

Low-Cost, Software-Based Self-Test Methodologies for Performance Faults in Processor Control Subsystems

Sobeeh Almkhaizim, Peter Petrov, Alex Orailoglu

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92039
{sobeeh, ppetrov, alex}@cs.ucsd.edu

Abstract

A software-based testing methodology for processor control subsystems, targeting hard-to-test performance faults in high-end embedded and general-purpose processors, is presented. An algorithm for directly controlling, using the instruction-set architecture only, the branch-prediction logic, a representative example of the class of processor control subsystems particularly prone to such performance faults, is outlined. Experimental results confirm the viability of the proposed methodology as a low-cost and effective answer to the problem of hard-to-test performance faults in processor architectures.

1 Introduction

Silicon manufacturing technology evolution has led to a monumental rally in terms of chip size and transistor capacity. As a result, thorough testing of functionality and performance is essential to guarantee product quality. As the competition for features and performance leaves little space for on-chip design-for-test (DFT) hardware, it is essential that reliance be placed on test programs, ideally capable of testing all faults. While such thorough test programs necessitate costly ATE use typically, the processor domain offers the possibility of lower cost test application, as the processor instruction set can be used to apply the test software.

Certain processor subsystems exhibit heavy control characteristics; pseudorandom techniques are ill suited for such control heavy subsystems, while deterministic techniques may be applicable but require further exploration. The situation is complicated further with a number of such subsystems, as they are not only control rich but furthermore are used to enact various forms of speculation in order to boost processor performance. Branch prediction [1], cache management techniques [2] and value prediction techniques [3] belong to this category. By definition, all such speculative techniques are self-correcting functionally, as the speculation needs always to be verified, and, if incorrect, its effects nullified; a faulty behavior typically would also enjoy

the same self-healing capabilities even though performance may suffer significantly.

With the advent of the VLSI technology, a large number of system-on-a-chip designs are steadily becoming the prevailing technology and state-of-the-art in numerous market segments. These devices typically contain several micro-processor cores embedded into them. At the same time, system reliability and dependability are the crucial factors for the market success of such devices. However, the traditional testing techniques are ill-suited for these types of designs, because of the inherent inability or associated high-cost DFT for controlling and observing the deeply embedded processor cores. Consequently, low-cost BIST can provide a high-quality test solution for certain important processor core components, thus making it ever more appealing to the SOC design community.

A quite common subsystem in many current processors, prone to hard-to-detect performance faults, is the Branch Prediction Unit (BPU). The BPU facilitates the process of eliminating the control hazards in a program by predicting the direction of branches. While correctly predicted branches ensure effective processor utilization, incorrect predictions are tolerated by the control recovery mechanism.

A faulty branch predictor is functionally non-differentiable from a correct one. Individual instances of prediction can of course fail, while conversely faulty predictors may end up generating the desired result! Correctness of execution is not affected, as after all a faulty branch prediction unit will correct itself. Yet, as the number of fault manifestations increase, prediction accuracy drops and performance of applications greatly suffers. It is essential to test for such predictors as the consequent Clocks/Instruction (CPI) degradation results in frequent violation of the performance specifications.

A software-based, self-testing methodology for testing processor cores, with an emphasis on the datapath is proposed in [4]. The authors define a methodology for utiliz-

ing test programs that generate and apply test vectors to a particular functional unit under test, overcoming the constraints imposed by the instruction set architecture in terms of arbitrary control signal definition. Bose [5] presents key concepts for generating architectural test cases for performance validation. A fault-free signature, representing high-level parameters that characterize the processor microarchitecture and composed of specific operation completion timings, is generated for various architectural classes and is used as a test case to detect and diagnose performance faults in the design.

We present an approach to test the correctness, particularly against performance faults, of one of the processor control subsystems: the BPU. The proposed testing scheme automates the test by employing the deterministic behavior of the BPU based on a predetermined sequence of branches. We show a fault detection implementation using a special DFT logic, and an alternative solution utilizing a low cost Multiple Input Signature Register (MISR). The experimental results confirm the high fault coverage for both implementations.

2 Algorithm

A branch pattern history is a counter-like structure that accumulates the previous history of the branch. The counter is stored in a *Pattern History Table* (PHT) and indexed via the *Global History Register* (GHR). The GHR contains the direction of the most recently encountered n branches, where n is the length of the GHR in bits. When a new branch is encountered, the GHR is used to index into the PHT, read the counter and make a prediction based on its value. Once the outcome is resolved, the GHR and the counter are updated with the direction of the branch.

Each PHT entry is accessed based on the indexing methodology of the prediction scheme. Most of the branch prediction schemes follow an indexing technique similar to the *Gshare* predictor [6]. In this paper we illustrate the methodology by applying it to *Gshare* only. The extensions to the *Agree* predictor [7] and the *Two-Level* local branch predictor [8] are straightforward; results for all branch predictors are presented in section 3.

2.1 Gshare Fault Detection

Checking the correctness of all the FSM transitions for each PHT entry ensures the validity of the BPU operational logic. The main challenge in the fault detection process is in the way we can access the entries of the table. As we plan minimal DFT modifications, we utilize the GHR, the standard indexing mechanism, to index the table entries. The GHR is updated by shifting in the outcome of the branch being executed. An optimal way would be to visit all the entries that the GHR is capable of addressing, with no redundant checking for an entry. The PHT is traversed by forcing through branch execution a sequence of bits to be

shifted into the GHR. The sequence length would be minimal if each bit in the sequence would result in a value in the GHR distinct from any other visited so far, thus ensuring that all the entries are visited once only.

The inherent GHR behavior motivates the consideration of an analogous structure, namely an external-XOR LFSR, so that by modeling the behavior of this special LFSR the GHR can be controlled by means of software. Such an LFSR, based on a primitive polynomial and having a feedback to the least significant bit position only, controls the GHR by executing a sequence of branch instructions. The condition (direction) of these branches would be defined as the value shifted at the least significant bit of the software-modeled LFSR. The aforementioned mechanism controls the GHR and thus generates unique indices for all the PHT entries in the range of 1 to $2^n - 1$.

The shift value of the software-emulated LFSR, which defines a branch condition in the test program, determines whether to increment or decrement the counter of the entry indexed by the GHR. Since the software-modeled LFSR generates unique indices for all the PHT entries in the range of 1 to $2^n - 1$, the state of the counter for all the entries in the table is *updated*, except for the zero entry. We denote this sequence of indices as a *forward update*. An alternative sequence with the inverse effect of the updates to the table entries, which is denoted as a *reverse update* sequence, can be easily generated. In this case the inverse of the least significant bit of the LFSR is used as a branch direction. Each update effect to an entry is thus reversed; an increment for a counter in the forward sequence will be decremented in the reverse sequence and vice versa.

In Figure 1, we show how to apply the sequences on the FSM of a 2-bit saturated counter; such a counter is typically used to capture the branch pattern history [9]. We start by applying the forward sequence three times to initialize the test, thus putting every entry in either the *strongly taken* or the *strongly not taken* state. The reverse sequence is then applied twice; a counter at S0 will be in S2 while a counter at S3 will be in S1, with both cases resulting in a mispre-

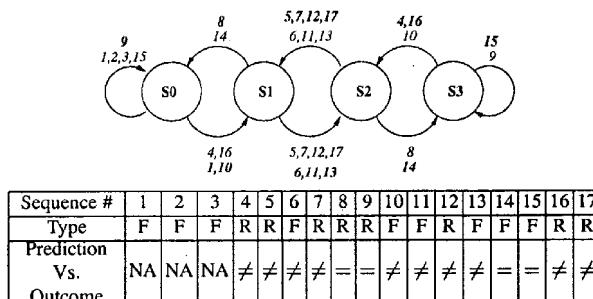


Figure 1. 2-bit Saturated counter FSM. The sequences are numbered to show the sequence effect. Each number corresponds to a sequence number with italics representing the effect for the first half and bolds for the second half.

diction. We continue applying the sequences with the types shown in the second row of the table of Figure 1. The sequences are applied in this order to detect all the possible faulty transitions. Each prediction made depends on the current state, which is reached based on the previous transitions. If the prediction for any entry during these sequences is different than what is expected, then one of the previous transitions is incorrect for that entry.

Along with the 0 entry, which fails to perform the forward update, the $(2^n - 1)^{th}$ entry fails to perform the reverse update. This happens because the entry index contains all 1's and the modelled LFSR cannot proceed with another 1 (to increment the entry), since it will then remain in the same state. If this happens, we effectively visit an entry more than once, thus, not obtaining an exact reverse effect of the forward update. Therefore, the regularity of the pattern, shown in the third row of Figure 1, is lost.

The purpose of the detection hardware, shown in Figure 2, is to compare the outcome from the branch predictor table with the expected value that is previously defined by the testing sequence. As shown in the third row of the table in Figure 1, the sequence of branch predictions and outcomes follows a certain pattern. The hardware in Figure 2 identifies this pattern and generates the test output signal. Once the testing procedure is started, the sequence number in the counter differentiates between a correct prediction and an incorrect prediction. If at any time the output becomes 0, then a fault is detected. The PHT border entries, 0 and $2^n - 1$, are excluded from the test comparison shown in Figure 2.

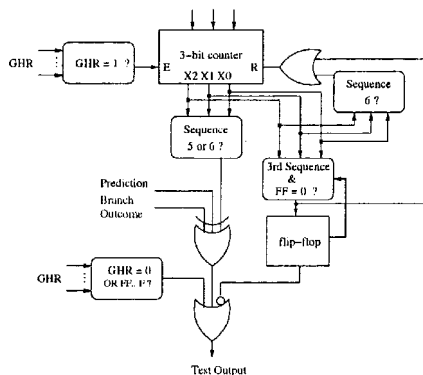


Figure 2. Branch prediction fault detection hardware.

An alternative approach is to use a MISR to compress the responses of the Circuit Under Test (CUT) into a signature. The final signature is compared against a predetermined signature computed under the no fault assumption. A defect in the BPU corrupts the signature and a failure indicator is asserted. While the detection hardware shown above provides complete fault coverage, its cost is rather high compared to a MISR solution; we present experimental evidence to assess the applicability of the MISR and its possible fault coverage loss due to aliasing in section 3.

2.2 Primitive Polynomial Implementation

To automate the production of branches, a software program must be written that produces these branches. The program mimics the behavior of the primitive polynomial by executing branches that follow the polynomial behavior.

```

FORWARD ()
  FOR ( i = 1 to 2^n - 1 ) // For all the sequence
    IF (Primitive(X)) then {} // Shift in GHR
    X << (Primitive(X)) // make X = GHR

REVERSE ()
  FOR ( i = 1 to 2^n - 2 ) // All the sequence except 2^n - 1
    IF (Primitive(X)) then {}
    X << (Primitive(X))
  IF ( false ) then {} // Initialize GHR to a value of 1
  X = 1 // Make GHR & X value = 1
  IF ( true ) then {} // Fake branch
  
```

Figure 3. FORWARD and REVERSE subroutines invoked by PHT.TEST

The branch generation software is composed of invocations to the FORWARD and REVERSE procedures defined in Figure 3. The procedures are invoked based on the order shown in the table of Figure 1. The variable X represents the value of the GHR. Each FOR loop contains an IF statement that generates the branches to be shifted into the GHR. The FOR loop construct generates a sequence of branches covering all the PHT entries. The direction of the branches depends on the primitive polynomial function. The function returns either a 0 or a 1 depending on the bit value to be shifted. The value of X is updated to correspond to the new value of the GHR. In the reverse sequence generation, the direction of the branch is the opposite of what the primitive polynomial function returns. Once all the values in the sequence are generated, the value of the GHR becomes 0. To return the value to 1, another *IF (false)* is used to shift in a 1 in the GHR.

As the FOR loop constructs generate branches that interfere with the branches generated by the IF statement, some extra branches are included, as is the case for the reverse sequence, to make the generated branches exhibit a complete alternation between the FOR loop branches and the IF statement branches. The FOR loop effect can be masked using a simple flip-flop that blocks every other branch in the sequence.

In order to check the 0 and $2^n - 1$ entries, additional sequences have to be applied. The number of additional sequences is 14, as no setup is required. These sequences, even though shorter, do not follow the primitive polynomial. The new sequences follow the FSM transitions for each one of the two entries in the same manner depicted earlier. The hardware now must be able to recognize which entry is being tested and perform the checking for that entry only. Comparing the branch predictor outcome with the expected outcome value becomes more complicated, since

the regularity of the sequence pattern with regard to the expected outcome, shown in the table of Figure 1, is lost.

3 Experimental Results

The complete algorithm, including the PHT border entries, was applied on *Gshare*, *Agree* and *Two-level local* branch predictors. The program counter value was set to zero when used in these predictors to exclude the effect of the position of the branch in the software. A fault was injected in the transitions for random entries. The maximum number of faulty transitions for an entry was set to 1. After applying the algorithm and utilizing the hardware structure in Figure 2, all the faulty entries were detected. The fault coverage of the algorithm was thus confirmed to be 100%. The deterministic behavior of the test is the main reason for achieving this *perfect* fault detection.

In the case of the MISR, the predictions made by the predictor are compressed into the signature. The program used in the MISR implements the basic traversal algorithm, not covering the PHT border cases. Under a single-entry, single-transition fault assumption, we inject a fault in one of the transitions of the FSM of the entry. All possible single faults are injected for all the entries, one at a time. Table 1 provides the percentage of faults detected for various PHT and MISR sizes.

PHT Size	256	512	1024	2048	4096
8-bit MISR	99.09	99.36	99.63	99.75	99.75
16-bit MISR	99.25	99.63	99.81	99.86	99.88
32-bit MISR	99.25	99.63	99.81	99.89	99.93

Table 1. MISR fault coverage.

The achieved results confirm the expected high fault coverage of the proposed approach. All of the fault coverage results exceed 99%. The insignificant drop from a perfect fault coverage is due to the very small amount of MISR aliasing and the incomplete testing of the PHT border entries. The number of undetected faults in the PHT border entries is a small constant regardless of the table size or the number of injected faults, hence the strong monotonicity in the fault coverage values for MISRs with sizes 16 and 32. The percentage of faults undetected is almost reduced by half each time we increase the PHT size, since the number of injected faults gets doubled while the number of undetected ones remains effectively the same. This effect is even stronger for a 32 bit MISR, since the amount of aliasing is virtually zero. It can also be noted that an 8-bit MISR suffices for the observability needs for testing PHTs of practical sizes.

4 Conclusions

In this paper we present a test methodology for deterministic, software-based test for branch predictors in microprocessors. The proposed algorithm deterministically traverses

the branch prediction table and achieves 100% fault coverage if the special DFT is utilized. Alternatively, we show that using a low cost MISR solution for observability purposes produces consistently very high fault coverages in the range of 99%. The proposed test solution utilizes the instruction set of the processor core; thus the test is applied at speed with no need for additional test access pins on the primary inputs and outputs of the microprocessor.

The proposed test methodology captures faults that may remain undetected by conventional functional testing schemes due to the inherent self-correcting ability of the microprocessor's speculative control subsystem. By capturing all these faults in a deterministic manner, we ensure the correct operation of the branch prediction logic and avoid the significant performance degradation that might result from a faulty prediction unit. Furthermore, the compactness of the test program vehicle makes it possible to store and apply this test in a BIST fashion, thus enabling it to capture faults in the device life-cycle. All these important characteristics make the test methodology we propose applicable to a large number of modern microprocessors and complex system-on-a-chip applications and furthermore provide an avenue of research exploration for handling the challenging case of the ever-increasing number of hard-to-test control faults in the state-of-the-art microprocessor architectures.

References

- [1] J. E. Smith, "A Study of Branch Prediction Strategies", in *Proceedings of the 8th ISCA*, pp. 135-148, May 1981.
- [2] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", in *Proceedings of the 17th ISCA*, pp. 364-373, May 1990.
- [3] K. Wang and Manoj Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", in *Proceedings of the 30th MICRO*, pp. 281-290, 1997.
- [4] L. Chen, S. Dey, P. Sanchez, K. Sekar and Y. Chen, "Embedded Hardware and Software Self-Testing Methodologies for Processor Cores", in *Proceedings of the 37th DAC*, pp. 625-630, June 2000.
- [5] P. Bose, "Performance Test Case Generation for Microprocessors", in *Proceedings of the 16th VTS*, pp. 54-59, 1998.
- [6] S. McFarling, "Combining Branch Predictors", Technical Report TN-36, Western Research Laboratory, DEC, June 1993.
- [7] E. Sprangle, R. Chappell, M. Alsup and Y. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference", in *Proceedings of the 24th ISCA*, pp. 284-291, May 1997.
- [8] T. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History", in *Proceedings of the 20th ISCA*, pp. 257-266, May 1993.
- [9] D. A. Patterson and John L. Hennessey, *Computer Architecture: A Quantitative Approach*, ch. 4, Morgan Kaufmann Publishers, Inc, Second Edition, 1996.