[Proceeding] On the functional test of the BTB logic in pipelined and superscalar processors

On the functional test of the BTB logic in pipelined and superscalar processors

D. Changdao, M. Graziano, E. Sanchez, M. Sonza Reorda, M. Zamboni, N. Zhifan
Politecnico di Torino
Torino, Italy

Abstract—**Electronic systems are increasingly used for safety-critical applications, where the effects of faults must be taken under control and hopefully avoided. For this purpose, test of manufactured devices is particularly important, both at the end of the production line and during the operational phase. This paper describes a method to test the logic implementing the Branch Prediction Unit in pipelined and superscalar processors when this follows the Branch Target Buffer (BTB) architecture; the proposed approach is functional, i.e., it is based on forcing the processor to execute a suitably devised test program and observing the produced results. Experimental results are provided on the DLX processor, showing that the method can achieve a high value of stuck-at fault coverage while also testing the memory in the BTB.**

Keywords: SBST, Branch Prediction Unit, Branch Target Buffer, Test Program Generation.

I. INTRODUCTION

Branch Prediction Units (BPUs) are crucial components in pipelined processors, since they allow reducing the performance penalty stemming from branches. Among the several architectures, Branch Target Buffers (BTBs) offer the advantage that they not only provide a prediction on the branch result (i.e., Taken or Not Taken), but also give a prediction on the address of the target instruction. Hence, the correct behavior of the BPU (which may be impaired by possible faults) significantly impacts the processor performance, and this situation may not be acceptable when the processor is used in a real-time safety-critical application. Examples of these situations may easily be found in several domains, such as automotive, factory automation, avionics, space.

For the above reasons, several regulations (e.g., ISO 26262 in automotive, or DO-254 in avionics) define strict constraints (e.g., in terms of achieved fault coverage) for the tests to be performed during the manufacturing process and the operational phase in order to detect possible faults.

In order to implement an effective on-line test for processor-based systems, different approaches can be exploited. Most of them are based on suitable Design for Testability (DfT) solutions (e.g., BIST). As an alternative, the functional approach is increasingly used, especially when processor-based systems are considered. In this case it is also called *Software-Based Self-Test* (SBST) [1] and requires the development of a suitable test program that exercises the processor components and makes possible faults observable by looking at the produced results. Although already proposed and largely investigated in the past decades, the functional approach suffered from a period of lower popularity due to the wide adoption of Design for Testability techniques, such as scan and BIST, that can guarantee high fault coverage with low cost (in terms of area, performance, and development time). More recently, there is a clear trend for a new interest in functional test, due to a number of reasons:

- the System on Chip (SoC) design paradigm often mandates the usage of existing IP cores, which not always support DfT, and that can often not be modified to include it
- DfT solutions generally test the system after performing some reconfiguration; hence, the Unit Under Test may be tested under different operating conditions than in the normal mode; this may cause overtesting in some cases, while in others may prevent from achieving a sufficiently high defect coverage (e.g., with respect to delay testing)
- DfT solutions are not always usable during the operational phase, either because they have not be designed for this purpose, or because the device designer and/or manufacturer does not disclose enough information on it to the system company, or because they are too invasive with respect to the operational constraints of the system (e.g., in terms of memory content).

On the other side, the functional approach has a major drawback: it requires the development of a suitable test program. This task is currently not automated, and often it also lacks any guideline or test algorithm to be followed during the test program development. Hence, it may involve a high cost, especially if high fault coverage figures must be achieved.

For this reason, several research efforts have been made in the recent years to develop effective techniques able to guide the test engineer in the development of functional test programs, or even to develop automatic techniques for their generation.

When addressing complex processors, a common approach is based on addressing separately specific units within the processor, such as the basic pipeline [3], the cache [5][6], or the BPU [7]. Some of the addressed units (e.g., the caches) include relatively large memories [6]: for this reason the test often addresses first the memory, and the proposed algorithms have been shown to test them well. In a second step, the surrounding logic is also addressed, and the proposed algorithms are possibly improved to test it as well [5].

The same approach is followed in this paper for BPUs: following the idea in [7], we first extend it to BTB-based BPUs addressing the memory. Then, we evaluate the coverage (in terms of stuck-at faults) that can be achieved in the logic

surrounding the BTB memory using the test program addressing the BTB memory. Moreover, we identify the main sources of untested faults. Finally, we devise some extensions to the test program that allows to further increase the fault coverage. The proposed algorithm is characterized by the fact that it does not require any specific information about the architecture of the BPU, but only relies on the function performed by this unit. Hence, it can be easily adapted to BTB-based BPUs whose structure and function are different than the ones adopted by the paper.

The proposed approach is hardly suitable for being adopted by processor manufacturers for end-of-production testing: for these situations, DfT solutions are typically more suitable. On the other side, the proposed approach may turn out to be effective for SoC testing, or for the on-line test of processor-based systems.

This paper extends the solutions and results presented in [8], where BTBs characterized by a small number of entries and by a FIFO-based entry selection mechanism are considered.

Experimental results gathered on a modified version of the DLX processor [2] show the effectiveness of the proposed approach and allow the reader to evaluate the duration and size of the test program. The method can be easily extended to other processors, and only requires the knowledge about the key parameters of the BTB (e.g., number of entries and fields).

The paper is organized as follows. Section II briefly overviews the behavior and architecture of BTB-based BPUs. Section III describes the algorithm for functionally testing the BTB memory, while Section IV explains how to extend the algorithm to better cover the stuck-at faults in the surrounding logic. Experimental results on the selected case study and their analysis are presented in Section V. Finally, conclusions and future works are described in Section VI.

II. BTB-BASED BPUS

Several kinds of BTB-based BPUs have been proposed and adopted in practice. The solution we considered in this paper is basically the one described in [2], whose behavior and architecture are briefly summarized in the following.

When a processor embeds a BTB-based BPU, it accesses it each time an instruction (not necessarily a branch one) is fetched. The BTB receives the instruction address and answers in two possible ways (depending on the previous history):

- If the instruction is a branch and its result was a branch at its last execution, it returns the target address, which is used by the processor to execute the fetch of the following instruction
- Otherwise, the prediction is for not performing any branch, and the processor continues to fetch instructions sequentially from memory.

When the result of a branch instruction is known (which normally happens in the Execute stage), the prediction is checked and, if incorrect, the BTB is accessed to store the updated information about the instruction.

From an implementation point of view, a BTB is built around a memory composed of different entries; each of them stores the key information concerning a single branch instruction:

- its *address*
- its *target* address.

In more complex implementations, other fields are also used, which are not considered in this paper for sake of simplicity.

The BTB entry storing the information about a given branch instruction is normally selected using the least significant bits in the instruction address.

The behavior of a BTB and its interaction with the stages of a typical pipelined processor are summarized in Fig. 1.

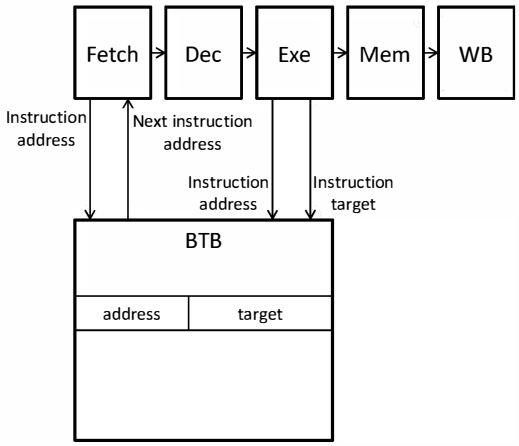

Fig. 1. Interaction of a BTB with the stages of a pipelined processor

III. TEST ALGORITHM FOR THE BTB MEMORY

By suitably extending the concepts introduced in [7], it is possible to transform whichever March algorithm into a corresponding test program executing the same sequence of read/write operations on the BTB memory.

In particular, a read operation on the generic x-th BTB entry can be performed by executing a branch instruction whose address in memory is chosen so that the x-th entry is accessed, i.e., whose least significant bits correspond to the address of the entry. The result of this read operation (performed while the instruction is in the Fetch stage) is the predicted target address.

A write operation on the generic x-th BTB entry can be performed by executing a branch instruction which is located in memory in such a way that it refers to the x-th BTB entry and which produces a misprediction, so that the entry is updated when the instruction reached the Execute stage.

By checking whether the expected prediction is given by the BTB when a branch instruction is executed, possible faults affecting the BTB memory can be detected. Notably, these faults typically belong to the category of *performance faults* [9], i.e., they do not produce wrong results, but rather cause a change in the time required to execute a given sequence of instructions. Hence, their detection requires suitable techniques to measure the program performance, or to count the number of predictions/mispredictions provided by the BPU [10].

Transforming a March algorithm for testing the BTB memory into a test program raises two major issues:

- How to execute a sequence of branch instructions suitably placed in memory, so that the BTB memory can be accessed in one order, or in the opposite; this result is difficult to achieve since no other branch instructions that modify the BTB have to be executed, apart from those required by the algorithm. This issue can be solved by resorting to procedures, since CALL and RETURN instructions typically do not affect the BTB [7].
- The resulting test program requires the branch instructions to be distributed widely over the all memory space accessible by the processor; while this constraint can be easily matched when the device is tested by itself, it is really hard to fulfill when the device is already deployed in the field, and the other constraints related to the application also have to be considered. This issue can only be solved by adopting hybrid solutions involving some hardware support, such as the one proposed in [11], or by resorting to some Memory Management Unit .

Let us now make some assumptions and introduce some notations:

- the processor code memory is composed of $M=2^m$ words
- the BTB is composed of $N=2^n$ words
- each BTB entry includes an address and a target field; the values of these two fields in the BTB i-th entry are denoted by a_i and t_i, respectively.

In order to access the BTB x-th line we can force the processor to execute an instruction whose n least significant address bits hold the value x. We obviously neglect the k least significant bits in the instruction address, being 2^k the byte size of each instruction, since the least significant k bits of the instruction address always hold the value 0 for sake of alignment.

Hence, by denoting as $<a_x,t_x>$ the content of the BTB x-th line at a given time, we can read it by executing a branch instruction stored at the address a_x:x:$zero^k$ (corresponding to the concatenation of three fields, whose length in bits is m-n-k, n, and k, respectively, and $zero^k$ is a sequence of k 0s) and performing a branch to the address t_x. If the BTB is fault-free it correctly predicts the branch result and target.

If we want to write into the x-th line of the BTB we can execute a branch instruction located at the address b_x:x:$zero^k$, where b_x is different than the value a_x currently stored in the BTB line; hence, the instruction outcome cannot be correctly predicted when the BTB is accessed and this causes an update in the BTB content. If we denote by r_x the target address of the instruction, the considered instruction writes the value $<b_x, r_x>$ in the x-th line of the BTB.

If intra-word coupling faults are not an issue, the conventional 0 and 1 values used by the generic March algorithm can be modified to support the test of an m-bit memory, provided that 0 = s (s being any value on m bits), and 1=not(s). Hence, we can assume that the 0 value corresponds to the concatenation of two values a_0 and t_0: the sequence of operations mandated by a March Element such as ↓w0 can now be obtained through a sequence of branch instructions stored at consecutive addresses in memory (starting from the address a_0:$zero^n$:$zero^k$), and all jumping to the same address t_0. The value of a_0 depends on the location of the memory area storing the sequence.

To provide the reader with a clarifying example, Fig. 2 reports the sequence of instructions executing the ↓w0 March Element on a N-entries BTB. In the following we will adopt the assembly language of the DLX processor, which has been used to experimentally evaluate the proposed approach. The CALL instruction is `jalr`, which jumps to the procedure whose starting address is stored in the operand register (in our case `r2)`, saving in the `r31` register the return address. The procedure code includes a jump instruction (`beqz`) whose result is a jump or not, depending on the value in `r1`. The RETURN instruction is `JR`, whose operand is the register storing the return address (i.e., `r31`). A suitably allocated memory vector (pointed to by `r0`) is used to store the addresses of the procedures, which are placed in memory in suitable locations, so that the `beqz` instruction existing in each of them refers to a different BTB line. By suitably adjusting the value of `r2`, one can activate the procedures in one order or in the reverse one.

Using the above techniques one can easily write the test program corresponding to whichever March algorithm.

IV. THE PROPOSED ALGORITHM

In Section V we experimentally show that a significant percentage of the stuck-at faults existing in the logic surrounding the BTB memory are not detected by the test program corresponding to a generic March algorithm for the test of the memory. This experimentally demonstrates the importance of developing a more focused test algorithm specifically addressing the logic in the BTB.

The size of this logic is not negligible, and may easily account for some thousands of equivalent gates; hence, it is desirable to devise solutions able to improve the related fault coverage.

Most of the hardware existing in the addressed chunk of logic implements multiplexers (MUXs) and comparators (CMPs). We will address these two types of components separately. Please note that the decoders required to implement the memory are already properly tested by any March algorithm [12].

A. Test of the MUXs

When the size of the BTB memory is not too large, the data output is connected to the selected word through a MUX.

In [13] the authors address the test of MUXs from a functional point of view, and show that independently on its implementation, a MUX can fully be tested with respect to

stuck-at faults if a given set of vectors is applied to its inputs. As an example, we report in Table I the set of test vectors required to test an 8-to-1 MUX with 1 bit parallelism. *Si* denotes a selection signal, *Di* an input data signal, while *Z* is the output. The approach can be easily extended to MUXs of any size.

```
Address                 Instructions

                        ld r2, 0(r0)
                        jalr r2
                        jalr r2
                        jalr r2
            …
                        jalr r2
                        jalr r2

a0:0:zero^k             beqz r1, #8
                        add r2, r2, 4
                        jr 31
            …
a0:1:zero^k             beqz r1, #8
                        add r2, r2, 4
                        jr 31
            …
            …
a0:N-2:zero^k           beqz r1, #8
                        add r2, r2, 4
                        jr 31
            …
a0:N-1:zero^k           beqz r1, #8
                        add r2, r2, 4
                        jr 31
            …
```

Fig. 2. Sequence of instructions implementing the March Element ↓w0

Vector No.	S2	S1	S0	D0	D1	D2	D3	D4	D5	D6	D7	Z
0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	1	0	1	0	0	0	0	0	0	1
2	0	1	1	1	1	1	0	1	1	1	1	0
3	0	1	0	0	0	1	0	0	0	0	0	1
4	1	1	0	1	1	1	1	1	1	0	1	0
5	1	1	1	0	0	0	0	0	0	0	1	1
6	1	0	1	1	1	1	1	1	0	1	1	0
7	1	0	0	0	0	0	0	1	0	0	0	1
8	1	0	0	1	1	1	1	0	1	1	1	0
9	1	0	1	0	0	0	0	0	1	0	0	1
10	1	1	1	1	1	1	1	1	1	1	0	0
11	1	1	0	0	0	0	0	0	0	1	0	1
12	0	1	0	1	1	0	1	1	1	1	1	0
13	0	1	1	0	0	0	1	0	0	0	0	1
14	0	0	1	1	0	1	1	1	1	1	1	0
15	0	0	0	1	0	0	0	0	0	0	0	1

Table I: input vectors for testing an 8-to-1 MUX

The set of patterns reported in Table I corresponds first to a marching "0" and then to a marching "1". Hence, testing this MUX requires executing on the BTB memory the following four March elements: ↕ w0, ↕ (w1, r1, w0), ↕ w1, ↕ (w0, r0, w1). In order to combine the requirements for testing this MUX with those for testing the memory faults, one can adopt for example the March M algorithm [14], whose complexity is 16n, and which guarantees the detection of all linked multiple faults related to the Address Decoder, Single Cells, and Coupling Faults.

For testing both the MUX connected to the address and the one connected to the target field, the March M algorithm can be easily transformed into a sequence of branches executing the required read/write operations, following the transformation rules reported in the previous section.

B. Test of the CMPs

The logic surrounding the BTB memory includes some large comparators that are used to compare the values stored in the address and target fields of a selected BTB entry with the value of the current PC (in the Fetch stage) and with the computed target address (in the Execute stage), respectively.

Theoretically, these comparators can be tested independently on their implementation by resorting to the algorithm proposed in [4], which requires *2m+2* vectors, being *m* the parallelism of each comparator, as shown in Fig. **3**.

This test can be translated into a test program composed of suitably located branches. However, in order to match the required test vectors, these branches should be stored in memory in locations that are highly unlikely to be available (such as the two having the all 0s or all 1s address, respectively), or in locations that are spread over the whole memory space: this would mean that the test program size would practically equal the addressable memory space. While this constraint could be acceptable for manufacturing test (which is performed while the processor is on the ATE), it can hardly be matched for on-line test, where severe constraints on the memory requirements for the test typically exist [15].

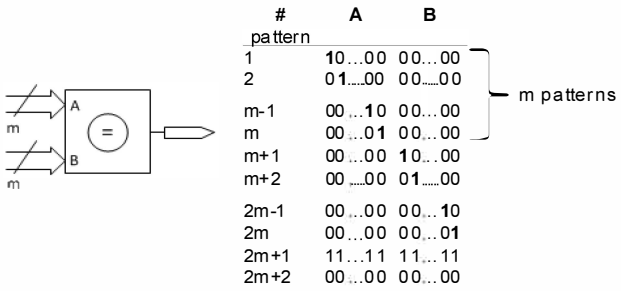

Fig. 3. Test patterns for a CMP.

Unless a Memory Management Unit can freely be exploited during on-line test, the test will only be able to partially test the addressed comparators. In particular, a reduced version of the above algorithm can be proposed, in which the following test vectors are applied:

- X, X, being X whichever value, freely chosen on the basis of the memory area available to store the test program
- $\bar{X}$, $\bar{X}$, being $\bar{X}$ the bit-by-bit complement of X
- A set of vectors including a walking 0 and a walking 1 within the memory area which can be used to store the test program.

In this way, a suitable trade-off between the achievable Fault Coverage and the requirements in terms of code area can be chosen.
The above vectors can be easily translated into the corresponding branch instructions. Once more, the existence of possible faults is detected by looking at the time required by the test program to execute.

V. Experimental Results

In order to validate the proposed approach and to evaluate its cost and effectiveness, we developed a modified version of the DLX processor [2], in which a parametric BTB-based BPU is implemented. The processor model, originally developed at the RT level, is then synthesized with Design Vision V. B-2008.09-SP3 by Synopsys using a generic library.

When implemented with a 16 entries BTB, the total size of the logic surrounding the BTB memory accounts for about 5,000 equivalent gates.

We then implemented different test programs addressing the BTB:

- *Version 1* implements the MATS+ algorithm, thus guaranteeing a 100% fault coverage of the single cell stuck-at faults in the BTB memory, as well as 100% stuck-at fault coverage of the address decoder
- *Version 2* implements the March M algorithm, which guarantees a higher fault coverage with respect to the memory faults, as well as the coverage of the output MUX
- *Version 3* is an improved version of the previous one, which also implements the algorithm for testing the comparators.

Table II reports the main characteristics of the 3 test programs in terms of size and duration. The correctness of Versions 1 and 2 (i.e., their ability to execute exactly the same sequence of read and write operations on the BTB memory mandated by the March algorithm) has been checked by simulation.

Table II also reports the percent stuck-at fault coverage achieved by each of them on the logic surrounding the BTB memory, as computed by Synopsys Tetramax (V. B-2008.09-SP3). When computing the fault coverage, we preliminarily identified the faults that cannot be tested in a functional manner (e.g., those requiring activating the reset signal) and removed them from the fault list, using the technique proposed in [16].

Test program	Size [#instructions]	Duration [#ccs]	FC %
Version 1	402	642	94.34
Version 2	930	2,050	97.86
Version 3	1,278	2,746	98.42

Table II: characteristics of the original and improved test programs

As the figures in Table II show, the proposed method is effective in increasing from about 94% to about 98% the stuck-at fault coverage that can be reached on the logic circuitry surrounding the BTB, which is nearly completely tested when Version 3 is applied.

We are currently investigating the few remaining faults that appear to be connected to the limitations in the memory area that can be used for storing the test program in the implementation of the processor that we are using.

We also verified experimentally that both the test program size and its duration scale linearly with the BTB size.

VI. Conclusions and Future Works

This paper focuses on the functional test of a Branch Prediction Unit based on the Branch Target Buffer architecture. An algorithm for writing a test program for the BTB memory is first provided. The algorithm allows transforming whichever March algorithm into a test program executing the same sequence of read/write operations on the memory, thus achieving the same defect coverage. Secondly, since the previous algorithm can hardly achieve a high stuck-at fault coverage on the surrounding logic, a new one is proposed, which complements the former and improves the final result.

The proposed test algorithm can be effectively adopted for both end-of-production test and on-line test of pipelined and superscalar processors.

Experimental results are provided resorting to the DLX processor model, which has been purposely extended to include a BTB-based BPU. The gathered data allow evaluating the cost (in terms of test code memory area and test duration) of the proposed approach, which grow linearly with the number of entries in the BTB.

Work is currently being performed to explore the possibility of extending the proposed test program to the test of different fault models (e.g., transition delay faults).

References

[1] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda,"Microprocessor software-based self-testing," IEEE Design & Test of Computers, vol. 2, no. 3, pp. 4-19, May-June 2010.

[2] J. L. Hennessy and D. A. Patterson,"Computer Architecture: A Quantitative Approach". Morgan Kaufmann Publishers, 2006.

[3] D. Gizopoulos et al., "Systematic Software-Based Self-Test for Pipelined Processors," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 11, pp. 1441-1453, November 2008.

[4] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, Y. Zorian, "Generic BIST Architecture for Testing of Content Addressable Memories", Proc. of IEEE Int. On-Line Testing Symp., 2011, pp. 86-91

[5] W. J. Perez et al., "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs", Proc. IEEE Int. On-Line Testing Symposium, 2008, pp. 143-148

[6] S. Di Carlo, P. Prinetto, A. Savino, "Software-Based Self-Test of Set-Associative Cache Memories", *IEEE Transactions on Computers,* vol. 60, n. 7, pp. 1030 - 1044 , 2011

[7] E. Sanchez, M. Sonza Reorda, A. Tonda, "On the Functional Test of Branch Prediction Units Based on the Branch History Table Architecture", VLSI-SoC: Advanced Research for Systems on Chip 19th IFIP WG 10.5/IEEE International Conference on Very Large Scale

Integration, VLSI-SoC 2011, Revised Selected Papers. Springer, pp. 110-123. ISBN 9783642327704

[8] P. Bernardi, L. Ciganda, M. Grosso, E. Sanchez, M. Sonza Reorda, "A SBST strategy to test microprocessors' branch target buffer", Proc. IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, 2012. pp. 306-311

[9] 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

[10] 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

[11] L. Ciganda, P. Bernardi, E. Sanchez, M. Sonza Reorda, "An Effective Methodology for On-line Testing of Embedded Microprocessors," in Proc. IEEE International On-Line Testing Symposium, 2011, pp. 270-275.

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

[13] S.R. Makar, E. J. McCluskey, "On the Testing of Multiplexers", Proc. IEEE International Test Conference, 1988, pp. 669-679

[14] V.G. Mikitjuk, V.N. Yarmolik, A.J. van de Goor, "RAM Testing Algorithms for Detection Multiple Linked Faults", Proc. IEEE ED&T Conference, 1996, pp. 435-439

[15] P. Bernardi, L. Ciganda, M. De Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, O. Ballan, "On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors", Proc. IEEE European Test Symposium, 2012, pp. 1-6

[16] P. Bernardi, E. Sanchez, M. Sonza Reorda, O. Ballan, M. Bonazza, "On-Line Functionally Untestable Faults Pruning from Embedded Processor Cores", accepted for publication at the IEEE/ACM Design, Automation and Test in Europe Conference (DATE), 2013