On the Functional Test of Branch Prediction Units Based on the Branch History Table Architecture

Ernesto Sanchez[1], Matteo Sonza Reorda[1], and Alberto Paolo Tonda[2]

[1] Dipartimento di Automatica e Informatica,
Politecnico di Torino
Corso Duca degli Abruzzi 24
10129 Torino, Italy
{ernesto.sanchez,matteo.sonzareorda}@polito.it
[2] Institut des Systèmes Complexes
Paris Île-de-France,
Rue Lhomond 57-59,
75005 Paris, France
alberto.tonda@iscpif.fr

Abstract. Branch Prediction Units (BPUs) are commonly used in pipelined processors, since they can significantly decrease the negative impact of branches in superscalar and RISC architectures. Traditional solutions, mainly based on scan, are often inadequate to effectively test these modules: in particular, scan does not represent a viable solution when Incoming Inspection or on-line test are considered. Functional test may stand as an effective solution in these situations, but requires effective algorithms to be available. In this paper we propose a functional approach targeting the test of BPUs based on the Branch History Table (BHT) architecture; the proposed approach is independent on the specific implementation of the BPU, and is thus widely applicable. Its effectiveness has been validated on a BPU 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 BPU, reaching complete static fault coverage with reasonable requirements in terms of test program size and execution time.

Keywords: branch prediction unit, branch history table, functional test, SBST.

1 Introduction

Embedded applications characterized by high performance requirements often exploit 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.

Nevertheless, the complexity of these architectures, combined with the increased sensitivity to faults of new technologies, requires 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).

S. Mir et al. (Eds.): VLSI-SoC 2011, IFIP AICT 379, pp. 110–123, 2012.
© IFIP International Federation for Information Processing 2012

Unfortunately, a number of reasons may sometimes make traditional test solutions, mainly based on scan, inadequate: first of all, these solutions can hardly be exploited during the operational life (even for non-concurrent on-line testing) since they require an external tester to drive the chain and observe the results. Secondly, companies involved in processor design and manufacturing tend not to disclose details about scan (or any other kind of Design for Testability) architectures, in order to better protect the Intellectual Property of their designs; this means that for both incoming inspection and end-of-production test of System-on-Chip (SoC) devices, scan or other Design for Testability techniques can hardly be adopted. Thirdly, scan is generally inadequate for testing delay faults, that usually require at-speed stimuli application and response observation (not to mention the overtesting scan 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 the generation of such test programs.

When targeting processors, the functional test approach generally translates into a carefully written test program suitably stored in a memory accessible to the processor, which executes it when test is triggered. Possible faults are detected by observing the results produced by the program, or by observing its behavior during program execution. This approach, originally introduced in [2], is currently known as Software-Based Self-Test (SBST) [3].

BPUs are among the most critical components within high-performance embedded systems, since their behavior can significantly affect the performance of the whole system. Interestingly, 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 [5] and [6]. 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. In [4] the authors report a very convincing analysis of faults affecting BPUs, and propose 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 [7], 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 to improve the yield.

The authors of [8] propose a method to make BPUs 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 describe an algorithm for the generation of 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 it is based on a purely functional approach, i.e., it does not require any information about the actual implementation of the circuitry it is intended to test, nor about 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 thus 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 system 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 and does not require any modification in the processor or BPU hardware. 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.

Code size and execution time of the programs generated with the proposed methodology scale linearly with the number of entries in the BHT.

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

A preliminary version of the paper has been presented in [12].

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.

2 Background

2.1 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 identify which instruction should be fetched at the following clock cycle. If the prediction is correct, the performance penalty stemming from the branch is reduced. After the branch instruction result becomes known the BHT is possibly 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 2 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 twice. 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, and decremented otherwise. 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 2 executions of the branch.

2.2 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, each of which can be read, incremented or decremented
- A multiplexer logic receiving the N 2-bit values coming from the counters, and selecting the correct one, based on the n least significant bits of the address of the branch instruction
- 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.

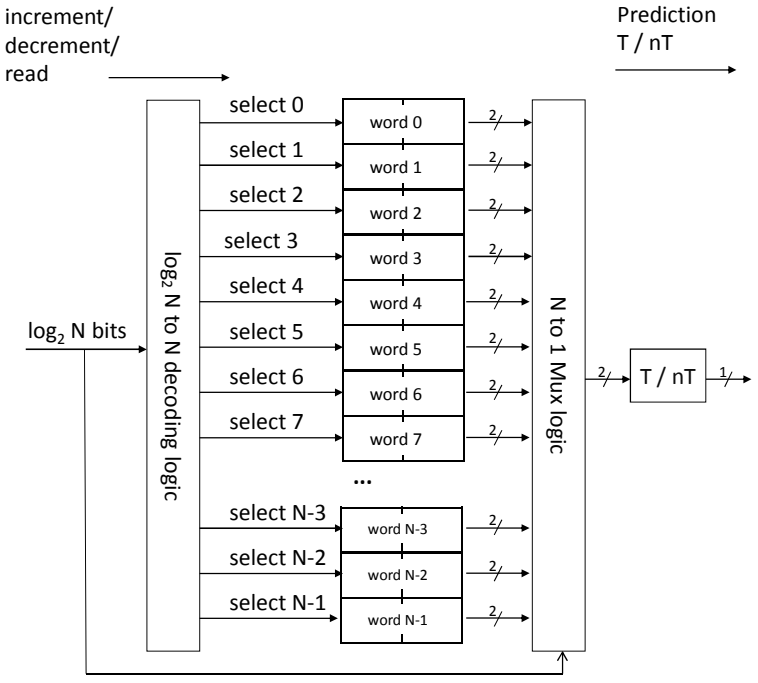

Fig. 1. BHT architecture when 2-bit saturated counters are considered

3 Proposed Approach

3.1 Basics

Testing a BHT-based BPU in accordance with the SBST paradigm requires checking that each line in the BHT

- Can be correctly accessed
- Stores the bit which was initially written to it
- Correctly implements the forecasted prediction.

In other words, the test aims at checking whether the decoding logic, the table and the surrounding logic all work correctly.

The first and second goals can be achieved by resorting to the following algorithm, largely used in memory testing; the algorithm is able to fully test the decoding logic of a memory and to detect any fault preventing a memory cell to work correctly [9]:

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

The symbols ↓ and ↑ correspond to a scan of the whole BHT in a given order, and in the opposite order, respectively, while ↕ corresponds to scanning the whole BHT in whichever order. Each line corresponds to a scan of the whole memory, performing the specified operation on each word before moving to the following word, according to the specified 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 it with 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 into it.

3.2 Test of a 1-Bit BHT

Let us first consider a BHT that only stores one bit per word: in this case the above algorithm is sufficient to fully test 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 [10] existing in many processors and able to monitor the number of correctly/incorrectly executed predictions
- By resorting to a 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 debug feature provided by the processor [13]
- By resorting to some ad-hoc module added to the system and able to monitor the bus activity [11].

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 observed looking at the provided prediction, thus testing also the multiplexing and T/nT logic.

3.3 Test of a 2-Bit BHT

Let us now consider a BHT where each line implements a 2-bit saturated counter.

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

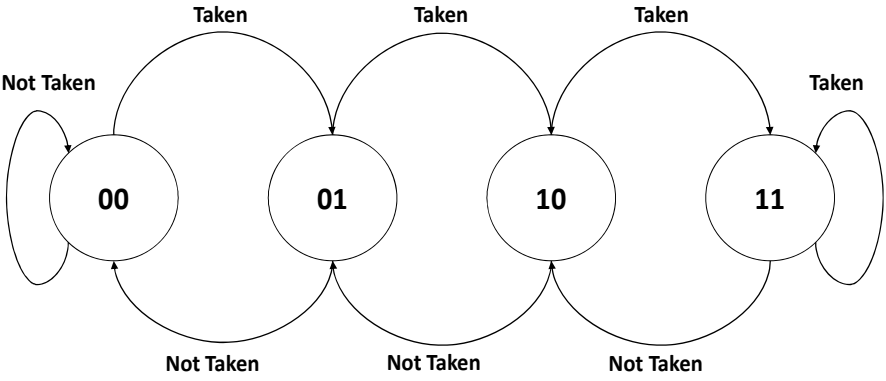

Fig. 2. 2-bit saturated counter behavior

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 of 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 of 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 the 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: instructions belonging to the latter type have the benefit of not accessing 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 in two steps: 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 desired order and for the desired 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 two different values and the reference register (*R1*) is initialized with the value in A; then, the three described phases are executed. Every phase initially configures the comparison register (*R2*) to a value according to the desired outcome of the conditional jumps: *Taken* in the case the value stored in registers *R1* and *R2* is the same, *Not Taken* if it is different. 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 four Not Taken branches for every BHT line in descending (↓) order. We also reported 3 out of the 8 required procedures, which are composed each 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 in the code) must be suitably selected, so that the branch instructions access the right lines in the BHT. However, it should be noted that this constraint can easily be fulfilled, since it only concerns the least significant bits of the branch instructions address; in other words, we need to store the test program in some area of the memory that can freely be located at the most suitable addresses from the application point of view, while the branch instructions must be carefully located within this area. For this reason, in the code above the addresses of the memory areas where the procedures are stored include some Xs in the code above.

3.4 Theoretical Analysis

In this sub-section we analytically calculate the number of instructions required to implement the approach proposed in the previous sub-section, and the number of clock cycles that the final program takes to execute.

As the reader should observe, the code derived from the proposed approach is mainly divided in 2 different parts: one part contains the 3 different code phases labeled as `M1`, `M2`, and `M3`, whereas another part is composed of the jump procedures (`JUMPTK0`, `JUMPTK1`…).

Each of the three phases described in section 3.3 follows a very regular pattern corresponding to:

- one control instruction
- $N \times m$ CALL instructions.

The control instruction writes into the processor register used by the conditional branch within each procedure, N is the number of lines of the BHT, and m is the number of accesses to each BHT line performed by each program phase (3, 4, 4, respectively).

On the other hand, every jump procedure counts on only 3 instructions (the conditional branch, a NOP and the return instruction).

Thus, the number of instructions required to implement the proposed approach can be calculated by the following formula:

$$N\times 3 + \sum_{i=0}^{2}(N\times m_i + 1) \quad (1)$$

with m_i that varies according to the BHT accesses required by every phase (3, 4, 4, respectively). For example, in the case of a BHT containing 1,024 entries the final test program corresponds to the execution of 14,339 instructions.

In order to better determine the test program execution time, it is necessary to take into account some architectural characteristics of the processor pipeline, such as the pipeline length, and other parameters, such as the result of each performed branch prediction, or the instruction type. For this purpose, we make the following assumptions:

- all the instructions different from branch instructions always require 1 clock cycle to execute; this means that no pipeline stalls arise (e.g., due to cache misses)
- branch instructions correctly predicted require 1 clock cycle
- branch instructions incorrectly predicted require *p_c* additional penalty cycles, whose exact number depends on the processor architecture
- *CALL* and *RET* instructions always require *p_c* clock cycles, since these instructions are not predicted (and do not use the BHT).

It is also important to note that the proposed program behaves in a different way according to the exercised branch in every one of the three program phases:

- Taken branch: for every *CALL*, the program executes the branch instruction, and the *RET* instruction only.
- Not Taken branch: for every *CALL*, the program executes the branch instruction, a NOP instruction, and the *RET* instruction.

At this point, it is possible to state the following expression that computes the execution time considering the three phases of the program:

$$T_{exe}=N\times\{T_{p!=b}[p\_c + S_{tb}] + T_{p=b}[1 + S_{tb}] + nT_{p!=b}[p\_c + S_{ntb}] + nT_{p=b}[1 + S_{ntb}]\} \quad (2)$$

where N is the number of entries in the BHT, $T_{p!=b}$ is the number of conditional branches wrongly predicted when branches are taken, p_c is the penalty due to wrong prediction branches, S_{tb} is the number of clock cycles required to execute the procedure instructions when the branch is taken, $T_{p=b}$ is the number of conditional branches correctly predicted when branches are taken, $nT_{p!=b}$ is the number of conditional branches wrongly predicted when branches are not taken, S_{ntb} is the number of clock cycles required to execute the procedure instructions when the branch is not taken, and $nT_{p=b}$ is the number of conditional branches correctly predicted when branches are not taken.

From equation (2) we can state that the complexity of the proposed test algorithm grows linearly with the BHT size.

Considering a 1,024 entries BHT, and a penalty of 2 clock cycles for incorrect prediction, equation (2) can be expressed as follows:

1,024×{ 4[2+4] + 3[1+4] + 2[2+5] + 2[1+5]} = 66,560 clock cycles.

4 Experimental Results

In order to validate the proposed approach we resorted to SimpleScalar [14], 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 (whose ISA is very similar to the MIPS one) has been selected for our experiments, and SimpleScalar has been configured to use a 2-bit saturated counter BHT.

In order to check the correctness of the method, the SimpleScalar source code has been modified to store additional data during the simulation, thus recording each time a transition of a 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. Using a 1,024 entries BHT, by comparing the data saved when the dummy instructions were executed, we verified that every transition in each counter of the BHT was correctly fired, as expected, thus thoroughly exciting and observing the BHT, as desired. In this way we were able to validate the correctness of the proposed algorithm.

Furthermore, we performed some experiments to practically validate our previous assumptions in terms of code size and execution time. In particular, we performed four different experiments tackling BHTs of incremental size. Starting from a high-level *pseudocode* that describes the test algorithm, a program in assembly language was parametrically generated. The program was then compiled and run on SimpleScalar. For every experiment, we used the `sim-outorder` simulator customized as described before. Additionally, some of the cache configuration parameters were set in order to minimize the collateral effects due to cache misses during the programs simulations. Table 1 shows the different BHT configurations in the first column (*BHT size*), the expected time (in terms of number of clock cycles) in the second column, and the execution time for every case in the third column.

Table 1. Test program execution times for different BHT sizes

BHT size	*Expected [c.c.]*	*Simulated [c.c.]*
256	16,640	18,890
512	33,280	37,084
1,024	66,560	73,141
2,048	133,120	145,559

Interestingly, the collected results present the same linear trend we theoretically forecasted in the previous sections. Figure 3 illustrates a plot of the collected results (extracted from the simulation and from the theoretical forecast), showing a significant match between the expected results and the simulated ones.

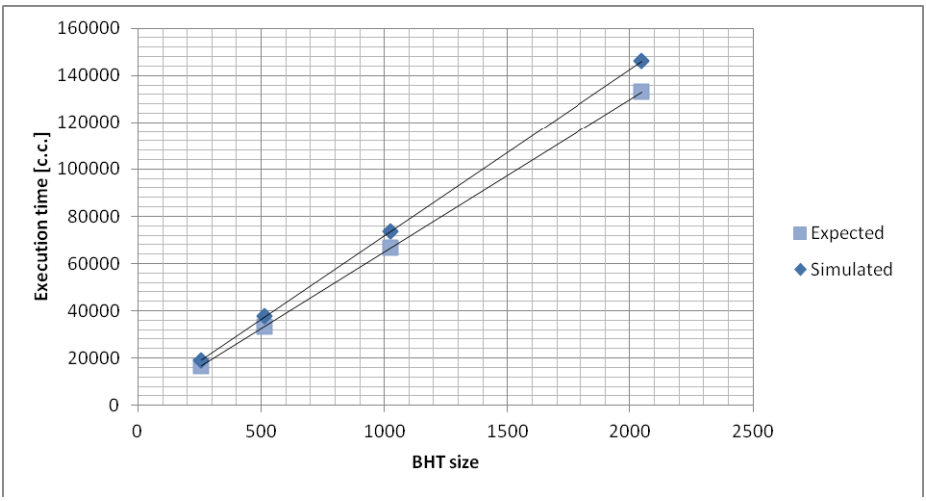

Fig. 3. Test program execution time vs. BHT size

Referring once more to a 1,024 lines BHT, a VHDL implementation of the BPU was then 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 validate the behavior of the approach, the execution of the program running on Simplescalar was then traced. The resulting signals were converted to VHDL and included in a testbench for the 1,024 entries BHT model. A fault simulation campaign was finally performed on Tmax v. B-2008.09-SP3 by Synopsys. For the purpose of these experiments, we assumed to be able to fully observe the processor behavior during the test program execution, both in terms of produced data and of timing execution. 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.

5 Conclusions

We described a method for functionally 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 or when the test cannot exploit any Design for Testability solution; during incoming inspection, since 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. The test program can be allocated in any part of the

memory, while the constraints about the position of the branch instructions within this area have been clearly specified. Both the execution time and the code memory size of the test program scales linearly with the BHT size.

Experimental results confirmed the analysis and 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. Finally, we are evaluating the fault coverage capabilities of the method against different fault models (e.g., delay defects).

References

1. Bushnell, M.L., Agrawal, V.D.: Essential of Electronic Testing. Kluwer Academic Publishers (2000)
2. Thatte, S., Abraham, J.: Test Generation for Microprocessors. IEEE Transactions on Computers 29(6), 429–441 (1980)
3. Psarakis, M., Gizopoulos, D., Sanchez, E., Sonza Reorda, M.: Microprocessor Software-Based Self-Testing. In: IEEE Design & Test of Computers, vol. 27(3), pp. 4–19 (2010)
4. Hatzimihail, M., Psarakis, M., Gizopoulos, D., Paschalis, A.: A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In: Proc. IEEE International Test Conference, pp. 1–10 (2007)
5. Almukhaizim, S., Petrov, P., Orailoglu, A.: Low-Cost, Software-Based Self-Test Methodologies for Performance Faults in Processor Control Subsystems. In: Proc. IEEE 2001 Custom Integrated Circuits Conference, pp. 263–266 (2001)
6. Hatzimihail, M., Psarakis, M., Gizopoulos, D., Paschalis, A.: A Methodology for Detecting Performance Faults in Microprocessors via Performance Monitoring Hardware. In: Proc. IEEE International Test Conference, pp. 1–10 (2007)
7. Hsieh, T.-Y., Breuer, M.A., Annavaram, M., Gupta, S.K., Lee, K.-J.: Tolerance of Performance Degrading Faults for Effective Yield Improvement. In: Proc. IEEE International Test Conference, pp. 1–10 (2009)
8. Almukhaizim, S., Sinanoglu, O.: Error-Resilient Design of Branch Predictors for Effective Yield Improvement. In: Proc. IEEE Latin-American Test Workshop, pp. 1–6 (2011)
9. Van de Goor, A.J.: Testing Semiconductor Memories, Theory and Practice. John Wiley & Sons (1991)
10. Spunt, B.: The Basics of Performance-Monitoring Hardware. IEEE Micro, 64–71 (2002)
11. Perez, W.J., Velasco-Medina, J., Ravotto, D., Sanchez, E., Sonza Reorda, M.: A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs. In: Proc. IEEE International On-Line Testing Symposium, pp. 143–148 (2008)
12. Sanchez, E., Sonza Reorda, M., Tonda, A.: On the functional test of Branch Prediction Units based on Branch History Table. In: Proc. IEEE/IFIP 19th International Conference on VLSI and System-on-Chip (VLSI-SoC), pp. 278–283 (2011)
13. Grosso, M., Sonza Reorda, M., Portela-Garcia, M., Garcia-Valderas, M., Lopez-Ongil, C., Entrena, L.: An on-line fault detection technique based on embedded debug features. In: Proc. IEEE International On-Line Testing Symposium, pp. 167–172 (2010)
14. SimpleScalar LLC, `http://www.simplescalar.com/`