the 604's branch prediction mechanism at the microarchitecture level. In Section 3, we use these models to derive test sequences for validating the 604 branch prediction mechanism and in Section 4, we describe our method for transforming these test sequences into simulatable instruction sequences (i.e. test programs). In Section 5, we report experimental results by making comparisons with some real programs. We also evaluate the effectiveness of our sequences using mutation analysis [13]. Finally, in Section 6 we summarize our contributions and give directions for future work.

[1]This research effort is sponsored by the Semiconductor Research Corporation under contract DC068.070.

2 PowerPC604 Branch Prediction

The PowerPC604 branch prediction mechanism is a dynamic scheme that utilizes a branch target address cache (BTAC) and a branch history table (BHT) [10]. The BTAC is a 64-entry, fully associative cache with a round-robin replacement policy. It stores the target addresses of both unconditional and predicted-taken conditional branches.

The BHT is a 512-entry, direct-mapped cache that stores the execution history of conditional branches. Each two-bit entry of the BHT encodes the histories: strong not taken (SNT), weak not taken (WNT), weak taken (WT), and strong taken (ST). The history state is used to make predictions for conditional branches. If the state is either ST or WT, the conditional branch is predicted to be taken. Otherwise it is predicted not taken. The history table is updated when the actual branch is resolved to be taken (RT) or not taken (RNT).

Here, our focus is on the validation of the branch prediction mechanism only. In order to accomplish this task systematically, precise models of the prediction mechanism at the microarchitecture level are required. Small, FSM-like models are derived from the PowerPC 604 microarchitecture specifications [14]. It is important to note several characteristics about these models. They are not controlling FSMs that model real hardware in the 604 but microarchitecture abstractions of the branch prediction mechanism. Unlike traditional FSMs, state transitions in these models may require the execution of a sequence of PowerPC instructions. Conversely, a single PowerPC instruction may cause several state transitions.

2.1 Branch Target Address Cache

The BTAC is a table storing prediction information for both unconditional and conditional branches. The BTAC is modeled on an entry-by-entry basis using small FSMs. We use separate FSMs for the unconditional and conditional branches. Figure 1 shows the FSM model for a BTAC entry that corresponds to an unconditional branch. When an unconditional branch instruction is first encountered, a start transition (the RM self-loop in Figure 1) is made into the predicted-not-taken state (PNT), where the prediction of not taken (T=0) is made. The fetch address of the branch and its target address are loaded into the BTAC at the position of the round-robin pointer. After the branch is executed, the FSM transitions to the predicted-taken (PT) state. Now if the same unconditional branch is encountered, it is predicted to be taken (T=1) and the FSM responds by self-looping on the PT state. The transition from PT to PNT occurs when the branch is removed (the RM transition) from the BTAC by the entrance of another branch instruction when the round-robin pointer is at this entry.

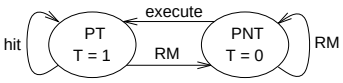

Figure 1: The FSM model of a BTAC entry for an unconditional branch instruction.

The model for a BTAC entry containing a conditional branch is somewhat more complex. Figure 2 shows a 6-state FSM that makes predictions based on branch histories. When a new conditional branch enters the BTAC, one of the four RM transitions (which is a BTAC miss) is made into a state where a prediction of not taken (T=0) is made. The specific RM transition performed depends on the branch history stored in the corresponding direct-mapped entry of BHT. When the conditional branch is resolved in the execution stage, a transition is made based on the outcome. (Thus, upon the first encounter of a branch, two transitions are made in this FSM-like model.) A branch resolved taken (RT) causes a state transition along the RT arc. A resolved not taken (RNT) outcome causes an RNT transition. The not-taken states (WNT and SNT) make not-taken predictions (T=0) while the taken states (WT and ST) predict branches to be taken (T=1). Future encounters of the branch instruction use the output value indicated in the current state (T=0 or T=1) for the prediction.

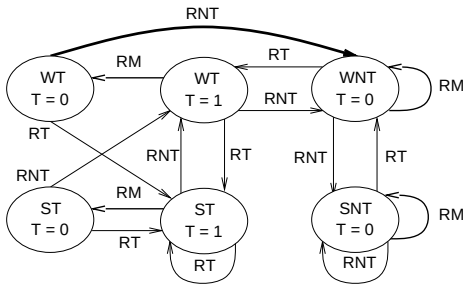

Figure 2: The FSM model of a BTAC entry for a conditional branch instruction.

2.2 Branch History Table

Similar to the BTAC, the operation of each BHT entry is modeled using a simple FSM. In the case of the BHT, the FSM shown in Figure 3 is the 2-bit saturating up-down counter. A cold start initializes all entries in the BHT to the start state SNT. *Any* conditional branch whose address directly maps to the same BHT entry will cause transitions in the FSM when the branch is resolved in the execution stage. The current state of the FSM is used to make predictions (possibly overriding BTAC predictions) for instructions in

the decode and dispatch stages. States WNT and SNT make a branch prediction of not taken (T=0) while states WT and ST predict taken (T=1).

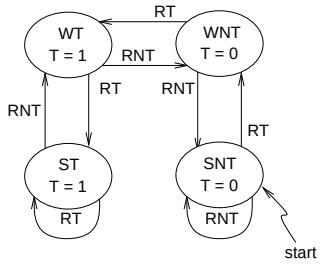

Figure 3: The FSM model of an entry in the BHT.

3 Branch Prediction Validation

The branch prediction mechanism typically consists of a series of operations that include retrieving the branch history information from storage, using the history to make a branch prediction, and updating the history once the branch is resolved. In the PowerPC604, a faulty prediction can result from the failure of any of these operations.

In the following, we describe our use of *partial transition tours*, *complete transition tours* and *checking sequences* for the BTAC and BHT FSM models to achieve various levels of branch prediction validation. The different testing sequences illustrate an interesting tradeoff between the test sequence complexity (in terms of length and effort in generating the sequence) and the amount of validation achieved.

The test sequences generated in RAM testing [9] target various types of manufacturing faults including stuck-line, coupling and pattern-sensitive. A test sequence for a given fault type can conceptually be viewed as a partial transition tour of an FSM model of one or more storage array cells or entries. A greater level of validation can be achieved by performing a complete tour of all FSM state transitions [7].

Probably the most powerful form of validation involves the use of a checking sequence [6]. It consists of three phases. The first phase synchronizes the machine into particular state given the general assumption that the starting state is unknown. The second phase verifies the existence of all the specified states while the third ensures the correctness of each state transition. Analysis of BHT's FSM (Figure 3) indicates that the FSM does not possess a distinguishing sequence. Therefore, identifying sequences must be used in any checking sequence for the BHT. The checking sequence for the BHT guarantees that the FSM of a BHT entry is indeed the machine described by Figure 3 and not one of the 16 million other possible machines of four or less states. Thus, any BHT entry not completely satisfying the microarchitecture specification will be discovered by the simulation of a checking sequence.

4 Test Sequence Generation

This section describes the instruction test sequence generation for the BTAC and BHT validation. Test sequence generation involves: (*i*) the translation of test sequences (described in the previous section) into PowerPC assembly instructions, and (*ii*) the incorporation of additional PowerPC instructions to ensure that every BHT and BTAC entry has a sequence applied for every type of branch instruction.

FSM test sequences are translated directly into PowerPC assembly instructions in order to minimize any unnecessary instructions that might result from the use of a high-level language compiler. Since different types of branch instructions affect various parts of the branch prediction mechanism, it is important to distinguish the branch types. In the PowerPC assembly language and many other processors, branch instructions can be classified into three groups based on the parts of branch prediction mechanism affected. For the PowerPC 604, the four unconditional branches (`b, bl, ba, bla`) affect the prediction state and round-robin pointer of the BTAC. The four conditional branches (`bc, bcl, bca, bcla`) affect the prediction and history state of the BTAC and BHT. These branches can also affect the round-robin pointer of the BTAC. Finally, there are four special branches (`bclr, bclrl, bcctr, bcctrl`) that do not affect the BTAC or the BHT at all.

State transitions in the BTAC and BHT FSM models are performed when branch instructions are executed. Hence, the validation of these mechanisms, using predefined transition tours, requires a sequence of taken and not-taken branch instructions. Automatic test sequence generation is facilitated by small subroutines that we call *atomic sequences*. An atomic sequence is a subroutine of less than ten assembly instructions that performs a transition in the BTAC or BHT FSM models. For example, the RNT transition (in bold) from state (WT,T=0) to (WNT,T=0) of Figure 2 is performed by the atomic sequence listed in Figure 4. The first instruction creates the condition for the branch to be resolved not taken (RNT). The next instruction actually performs the state transition and the last instruction returns program control to the caller. Atomic sequences for the remaining transitions in Figure 2 and the other FSMs are constructed in a similar fashion.

A "complete" set of atomic sequences covers every FSM transition for every branch type. Once a complete set is determined, a test sequence (partial or

```
BC_ADDR0:
            cmpi    0, 30, 1        # Set CR0
            bc      12, 2, BC0_0    # Make transition
BC_0:
            bclr    20, 0           # Return to caller
```

Figure 4: Traversal of the bold RNT arc of Figure 2 using an atomic sequence of assembly instructions.

complete transition tour, checking sequence, etc.) of assembly instructions can be automatically generated by concatenating a set of atomic sequences. We have used Perl scripts to automatically generate our test sequences from atomic sequence building blocks. Test sequences for complete transition tours of the BTAC and BHT have been generated. In addition, an instruction sequence that performs a checking experiment of the BHT has also been automatically constructed. Simulation results for these sequences along with comparisons to some real programs are presented in the next section.

5 Experimental Results

Our test programs are simulated on the PowerPC604 behavioral model using the trace-driven performance simulator MW [11, 12]. We have added checking code to MW in order to measure the amount of coverage obtained from our test sequence and real program simulations. Coverage is defined to be the percentage of targeted functionality exercised which, in our case, is traversed edges (transitions) in the FSM-like models.

5.1 Comparison to Real Programs

Figure 5 lists the statistics of some real programs. Given is the information about the number of simulation cycles needed to complete each program, the total number of instructions executed, and the number of different types of branch instructions. Note that the branch instruction types `ba`, `bca` and `bcla` are not present in any of the programs. Similar statistics are provided in Figure 6 for our four test instruction sequences. Our test sequences consist of all the branch types except `bca` and `bcla`. These two branch types could not be processed by the PowerPC assembler and loader, and as a result were excluded from our experiments.

Figure 7 shows the percentage of edges covered by the real programs and our generated sequences in the FSM models of the BTAC and the BHT. Each FSM transition in the BTAC FSMs can be traversed by any type of branch. Hence, the total number of times that a transition can be traversed by unconditional branches is $64 \times 4 = 256$ and $64 \times 2 = 128$ for conditional branches. It can be clearly seen that no individual program achieves 100% coverage (Figure 7a) and that the combined program coverage in both cases is at most 75% (Figure 7b).

	cjpeg	compress	eqntott	grep	mpeg	quick	All
Instr.	2.7M	72M	18M	2.3M	9M	737K	105M
`b`	76K	1.4M	334K	91K	67K	47K	2M
`bl`	13K	252K	104K	9.5K	8.8K	5.3K	393K
`ba`	0	0	0	0	0	0	0
`bla`	25K	0	2K	7	2.4K	5K	34K
`bc`	215K	12M	5.6M	348K	1.3M	128K	20M
`bcl`	53	144	16	275	19	18	525
`bca`	0	0	0	0	0	0	0
`bcla`	0	0	0	0	0	0	0
Cycles	1.9M	42M	10M	1.4M	4.8M	506K	61M

Figure 5: Dynamic instruction statistics for some real programs.

	BTAC unconditional transition tour	BTAC conditional transition tour	BHT transition tour	BHT checking sequence
Instr.	8.2K	240K	86K	1.3M
`b`	256	32K	0	0
`bl`	258	4	2	16
`ba`	256	0	0	0
`bla`	256	0	0	0
`bc`	0	832	4.1K	59K
`bcl`	0	832	4.1K	59K
`bca`*	0	0	0	0
`bcla`*	0	0	0	0
Cycles	8.2K	206K	72K	1M

Figure 6: Dynamic instruction statistics for our generated test sequences.

For the BHT FSM, each transition can be traversed by any conditional branch (excluding `bca`, `bcla`). Therefore, each BHT entry must be traversed 1024 times to achieve 100% coverage. The highest coverage achieved by an individual program (mpeg) is 17.6% (Figure 7a) and the combined program coverage is a mere 35% (Figure 7b).

As shown in Figure 7c, our test programs achieve 100% coverage with far fewer instructions. The real programs have a good coverage of the BTAC FSMs probably due to its small size. However, in the case of the BHT, the real programs have an edge coverage of less than 35%. Analysis of the real programs revealed that a large portion of the dynamic branch instructions in the real programs map to the same BHT entry, thereby causing the traversal of the same edge many times. Hence, the transition coverage in the BHT FSM is quite low despite the large number of branch instructions executed.

Our generated instruction test sequences achieve 100% transition coverage while keeping the number of dynamic instructions and the execution cycles low. The generated sequence that performs the checking experiment (Figure 6) is much larger but is still 60 times smaller than the combined set of the real programs. A direct comparison between some real programs and the

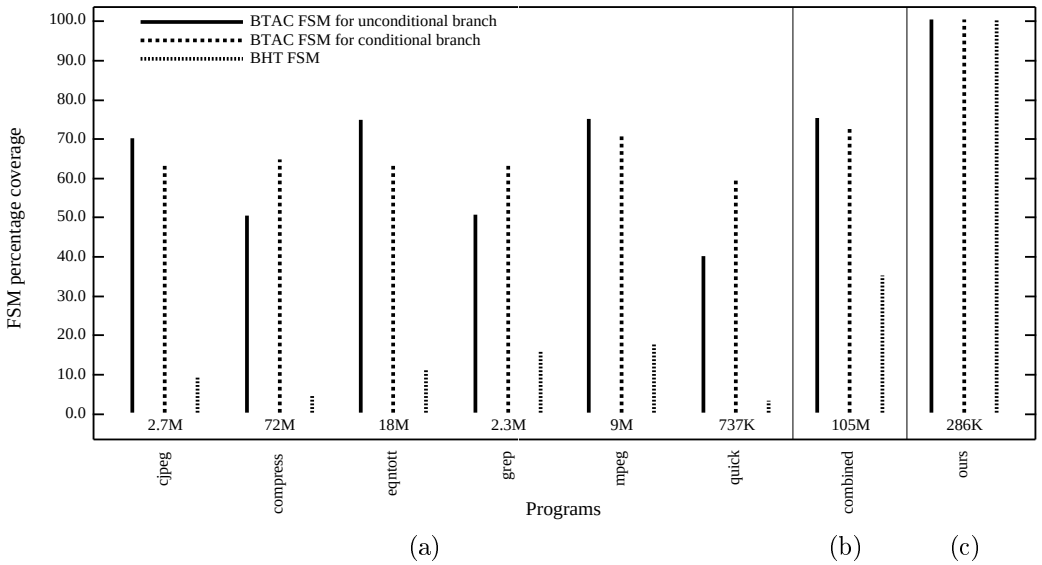

Figure 7: FSM transition coverage comparison between (a) real programs, (b) combined real programs and (c) our generated test programs.

checking sequences cannot be made since there exists a large number of input sequences that can serve as checking sequences. But given the fact that the coverage for all real programs is less than 100%, we can easily conclude that application of a checking experiment by the real programs does not occur.

5.2 Mutation Analysis

The increased use of software models in the design of digital systems has produced a variety of validation techniques based on software testing. *Mutation analysis* [13] is a software testing technique that develops test data for which the original program works correctly but, hopefully, for all other variations of the original program fail. A variation of the original program is called a *mutant*. Each mutant contains a simple programming error, or in our case, a design error. A mutant "dies" (is detected) if the generated test data produce incorrect output.

We use mutation to modify the behavioral descriptions of the BTAC and the BHT. Figure 8 shows an example of mutated design code. Figure 8a is the original code of the BTAC operation that checks the prediction of the BHT (`BHT_state`) before the target address of a branch instruction is added to the BTAC. Figure 8b shows the resulting mutants of the relational operator of Figure 8a.

The BTAC and the BHT behavioral descriptions that involve the update mechanism (like in Figures 8) are identified and modified to generate mutants. There is a total of 18 mutants from 9 different BTAC operations and 36 mutants from 12 other BHT operations. Simulation results show that all 54 mutants are detectable via transition coverage using our test programs. For a transition tour sequence, a mutant is considered to be detected if its transition coverage is different from that of the original behavioral model. For the BHT checking sequence, a mutant is detected if the sequence of resulting BHT states of any entry in the mutant is different from that of the original version. Figure 9 shows information that describes the detectability of the mutants. The BTAC and the BHT coverage columns show the percentage of BTAC and BHT mutants detected by each of our sequences. The total mutant coverage column shows the percentage of all detected mutants over all mutants. Note that some of the BTAC mutants are detected by the sequences that are targeted for the BHT, and vice versa. This

```
if (BHT_state > 1)
```
(a)

```
if (BHT_state ≥ 1)
if (BHT_state ≤ 1)
if (BHT_state < 1)
if (BHT_state == 1)
if (BHT_state ≠ 1)
```
(b)

Figure 8: Mutation of a relational operator: (a) Original code from the design description and (b) the resulting mutants.

	BTAC mutant coverage (%)	BHT mutant coverage (%)	Total mutant coverage (%)
BTAC (transition tour unconditional)	66.67	0.0	22.22
BTAC (transition tour conditional)	100.0	100.0	100.0
BHT (transition tour)	100.0	100.0	100.0
BHT (checking sequence)	100.0	100.0	100.0

Figure 9: Coverage of the mutants by our programs.

is analogous to the situation in single stuck-line (SSL) testing where SSL test patterns are found to serendipitously detect other types of faults such as bridging or delay faults. The results show that a significant number of the mutants are detected by test sequences that target a different part of the branch function.

Simulation results show that sequences generated for the BHT and BTAC conditional FSM transition tour detect all the mutants. But only two-thirds of the BTAC mutants can be detected via the sequence for transition tour of the BTAC unconditional branch FSM. The sequence generated for the BTAC unconditional branch FSM only contains unconditional branches, which obviously do not affect the BHT. As a result, the functionality of the BHT is not exercised and the BHT mutants cannot be "accidentally" detected by this sequence.

6 Summary

Our long term goal is to develop a rigorous methodology for automatically generating instruction sequences for exercising the control behavior of complex microarchitectures. Our initial approach has produced a procedure for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604. This was accomplished by (*i*) developing FSM-like models of the branch prediction mechanism at the microarchitecture level, (*ii*) generating test sequences based on these models, and (*iii*) automatically transforming the test sequences into PowerPC instruction sequences.

The effectiveness of our instruction test sequences was shown through comparative simulations with some real programs. Each generated sequence achieved 100% coverage of the targeted FSM transitions with significantly less instructions (four orders of magnitude on average) than those executed by the real programs. Simulations of the real programs produce transition coverages that range only between 34% and 75%. Simulation results from mutation analysis show that all 54 mutants of the BTAC/BHT functionality can be detected using our sequences.

We believe that the control complexities of many modern microarchitectures can be viewed as control logic reading and writing buffer entries. For future research, we want to generalize the approach for validating the branch prediction mechanism to validating other buffers in superscalar processors including: reservation stations, reorder buffers, rename buffers, and store buffers. The aim is to focus on a buffer, determine its functionality (i.e. reading and writing semantics), model each buffer's entry operation at the microarchitecture level using FSMs, and rigorously generate instruction sequences that systematically exercise these FSMs.

Acknowledgments

We would like to thank Bryan Black for sharing his knowledge of the PowerPC604 microarchitecture.

References

[1] M. C. McFarland. "Formal Verification of Sequential Hardware: A Tutorial," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 5, pp. 633–654, May 1993.

[2] E. M. Clarke and R. P. Kurshan. "Computer-aided Verification," *IEEE Spectrum*, pp. 61–67, June 1996.

[3] B. Tuck. "Taking the Mystery out of Formal Methods," *Computer Design*, pp. 67–75, Aug. 1996.

[4] R. K. Blackett. "As Good As Gold," *IEEE Spectrum*, pp. 68–71, June 1996.

[5] D. L. Beatty and R. E. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodology," In *Proc. of the 31st Design Automation Conference*, pp. 596–602, June 1994.

[6] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[7] S. Naito and M. Tsunoyama. "Fault Detection for Sequential Machines by Transition-tours," In *Proc. of the 11th Annual International Symposium on Fault-Tolerant Computing*, pp. 238–243, 1981.

[8] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* IEEE Press, Piscataway, NJ, 1991.

[9] M. S. Abadir and H. K. Reghbati. "Functional Testing of Semiconductor Random Access Memories," *Computing Survey*, Vol. 15, No. 3, pp. 175–198, Sept. 1983.

[10] Johnny K.F. Lee and Alan Jay Smith. "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, pp. 6–22, 1984.

[11] A. S. Huang and T. A. Diep. "MW Developer's Guide," *Technical Report, CMuART-95-1, Carnegie Mellon University*, Aug. 1995.

[12] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen. "Can Trace-driven Simulators Accurately Predict Superscalar Performance?," In *Proc. of International Conference on Computer Design*, pp. 478–485, Oct. 1996.

[13] A. T. Acree et al. "Mutation Analysis," *Technical Report, GIT-ICS-79/08, Georgia Institute of Technology*, 1979

[14] IBM Microelectronics and Motorola Inc. *PowerPC 604 RISC Microprocessor User's Manual.* IBM Corp. and Motorola Inc., 1994.