POLITECNICO DI TORINO
Repository ISTITUZIONALE

On the Functional Test of Branch Prediction Units based on Branch History Table

Original
On the Functional Test of Branch Prediction Units based on Branch History Table / SANCHEZ SANCHEZ, EDGAR ERNESTO; SONZA REORDA, Matteo; Alberto, Tonda. - STAMPA. - (2011), pp. 278-283. (Intervento presentato al convegno 19th IFIP/IFEE International Conference on Very Large Scale Integration and SoC tenutosi a Hong Kong (China) nel 3-5 October, 2011).

Availability:
This version is available at: 11583/2446375 since:

Publisher:
IEEE

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

12 October 2024

On the Functional Test of Branch Prediction Units based on Branch History Table

E. Sanchez, M. Sonza Reorda, A. Tonda
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
{ernesto.sanchez, matteo.sonza, alberto.tonda}@polito.it

*Abstract**— **Branch Prediction Units (BPUs) are highly efficient modules that can significantly decrease the negative impact of branches in superscalar and RISC processors. Traditional test solutions, mainly based on scan test, are often inadequate to tackle the complexity of these architectures, especially when dealing with delay faults that require at-speed stimuli application. Moreover, scan test does not represent a viable solution when Incoming Inspection or on-line test are considered. In this paper a functional approach targeting BPU test is proposed, allowing to generate a suitable test program whose effectiveness is independent on the specific implementation of the BPU. The effectiveness of the approach is validated on a Branch History Table (BHT) resorting to an open-source computer architecture simulator and to an ad hoc developed HDL testbench. Experimental results show that the proposed method is able to thoroughly test the BHT, reaching complete static fault coverage.**

Keywords-branch prediction unit; branch history table; functional test; sbst.

I. INTRODUCTION

Embedded system applications characterized by high performance requirements often resort to RISC or superscalar processors. In order to increase their performance, it is common practice to equip them with highly efficient Branch Prediction Units (BPUs), which can significantly decrease the negative impact of branches.

However, the complexity of these architectures, combined with the increased sensitivity to faults of new technologies, ask for suitable techniques able to effectively detect possible faults affecting them, at the end of the manufacturing process, for incoming inspection, and during the operational life (on-line test).

Unfortunately, traditional test solutions, mainly based on scan test, are often inadequate. First of all, because these solutions can hardly be exploited during the operational life, even for non-concurrent on-line testing; secondly, because companies involved in processor design and manufacturing tend not to disclose details about scan test architectures, in order to better achieve IP protection; this means that both for incoming inspection, and for end-of-production test of System-on-Chip (SoC) devices, scan test can hardly be adopted; thirdly, because scan test is generally inadequate for testing delay faults, that usually require at-speed stimuli application and response observation (not to mention the overtesting scan test tends to produce). For all these reasons, a functional test approach based on developing suitable test programs to be executed by each core and on observing the produced results is a much more suitable solution, provided that effective techniques are available for generating such test programs. This approach is also known as Software-Based Self-Test (SBST) [2].

Branch Prediction Units are among the most critical components within high-performance embedded systems, since their behavior can significantly affect the performance of the whole system. More specifically, faults affecting BPUs do not cause the generation of erroneous results, but rather slow down the system, increasing the number of mispredictions and possibly causing the system not to match the expected target in terms of performance.

BPU testing has been the subject of a few previous papers, such as [4] and [5]. The former proposes a hardware-based method, which requires the insertion of proper circuitry in the processor for BPU test. The latter follows the SBST approach, and mainly focuses on BPUs based on the Branch Target Buffer architecture. This paper reports a very convincing analysis of faults affecting BPUs, and proposes the usage of performance counters to detect them. However, the proposed method does not achieve full coverage of stuck-at faults, and requires very long test times. In [6], the authors use faults in BPUs as the typical example of the so-called Performance Degrading Faults, and analyze their impact on the performance of a processor, showing that their proper identification can significantly help improving the yield.

The authors of [7] propose a method to make Branch Prediction Units resilient to faults: however, the method is based on first detecting possible faults affecting each BPU, and then reconfiguring it, which raises even further the issue of how to test BPUs.

The purpose of this paper is to propose a new method to generate a proper test program to be executed by a processor in order to check whether the circuitry implementing its BPU works correctly. For the purpose of this paper, we target on BPUs implementing the Branch History Table (BHT) architecture. An important characteristic of our method is that

* This work has been partially supported by the Italian Ministry for University (MIUR) under the project PRIN08. Contact address: Matteo SONZA REORDA, Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino (Italy), e-mail matteo.sonzareorda@polito.it

it is based on a purely functional approach, i.e., it does not require any knowledge about the actual implementation of the circuitry it is intended to test, nor on the adopted semiconductor technology (and hence on the faults that can affect the circuitry). The test program is derived from the functional specifications of the circuitry under evaluation, only, and can therefore be reused on any circuit implementing the same branch prediction mechanism. Since the approach does not require the knowledge of any implementation detail, it is well suited to be adopted by OEM companies for both Incoming Inspection [1], and on-line test, as well as by semiconductor companies producing SoCs, when they decide to follow the functional approach (e.g., because they don't have access to structural information about the processor core).

The test approach proposed in this paper belongs to the SBST family; therefore, it can be applied at-speed, does not require any change in the processor or BPU hardware, and is particularly suitable to test delay faults. Moreover, being based on test programs to be executed by the processor, it can be activated at any time, even when the system is already in its operational phase. The relatively short duration of the test program makes it easily applicable even during concurrent on-line testing.

The approach has been validated resorting to a computer architecture simulator and a purposely developed VHDL module implementing a BHT.

The paper is organized as follows: section 2 reports some background about the BHT architecture and behavior. Section 3 describes the functional approach we propose for generating suitable test programs; section 4 reports some data about the experimental set up we devised and implemented to assess the effectiveness of the method. Section 5 draws some conclusions.

II. BACKGROUND

A. Branch History Table behavior

Branch prediction based on Branch History Table (BHT) exploits a data structure which stores the result (taken or not taken) of previously executed conditional branches. The BHT data structure contains N words, and is accessed during the Decode stage each time a conditional branch instruction is detected. To access the BHT, the n least significant bits of the instruction address are used, being $n = \log_2 N$. in this phase, the BHT returns a prediction, which is used by the processor to fetch the following instructions. If the prediction is correct, it can minimize the performance penalty stemming from the branch. After the branch instruction result becomes known, the BHT may be updated.

The BHT can be implemented in different manners: in the simplest version, each word in the table stores a single bit, recording whether the last time the associated branch has been executed its result has been taken (T) or not taken (NT).

In a different version, which is also considered in this paper, each word corresponds to 2 bits: their value records the results of the branch in the last 3 times it has been executed. If the branch has never been taken in that period, the stored value is 00, while the value is 11 if it has been taken 3 times. The value of the counter is used for prediction assuming that 00 corresponds to a "Strongly Not Taken" prediction, 01 to "Weakly Not Taken", 10 to "Weakly Taken", and 11 to "Strongly Taken". From an implementation point of view, this means that each word corresponds to a saturated 2-bit counter which is incremented each time the branch is taken, decremented elsewhere. During the Decode phase, the value of the counter associated to the branch is used to predict the branch result taking the dominant result over the last 3 executions of the branch.

B. Branch History Table architecture

Although the method proposed in this paper does not rely on any information about the implementation of the BPU, in the following we assume that the main components of a BPU based on a 2-bit saturated counters BHT are:

- A decoding logic, receiving the n least significant bits of the address of the branch instruction, and selecting the corresponding line of the BHT
- A table composed of a set of N 2-bit saturated counters, which can be read, incremented or decremented
- A multiplexer logic receiving the N 2-bit values coming from the counters, and selecting the line corresponding to the n bits
- A Taken / not Taken (T/nT) logic able to translate a 2-bit value in the branch prediction, as described before.

Figure 1 graphically summarizes the BHT architecture.

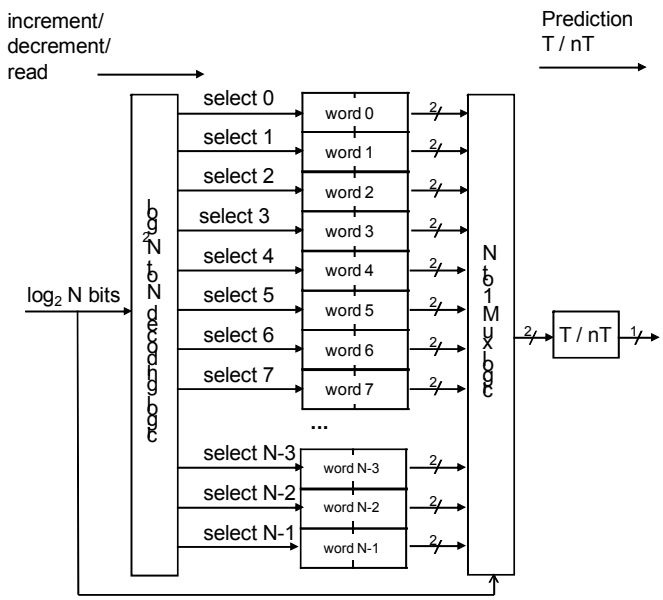

Fig. 1: BHT architecture when 2-bit saturated counters are considered

III. PROPOSED APPROACH

Testing a BHT-based BPU according to the SBST paradigm requires checking that each line in the BHT

- Can be correctly accessed
- Correctly implements the forecasted prediction.

In other words, the test aims at checking whether both the decoding logic and the memory implementing the BHT work correctly.

The first target can be achieved by resorting to the following algorithm, largely used in memory testing, which is known to be able to fully test the decoding logic of a memory [8]:

- M1: ↕ (w1)
- M2: ↑ (r1, w0)
- M3; ↓ (r0, w1)

The symbols ↑ and ↓ correspond to scanning the whole BHT with a given order, and with the opposite order, respectively, while ↕ corresponds to scanning the whole BHT in whichever order. Therefore, M1 corresponds to filling the memory with 1s, M2 to scanning the table in a given order, reading each word, checking whether it stores a 1 and writing a 0, M3 to scanning the whole memory in the opposite order, reading each word, checking whether it stores a 0 and writing a 1.

Let us now make the simplifying assumption that the BHT only stores one bit per word: in this case the above algorithm is sufficient to fully testing it. A read operation corresponds to accessing a word asking for a prediction, and checking whether the prediction is the expected one. This check can be performed in several possible ways, including the following:

- By resorting to the performance counters [9]existing in many processors and able to monitor the number of correctly/incorrectly executed predictions
- By resorting to some timer, able to measure the performance of the processor when executing a given piece of code, exploiting the fact that mispredictions imply longer execution time
- By resorting to some ad hoc module added to the system and able to monitor the bus activity [10].

On the other side, write operations correspond to updating a given word in the BHT: if the BHT implements a 1-bit prediction, this operation is performed when a misprediction arises. A w0 operation corresponds to an untaken mispredicted branch, a w1 operation to a taken mispredicted branch.

Hence, the test program for a BPU based on a 1-bit BHT requires 3 phases:

- Phase 1: N conditional branches, stored in suitable positions in the code memory, so that the n least significant bits of the corresponding addresses assume all the possible combinations over n bits; all the branches should be Taken, thus filling the BHT with 1s
- Phase 2: a set of N branches whose result is Not Taken: again, the N branches are suitably stored in the code memory, so that the n least significant bits of the corresponding addresses assume all the possible combinations over n bits; each branch is mispredicted, and their execution causes the BHT to be filled with 0s
- Phase 3: a set of N branches whose result is Taken: once more, the N branches are suitably stored in the code memory, so that the n least significant bits of the corresponding addresses assume all the possible combinations over n bits; each branch is mispredicted, and their execution causes the BHT to be filled with 1s.

The reader should note that the above algorithm is thoroughly able to detect faults not only in the decoding logic, but also in the table, since every cell is written with 0 and 1, and the written value is then read and checked. The result is then observed looking at the provided prediction, thus testing also the multiplexing and T/nT logic.

Let us now consider the case in which each line in the BHT implements a 2-bit saturated counter.

In this case the test to be performed on each line requires some more complex operation, corresponding to testing the correct functionality of the 2-bit saturated counter, which is graphically reported in Figure 2 for sake of clarity.

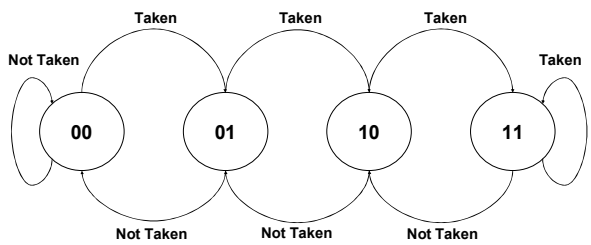

Fig. 2: 2-bit saturated counter behavior

In order to test the correct behavior of the 2-bit saturated counter we need to activate every transition, and then check whether it has been correctly executed, i.e., whether the correct state has been reached. In order to do so, the algorithm we propose is the following

- Initialize the counter to the 11 state: this can be achieved (no matter the initial state of the counter) by executing 3 Taken branches; this guarantees to reach the 11 state, regardless the initial state
- Execute 4 Not Taken branches: this moves the counter to the 00 state, producing 2 mispredictions, followed by 2 correct predictions, once again saturating the counter to the lower value and checking its ability to remain in the 00 state

- Execute 4 Taken branches: this moves the counter back to the 11 state, producing 2 mispredictions, followed by 2 correct predictions.

Now we should combine in a minimal program the test of the different components of the BHT, i.e., the decoding and multiplexing logic, the set of N 2-bit saturated counters, and the Taken / not Taken logic. Hence, the test program for a BPU based on a 2-bit BHT requires 3 phases:

- Phase 1: 3 × N Taken conditional branches, stored in suitable positions in the code memory, so that for every possible combination of the n least significant bits 4 branches exist; all the branches are Taken, thus moving all the counters in the BHT to the 11 state; the order of accesses to the BHT in this initialization phase is not important, nor it is important to check whether they cause correct or incorrect predictions (since we don't know the initial state of the counters);
- Phase 2: a set of 4 × N branches whose result is Not Taken: in this phase the order of accesses to the BHT lines is important, and the 4 accesses to each line should be completed before moving to the following line; this can be easily achieved by suitably storing the branches in the code memory; for every line 2 incorrect predictions followed by 2 correct ones should be produced; at the end of this phase all counters should be in the 00 state;
- Phase 3: a set of 4 × N branches whose result is Taken: in this phase the order of accesses to the BHT lines is also important, and should be the opposite than in the previous phase; the 4 accesses to each line should be completed before moving to the following line; for every line 2 incorrect predictions followed by 2 correct ones should be produced; at the end of this phase all counters should be back to the 11 state.

The third phase is particularly critical from an implementation point of view, since it requires that 4 Taken branches are executed for every BHT line; lines must be considered in a specified order, and the 4 accesses to a line must be performed before the 4 accesses to the following line. No further branch instructions can be executed to manage the program flow, because they would improperly access the table.

In order to solve this issue, we propose a technique exploiting the fact that the execution flow can be controlled either through branches, or through procedure call and return instructions: the benefit stemming from using the latter instructions in this case lies in the fact that they do not make access to the BHT. Therefore, what we suggest is to write a set of N procedures, whose body only contains a single conditional branch instruction, which is suitably stored in memory so that it refers to a different BHT line. Each phase in the test program outlined above can now be implemented by first setting the registers affecting the condition tested by the branch instructions so that they produce either a Taken or Not Take result, and then calling the procedures in the wished order and for the wished number of times.

To summarize, the whole algorithm requires the execution of 11 × N branches, being N the size of the BHT. For sake of clarity, the pseudo-code for the test program targeting a 8 lines 2-bits BHT is now reported.

```
.data
A: .word 0
B: .word 1

.code
…
lw R1,A(R0)

# M1: Phase 1
 lw R2,A(R0)
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK1
 …
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK7

# M2: Phase 2
 lw R2,B(R0)
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK6
 …
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK0

# M3: Phase 3
 lw R2,A(R0)
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK0
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK1
 CALL JUMPTK1
 …
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK6
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK7
 CALL JUMPTK7
```

```
END: end

.org 0xXX000
JUMPTK0: beq R1,R2,TK0
 nop
TK0: RET

.org 0xXX048
JUMPTK1: beq R1,R2,TK1
 nop
TK1: RET
…

.org 0xXX1F8
JUMPTK7: beq R1,R2,TK7
 nop
TK7: RET
```

In the reported code, a couple of data variables (*A* and *B*) are initialized to 2 different values and the reference register (*R1*) is initialized to A; then, the 3 described phases are reported. Every phase initially configures the comparison register (*R2*) to a value according to the desired outcome of the conditional jumps: *Taken* in the case registers *R1* and *R2* are equal, *Not Taken* if the registers are not equal. Then, a series of *CALL* instructions targeting every BHT line are executed guaranteeing the desired behavior. In the case of phase 1, it is expected to consecutively execute four taken branches for every BHT line in ascending (↑) order. Phase 2, on the contrary, executes 4 Not Taken branches for every BHT line in descending (↓) order. At the end of Phase 3, there are also reported 3 out of the 8 jump procedures, which are composed of a conditional jump instruction (*beq*), a *nop* instruction, and a *RET* instruction. It is important to note that the location in the code memory of the procedures, stated by the *.org* directive must be carefully selected.

IV. EXPERIMENTAL RESULTS

In order to validate the proposed approach we first resorted to SimpleScalar [11], an open-source system software infrastructure widely used for computer architecture research and teaching. SimpleScalar has several desirable features: it can implement a 2-bit BHT of arbitrary size, it can emulate several instruction sets (Alpha, PISA, ARM, x86), it can be modified to monitor and store the internal state of the processor, and its ISA is easily expandable to include new instructions.

The PISA architecture has been selected for our experiments, and SimpleScalar has been set to use a 2-bit saturated counter BHT of 1,024 lines. In order to check the effectiveness of the method, the SimpleScalar code has been modified to store some additional data during the simulation, allowing to record each time a transition of every 2-bit saturated counter in the BHT is fired. Finally, a dummy instruction has been added to the ISA to save the whole additional data structure to a file without altering the state of the emulated microprocessor.

Starting from a high-level *pseudocode* that describes the test algorithm, a program in assembly language comprising about 15,000 instructions is automatically generated. The program is then compiled and run on SimpleScalar.

The test program execution lasts for about 565,000 clock cycles. By comparing the data saved when the dummy instructions are executed, we verified that every transition in each counter of the BHT is fired, as expected, thus thoroughly exciting and observing the BHT, as desired.

On the other hand, a VHDL implementation of the targeted BPU was developed. The model described at RTL in VHDL counts about 600 code lines. The device was synthesized using Synopsys Design Vision targeting a homemade technology library. The synthesized version of the BHT counts 17,927 equivalent gates. The number of stuck-at faults for the entire BHT is 118,438.

To experimentally attest the efficiency of the approach, the execution of the program running on Simplescalar is then traced. The resulting signals are converted to VHDL and included in a testbench for the 1,024 entries BHT model. The fault simulation campaign was performed on Tmax v. B-2008.09-SP3 by Synopsys. The results gathered from this campaign showed that 100% stuck-at fault coverage on the model was reached, thus showing the effectiveness of the test program and confirming the results acquired from the SimpleScalar emulation.

V. CONCLUSIONS

We described a method for testing the circuitry implementing the Branch Prediction Unit of a processor, assuming that it is based on the Branch History Table architecture. The resulting test program can be used in different steps: at the end of the manufacturing step, exploiting its ability to perform an at-speed test; during incoming inspection, thanks to the fact that it does not rely on any information about the implementation of the circuitry; during on-line test, due to the fact that the method is relatively fast and does not require any special hardware to be applied. Experimental results showed the effectiveness of the approach.

We are currently working to extend the method to other Branch Prediction strategies and to decrease the code size and application times of the generated test programs, since in the proposed methodology, the code size and application times are proportional to the BHT dimension. Finally, we are evaluating the fault coverage capabilities of the method against different fault models (e.g., delay defects).

REFERENCES

[1] M.L. Bushnell, V.D. Agrawal, "Essential of Electronic Testing", Kluwer Academic Publishers, 2000

[2] M. Psarakis, D. Gizopoulos, E. Sanchez, M. Sonza Reorda, "Microprocessor Software-Based Self-Testing", *IEEE Design & Test of Computers*, vol.27, no.3, pp.4-19, May-June 2010

[3] M. Hatzimihail, M. Psarakis, D. Gizopoulos, A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware", Proc. IEEE International Test Conference, 2007, pp. 1-10

[4] S. Almukhaizim, P. Petrov, A. Orailoglu, "Low-Cost, Software-Based Self-Test Methodologies for Performance Faults in Processor Control

Subsystems", Proc. IEEE 2001 Custom Integrated Circuits Conference, pp. 263-266

[5] M. Hatzimihail, M. Psarakis, D. Gizopoulos, A. Paschalis, "A Methodology for Detecting Performance Faults in Microprocessors via Performance Monitoring Hardware", Proc. IEEE International Test Conference, 2007, paper 29.3

[6] T.-Y. Hsieh, M.A. Breuer, M. Annavaram, S.K. Gupta, K.-J. Lee, "Tolerance of Performance Degrading Faults for Effective Yield Improvement", Proc. IEEE International Test Conference, 2009, Lecture 3.1

[7] S. Almukhaizim, O. Sinanoglu, "Error-Resilient Design of Branch Predictors for Effective Yield Improvement", IEEE Latin-American Test Workshop, 2011

[8] A. J. Van de Goor, "Testing Semiconductor Memories, Theory and Practice", John Wiley & Sons, 1991

[9] B. Spunt, "The Basics of Performance-Monitoring Hardware", *IEEE Micro*, pp. 64-71, 2002

[10] W. J. Perez, J. Velasco-Medina, D. Ravotto, E. Sanchez, M. Sonza Reorda, "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs", Proc. IEEE International On-Line Testing Symposium, 2008, pp. 143-148

[11] http://www.simplescalar.com/