Harvesting Wasted Clock Cycles for Efficient Online Testing

Eslam Yassien*, Yongjia Xu*, Hui Jiang*, Thach Nguyen†, Jennifer Dworak*, Theodore Manikas*, Kundan Nepal†
* Lyle School of Engineering, Southern Methodist University, Dallas, Texas, USA
†School of Engineering, University of St. Thomas, Saint Paul, Minnesota, USA

***Abstract*—Mission-critical systems often require some testing to occur while the system is running. In many cases, this involves taking parts of the system off-line temporarily to apply the tests. However, hazards that occur during regular processor execution require the addition of stall cycles to maintain program correctness. These stall cycles generally perform no other function. In this paper, we focus on testing the ALU during those stall cycles to identify new errors or defects that arise during program execution due to aging and increased temperature that may slow down the circuitry or cause permanent defects. We investigate the time to detection of a fault (both stuck-at and transition) that may have caused silent data corruption. In addition, we identify the relationship between the programs running and the list of functional faults and how this impacts the test set length. Finally, we discuss area and performance impacts for the physical implementation of the approach.**

***Index Terms*—field testing, DFT, stall cycles, silent data corruption, online testing, ATPG**

I. Introduction

Mission-critical systems are often required to perform testing at power-up, periodically during circuit operation when one or more parts of the system can be temporarily taken offline, and/or at shutdown. Such test sessions must be highly effective and efficient. Especially during circuit operation, the need to take cores/circuits offline to implement tests that cover the cores/circuits repeatedly means that every dedicated testing clock cycle must be used to its maximum efficiency. Thus, test sets must be short, and the process of applying them must avoid significant performance degradation during normal operation.

Previous researchers have focused on different aspects of on-line testing. CASP (Concurrent Autonomous chip self-test using Stored test Patterns) provides architectural and operating system level support for testing one or more cores during normal operation using test patterns stored in non-volatile memory [1], [2]. VAST (Virtualization-Assisted concurrent autonomous Self-Test) expands on this idea and adds virtualization support to efficiently apply online self-test and diagnostics at the system level [3]. Similarly, Logic Built-In Self-Test (LBIST) and Memory Built-In Self-Test (MBIST) can be used both during manufacturing and in the field. LBIST is commonly implemented using an LFSR (Linear Feedback Shift Register) or another pseudorandom pattern generator to apply a large number of patterns to the circuit [4]–[11]. In these approaches, test results are captured as a signature using a MISR (multiple input signature register) that can be compared against an expected signature.

An alternate approach is Software-Based-Self-Test (SBST), which uses code snippets placed into a cache for test and verification. The focus of previous research in SBST is on the types of code that should be used for test [12]–[20]. When using this approach for field testing, the test procedure is generally reserved for defined test intervals that occur when the device under test can be temporarily taken offline.

To avoid the need to take the device offline for field testing, a related instruction level technique for processor core self-testing was proposed by Shamshiri [21], [22], with updates by Jafari [23]. The technique relied on the compiler to detect data/control hazards and insert TIS (i.e. test) instructions instead of NOPs directly into the code. While this approach addressed the limitations of previous SBST methods, it required a modification of the instruction set. In addition, the method could only be applied to code that had been generated with their proposed technique. Similarly, [24] uses a micro-architectural checkpointing mechanism to create periods of execution during which distributed on-line built-in self-test (BIST) mechanisms check for hardware failures in a 4-wide VLIW processor. For instance an ALU is checked by adding a BIST unit and an additional 9-bit ALU. If a failure is detected, the system operates at a reduced performance level. The approach is able to detect 89% of the stuck-at faults, but they do not consider delay/transition faults in that paper.

To address the limitations of previous research, we have developed a method for the dynamic insertion of tests into an already-running program so that cycles that would otherwise be wasted may be used for field testing. In particular, we show how stall cycles may be replaced with functionality that can be used to check at least some of core's circuitry without:

1) changing the state of the machine,
2) interfering with the execution of the program, or
3) requiring modification of the Instruction Set Architecture (ISA).

Specifically, we explore the detection of stuck-at and transition faults in the ALU, depending upon the number of consecutive stall cycles that are available. We explore different approaches to applying transition test pattern pairs to reduce the time to detection or increase coverage. We also investigate the degree to which the number of test patterns that need to be stored

This work was supported by NSF CCF1812777 and CCF1814928.

can be reduced if only those faults that are identified as likely to be of importance during functional operation are targeted. Finally, we consider the time and area overhead introduced.

II. Architecture

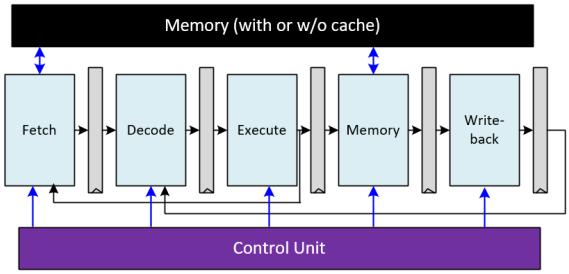

Fig. 1. A simplified view of the 5-stage pipeline.

In this paper, we consider the classic 5-stage in-order RISC-V pipeline shown in Figure 1, consisting of the following stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Data Memory (MEM), and Write Back (WB) [25]. Multiple events could cause stall cycles and NOPs to be inserted into this pipeline. For example, a load instruction followed by an instruction that uses the result of that load results in at least a one-cycle NOP being inserted into the pipeline between the ID and EX stages. For an architecture with cache memory, a miss in the instruction cache may cause multiple NOP cycles to enter the pipeline between the IF and ID stages, while a data cache miss may prevent instructions from advancing and insert multiple NOPs between the MEM and WB stages. Furthermore, mispredicted branches require that instructions along the incorrect branch be flushed from the pipeline so that the program execution is not corrupted.

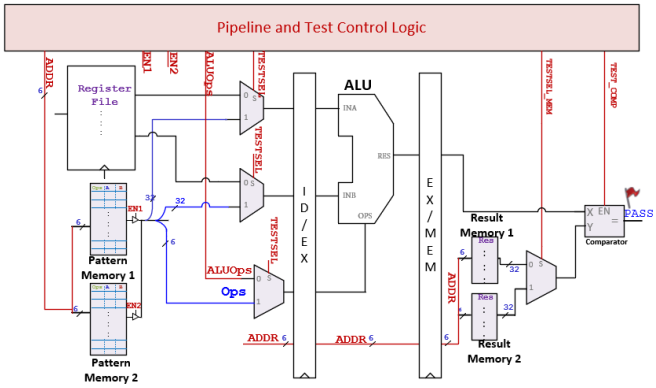

Fig. 2. Additional hardware (shaded) added to facilitate ALU test. All Test control signals are denoted in red. Only portions of the pipeline directly affected by our approach is shown.

In this paper, we explore the addition of limited circuitry as shown in Figure 2, to test the ALU (and possibly other circuitry in the EX stage. The additional circuitry consists of small memory to hold test patterns (inputs to the ALU and responses), muxes to switch to the test patterns when a stall is detected, and a comparator to determine if the test was passed. Additions to the control logic include the address generator and control for enable and select signals for the added circuitry.

In this work, we consider both stuck-at and transition fault detection. Stuck-at fault tests require only a single pattern memory and can help detect permanent defects that arise due to aging (e.g. electromigration that can cause a short between two circuit nodes.) However, transition faults are also important. In addition to aging, environmental variations, such as temperature and voltage, can cause extra delay to appear in the processor during program execution that was not present at power-on. Because stall cycles may appear more frequently than the time that would be required for a test pattern to shift through the chain, we use a second pattern memory to hold an independent observation pattern instead of using a traditional Launch-on-Shift (LoS) approach.

In Figure 2, the extra circuitry used to insert a test pattern is placed in the decode stage, and the inserted muxes are used to switch between the normal ALU operands and control signals and the ones inserted by the test on a stall. Similarly, the comparator and test responses are placed in the MEM stage. We do this so that unacceptable delays from the ID/EX through the ALU to the EX/MEM pipeline registers can be detected more reliably.

III. Fault Targeting and Test Generation

When manufacturing test sets are generated, generally all possible faults belonging to a particular fault model (e.g. stuck-at or transition) are targeted. It is common to find that a large percentage of the faults (e.g. 80%) are detectable with a relatively small number of patterns (e.g. the first 20% of patterns in the test set). The majority of the patterns are needed to detect a relatively small number of "hard-to-detect" faults.

In this work, we are concentrating on tests to be applied in the field, instead of at manufacturing. Furthermore, the tests are to be applied during program execution from a very small memory. Thus, we want to make sure that we are judicious in selecting the faults to target. Intuitively, faults that can actually affect the correctness of the ALU's operation for the running program are more likely to lead to silent data corruption if the problem is not detected. Thus, our first goal is to determine which ALU faults could actually cause errors when different programs are run. This will help determine the amount of test set reduction that is possible.

A. Extracting good machine states

In our experiments, we used the Verilog RTL source code of the B-RISCV 5-stage RV32i Processor core [26]. The Verilog netlist of the processor was synthesized and simulated using the Xilinx Vivado tool-chain. The memory was not synthesized, but was added during the testbench portion of Vivado simulation. The processor was configured with a memory system consisting of no cache. To provide realistic workloads for simulation, C programs were compiled using the B-RISCV compiler tool-chain, and the resulting binary was presented to the Verilog simulation engine. During simulation, at each clock

cycle, the machine state of the ALU—consisting of the inputs and the output of the ALU—were captured together with the processor stall cycle information.

B. Functional Fault Simulation

The machine states from the simulated program execution are saved to a file as "functional patterns." Using a commercial software tool, fault simulation can then be applied to the functional patterns to generate a fault dictionary that records which faults are detected by each pattern.

Once the fault dictionary has been generated for each C program, the list of faults in the ALU that could cause errors in that program can be obtained. For our experiments, the simulations are performed on the ALU circuit design and five different functional programs: Hanoi, Binary-Search, Factorial, Factorial-Fibonacci, and Fibonacci. The number of functional clock cycles required to run each program and the number of those cycles that are stalls are shown in Table I.

TABLE I
NUMBER OF STALL CYCLES IN RISC-V PROCESSOR WITHOUT CACHE.

Program	no. of functional cycles	no. of stall cycles
Hanoi	1518	251
Binary-Search	1990	811
Factorial	107276	59260
Factorial-Fib	128729	62290
Fibonacci	384792	54353

C. Test Set Generation

Automatic Test Pattern Generation (ATPG) is then run on the new fault lists. Thus, the ATPG software generates test patterns with a reduced number of patterns that cover all the faults in the functional fault lists (as compared to the full fault list). We experiment using: 1) stuck-at test sets for stuck-at fault testing, and 2) transition test sets for stuck-at and transition fault testing. As we can see from Table II, the number of test patterns is much smaller than the number of stall cycles, which will allow the test patterns to run multiple times during the program execution. This will allow us to detect problems that may not have been present at the beginning of the program, but that develop during the course of the program's execution itself.

The top rows of Table II show data where the fault lists and test sets are optimized for that particular program. However, in reality, the programs and/or data used for functional analysis will be different from the actual programs run in the field. Thus, the bottom row corresponds to this scenario. The unions of the fault lists for the three programs in the table: Hanoi, Binary-Search, and Fibonacci were used as the final fault lists for ATPG. The size of the fault lists (stuck-at and transition) and the corresponding test set sizes are shown in the row labeled "Cross-program." Later, we will use the "Cross-program" test set to insert tests for the Factorial-Fibonacci and Factorial stall cycles to determine the fault detection characteristics achieved when the program run is not a perfect match for the test set. The last row of the table corresponds to the number of test patterns required when the full fault list is used for ATPG. Clearly, there are significant savings from focusing on the faults that are more likely to be functional. This is not surprising but it shows the magnitude of the savings.

TABLE II
NUMBER TEST PATTERNS AND NUMBER OF FAULTS COVERED BY THEM.

Test Set	no. of stuck-at test patterns	no. of stuck-at faults covered	no. of transition test patterns	no. of transition faults covered
Hanoi	42	1675	43	745
Binary-Search	63	3007	64	1449
Factorial	53	1990	60	1005
Factorial-Fib	56	2033	58	1035
Fibonacci	42	1762	42	770
Cross-program	69	3074	70	1066
Full Test Set	194	6620	230	6620

IV. EXPERIMENTAL APPROACH AND RESULTS

A. Program Runtime Simulation

As described earlier, our approach aims to inject test patterns into the stall cycles of a running program. If a stall is detected in the program, a test pattern is inserted to utilize the empty clock cycle; otherwise, functional operands are inserted for normal operation. We evaluated our approach for each of the C programs described in the last section using the generated test sets and fault dictionaries. Figure 3 shows a simplified flowchart of how we simulated test pattern insertion into stall cycles to determine the ability of our approach to detect developing problems as quickly as possible.

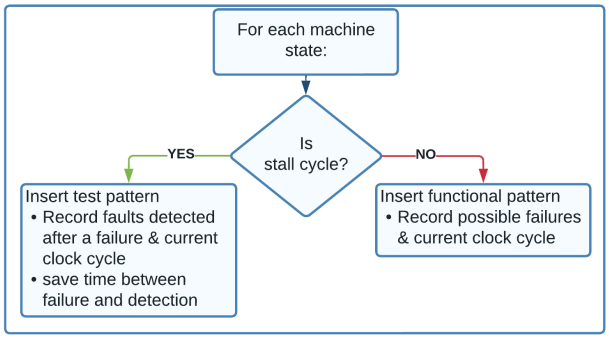

Fig. 3. Flowchart of the program runtime simulation and data collection

Figure 4 shows an example of the detection of node *E* stuck-at-1 by functional and test patterns. Specifically, after we parse the generated test sets, the pattern fault dictionaries, and the extracted program machine states, we iterate over the extracted program machine states. We also record the clock cycle difference between when the fault could have first caused an undetected error in a functional clock cycle and the current stall cycle that detected the fault (e.g. time to detection of 170 cycles for the E stuck-at-1 fault the first time and 12 for the second time).

Because the size of the test sets is well below the number of stalls in the programs, the test sets are applied multiple times; it is possible that a problem could develop during program execution that was not present at the beginning. This is even

more true for transition faults as the circuit heats up and delays increase. As a result, the clock cycle of first functional detection is reset every time the fault is detected by a test pattern. Thus, a new clock cycle difference, i.e. the time to detection, is saved every time a fault is "first" detected by a functional pattern (i.e. since the last test pattern) and then subsequently detected by a test pattern.

```
clk cycles  Faults Detected
1           Asa0, Bsa0, Esa0
2           Asa1, Esa1, Gsa1
3           Ysa0, Zsa0, Esa1
4           :
:           :
:           :
172         stall <- insert test pattern; detects Esa1
173         Asa0, Fsa1, Gsa1
:           :
185         Esa1, Gsa0
:
197         stall <- insert test pattern; detects Esa1

failed_at (Esa1) = 2 // first failure
time_to_detection(Esa1) = 172 - 2 = 170
new failed at (Esa1) = 185
time_to_detection(Esa1) = 197 - 185 = 12
```

Fig. 4. Example of E stuck-at-1 detection by functional and test patterns

B. Results for Stuck-at Fault Detections by Stuck-at Test Sets

Table III illustrates the results of our experiment. The first column of values (Percentage of functional faults detected), shows the fault coverage of our method for each of the five programs we used in our experiments. The top rows of the table show the Program-targeted test sets, while the bottom rows show the Cross-program test set with miss-matched fault lists. Even when the program is different, the Cross-program test set is capable of detecting more than 95% of the functional faults.

Then, we collected data for the time to detection for faults that would cause an error in the program and later be detected by a test pattern. For example, this would include both the 170 and 12 clock cycle times to detection shown in Figure 4. We compute the median of these times to detection for each fault individually. We then take the median of the medians across all faults detected by the test set after a functional error occurs. The computed median of these median times to detection are shown in the last column of Table III.

In the programs, the first stalls occur around the 100^{th} clock cycle; thus, the detections of the first occurrences of many of the faults are around 100 clock cycles. However, as the program continues running, the stall cycles appear more frequently and closely, making the programs' median in the range of 15-33 clock cycles.

Note that even though Factorial and Factorial-Fibonacci were not perfectly matched to the fault list of the Cross-program test set, the median time to detection for those faults that are detected is still relatively low.

Also note that some faults appear for the first time in a functional clock cycle very close to the end of the program and there are not enough stall cycles afterwards to allow for detection by a test pattern. In addition, the last functional clock cycle of the program could have faults that are not detected by subsequent tests. These cases are not included in the calculation of the last column in Table III. To avoid these scenarios, one could simply apply the full test-set at the end of program execution to guarantee coverage.

TABLE III
STUCK-AT FAULT DETECTION

Program-targeted	% functional faults detected	Median time to fault detection (clock cycles)
Hanoi	100	33
Binary-Search	100	17
Factorial	100	15
Factorial-Fib	100	16
Fibonacci	100	31
Cross-program		
Factorial	95.88	18
Factorial-Fib	95.97	25

C. Application to Transition Faults

We also explore the application of our approach to transition faults. In this section, we follow the same steps explained in the previous sections to create the equivalent input files for our program runtime simulation. We generate transition test pattern pairs and the corresponding pattern-faults dictionaries for the transition faults they can detect.

We use the obtained data files to simulate the previously mentioned five C programs. However, in the earlier sections, dealing with stuck-at faults was simpler due to the need for only one stall cycle to evaluate a test pattern for faults. In contrast, with transition faults, we need two consecutive stall cycles to allow us to inject a preconditioning pattern and then an observation pattern. Additionally, when we encounter only a single stall, we need to take the opportunity to apply a test pattern and use it to detect stuck-at faults.

In Figures 5 and 6, we present two different effective approaches for test pattern injection for testing for both transition and stuck-at faults.

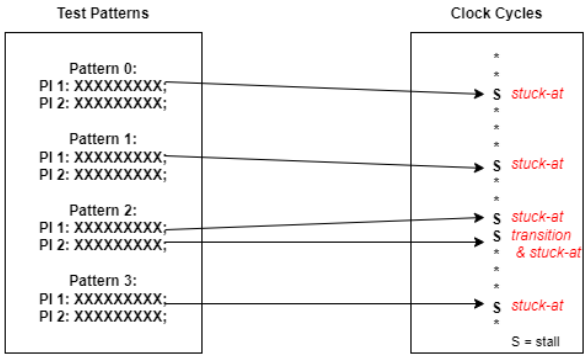

Fig. 5. Case 1: Apply preconditioning patterns at single stalls, one by one

The first approach, or Case 1, is illustrated in Figure 5. In Case 1, when a single stall cycle is encountered, the preconditioning pattern of the next pattern pair is injected and evaluated for stuck-at faults. If the following clock cycle is also

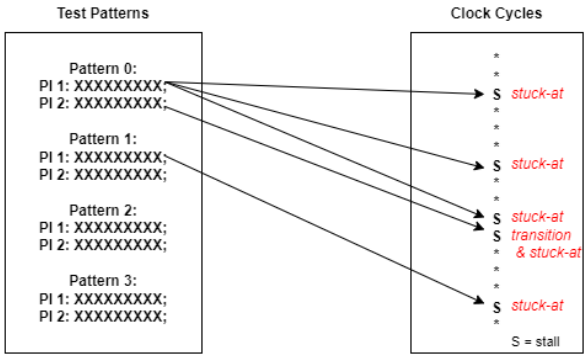

Fig. 6. Case 2: Apply the same preconditioning pattern at single stalls until we get two consecutive stalls and can apply the observation pattern and evaluate for transition faults

TABLE IV
STUCK AT DETECTION BY TRANSITION TEST SETS

	% faults detected		Median time to faults detection (clock cycles)	
Program-targeted	Case 1	Case 2	Case 1	Case 2
Hanoi	97.67	99.22	33	16
Binary-Search	98.80	98.80	23	27
Factorial	98.69	98.69	16	15
Factorial-Fib	99.46	99.46	18	10
Fibonacci	99.04	99.04	32	12
Cross-program				
Factorial	98.89	98.89	16	15
Factorial-Fib	98.92	98.92	22	15

a stall cycle, the observation pattern of the current test pattern pair is then injected, and the full pattern pair is evaluated for transition and stuck-at faults, and the address generator advances to the next pattern pair. Otherwise, if the next clock cycle is not a stall cycle, the address generator advances to the next pattern pair so in the next stall cycle, the preconditioning pattern of a new test pattern pair will be evaluated. This approach allows for testing different preconditioning patterns at single stall cycles but does not ensure testing with all pattern pairs.

The second approach, or Case 2, is illustrated in Figure 6. The main difference between Case 2 and Case 1 is how we deal with single stall cycles. In Case 2, if we do not encounter a consecutive stall cycle to apply the test pattern pair for transition faults, we do not change the advance the address generator. This means that the same preconditioning test pattern will be applied again when the next stall cycle eventually arrives. We only move to the next pattern pair in the test set if we are able to apply both patterns of the pair in two consecutive stall cycles. This approach allows us to ensure that we evaluate all pattern pairs for transition and stuck-at faults if the full test set fits the available stall cycle distribution. However, this generally leads to repeatedly testing for the same stuck-at faults when the same preconditioning pattern is applied because the program includes a limited number of consecutive stall cycles.

D. Results: Transition Test Sets

The results for the Case 1 and Case 2 scenarios are presented in Tables IV and V. Table IV illustrates the stuck-at detection results when we evaluate the application of the generated transition patterns for functional stuck-at fault detections (where the preconditioning and observation patterns are both capable of detecting stuck-at faults). The results show that a significant percentage of functional stuck-at faults were detected with both the Program-targeted and the Cross-program test sets (over 97%). Note that only the functional transition faults were targeted with the test set. As a result, some functional stuck-at faults were not targeted during ATPG, leading to less than 100% coverage. If higher coverage is desired, additional patterns can be generated targeting the missed stuck-at faults and added to the test set.

Case 2 appears to have a higher percentage of stuck-at faults covered than Case 1 for Hanoi. Hanoi has only a small number of two cycle stalls. In the case of a one-cycle stall, Case 2 is designed to ensure that the observation pattern of the pair is applied before we advance to the next pattern pair. The observation patterns are better at detecting stuck-at faults because the preconditioning patterns only need to setup the excitation of the transition faults and do not require the result to be propagated to an output. Not skipping observation patterns (as is done in Case 1) helps to achieve a higher functional stuck-at fault coverage.

The other programs have many two cycle stalls and thus the observation patterns are eventually applied even if they are sometimes skipped. This leads to coverage numbers of Case 1 and 2 to be identical for the other programs and test set combinations. Interestingly we were able to achieve better Cross-program stuck-at fault coverage with the transition test sets over the stuck-at test sets shown in Table III. We believe this is because the transition test sets allow for both patterns in the pair to detect stuck-at faults — roughly doubling the number of patterns applied compared to Table III.

For most of the programs, the median time taken to detect a fault (the median of the median number of clock cycles until detection) is faster for Case 2 than Case 1. Once again this may be due to ensuring the application of the observation patterns so that they are not skipped.

Next, we can examine the transition faults detection results in Table V. Our method achieves a very high percentage of transition fault detection. The Program-targeted test sets are all at 100% coverage except for Hanoi in Case 1, as the iteration through the pattern pairs causes some pairs to never get applied, especially because the double stall cycles are

TABLE V
TRANSITION FAULTS DETECTION

	% faults detected		Median. time to faults detection (clock cycles)	
Program-targeted	Case 1	Case 2	Case 1	Case 2
Hanoi	94.90	100	266	167
Binary-Search	100	100	66	72
Factorial	100	100	104	101
Factorial-Fib	100	100	101	101
Fibonacci	100	100	387	335
Cross-program				
Factorial	93.53	93.53	129	112
Factorial-Fib	93.43	93.43	157	120

scarce in Hanoi. The Cross-program test sets also show good coverage with over 93% coverage of the functional faults even when tested with patterns that target mismatched fault lists. Similar to what we saw earlier, the median time taken to detect a fault (the median of the median number of clock cycles until detection) is faster for Case 2 than Case 1 for most programs. Thus, overall, Case 2 appears to be the better approach to implement.

E. Overhead

To measure the area and performance overhead, we used the OpenLane [27] EDA toolset to convert the Verilog RTL descriptions of the BRISC-V core (without the memory blocks) and map them to standard cells for ASIC implementation using the SkyWater SKY130 PDK [28]. The overall increase in die area was 5.4%, and the overall impact on the clock frequency of the original design was just 1.2%. If the memories or caches that would normally accompany the processor were included in the base die area, then the overhead percentage would be even less.

V. Conclusion and Future Work

In this paper, we explored the insertion of test patterns into stall cycles in a RISC-V processor to detect faults in the ALU. When only the faults that could have caused an error for a program are targeted with the ATPG, the number of test patterns required is much less than a full test pattern set for both the stuck-at test set as well as the transition test set. If a set of applications is perfectly known *a priori* (e.g. as might occur in an embedded system) we achieve very high coverage. We also see that if the application is not perfectly known (e.g. Cross-program examples), we can still achieve reasonably high stuck-at (96% when targeting stuck-at faults and 99% when targeting transition faults) and transition (94%) fault coverage for functional faults.

We also collected data that showed that the time between when a fault could cause an error in the program and when it could be detected by a subsequent test in a stall cycle is quite small. This time is also reduced by the smaller test set size—which allows the patterns to be reapplied during execution of the program.

This work primarily focused on detecting faults in the ALU. Future work will apply the proposed approach that inserts tests during stall cycles to other parts of the processor, such as the other parts of the EX stage, the data memory and other logic in the datapath and/or control.

References

[1] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent autonomous chip self-test using stored test patterns," in *Design, Automation and Test in Europe, DATE*. IEEE, 2008, pp. 885–890.

[2] Y. Li, O. Mutlu, and S. Mitra, "Operating system scheduling for efficient online self-test in robust systems," in *Proc. of Intl. Conf. on Computer-Aided Design*. ACM, 2009, pp. 201–208.

[3] H. Inoue, Y. Li, and S. Mitra, "VAST: Virtualization-assisted concurrent autonomous self-test," in *Proc. Intl. Test Conf.* IEEE, 2008, pp. 1–10.

[4] B. Konemann, G. Zwiehoff, and J. Mucha, "Built-in test for complex digital integrated circuits," *IEEE Journal of Solid-State Circuits*, vol. 15, no. 3, pp. 315–319, Jun 1980.

[5] T. R. Damarla, W. Su, M. J. Chung, C. E. Stroud, and G. T. Michael, "A built-in self test scheme for vlsi," in *Proc.of ASP-DAC'95 with EDA Technofair*. IEEE, 1995, pp. 217–222.

[6] A. S. Abu-Issa, I. K. Tumar, and W. T. Ghanem, "SR-TPG: A low transition test pattern generator for test-per-clock and test-per-scan BIST," in *Intl. Design Test Symposium (IDT)*, Dec 2015, pp. 124–128.

[7] L. Lai, J. H. Patel, T. Rinderknecht, and W.-T. Cheng, "Hardware efficient LBIST with complementary weights," in *2005 Intl. Conf. on Computer Design*, Oct 2005, pp. 479–481.

[8] G. Zeng and H. Ito, "Hybrid BIST for system-on-a-chip using an embedded fpga core," in *22nd IEEE VLSI Test Symposium, 2004. Proceedings.*, April 2004, pp. 353–358.

[9] K. Ichino, T. Asakawa, S. Fukumoto, K. Iwasaki, and S. Kajihara, "Hybrid BIST using partially rotational scan," in *Proc.10th Asian Test Symposium*, 2001, pp. 379–384.

[10] A. Carbine and D. Feltham, "Pentium(R) Pro processor design for test and debug," in *Proc. of Intl. Test Conf. (ITC)*, Nov 1997, pp. 294–303.

[11] ——, "Pentium(R) Pro processor design for test and debug," in *Proc.Intl. Test Conf. 1997*, Nov 1997, pp. 294–303.

[12] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen, and S. Dey, "Embedded software-based self-test for programmable core-based designs," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 18–27, 2002.

[13] C.-P. Wen, L.-C. Wang, K.-T. Cheng, K. Yang, W.-T. Liu, and J.-J. Chen, "On a software-based self-test methodology and its application," in *VLSI Test Symposium, 2005. Proceedings. 23rd IEEE*. IEEE, 2005, pp. 107–113.

[14] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. of the 40th annual Design Automation Conf.* ACM, 2003, pp. 548–553.

[15] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 24, no. 1, pp. 88–99, 2005.

[16] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," in *Processor Design*. Springer, 2007, pp. 447–481.

[17] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[18] P. Bernardi, L. Ciganda, M. De Carvalho, M. Grosso, J. Lagos-Benites, E. Sánchez, M. S. Reorda, and O. Ballan, "On-line software-based self-test of the address calculation unit in risc processors," in *Test Symposium (ETS), 2012 17th IEEE European*. IEEE, 2012, pp. 1–6.

[19] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded processor-based self-test*. Springer Science & Business Media, 2013, vol. 28.

[20] A. Kamran, "HASTI: hardware-assisted functional testing of embedded processors in idle times," *IET Computers & Digital Techniques*, vol. 13, no. 3, pp. 198–205, 2019.

[21] S. Shamshiri, H. Esmaeilzadeh, and Z. Navabi, "Test instruction set (TIS) for high level self-testing of cpu cores," in *Test Symposium, 2004. 13th Asian*. IEEE, 2004, pp. 158–163.

[22] ——, "Instruction-level test methodology for CPU core self-testing," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 10, no. 4, pp. 673–689, 2005.

[23] R. Jafari, E. Z. Salehi, and Z. Navabi, "Utilizing NOPs for online deterministic testing of simple processing cores," in *2015 10th Intl. Conf. on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015, pp. 1–2.

[24] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 73–82, 2006.

[25] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

[26] S. Bandara, A. Ehret, D. Kava, and M. Kinsy, "BRISC-V: an open-source architecture design space exploration toolbox," in *Proc. of the Intl. Symp. on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 306–306.

[27] "OpenLane," Dec. 22, 2022. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenLane

[28] "Skywater PDK," Dec. 22, 2022. [Online]. Available: https://github.com/google/skywater-pdk