Software-Based Diagnosis for Processors

Li Chen and Sujit Dey
Dept. ECE, University of California at San Diego, La Jolla, CA 92093-0407
{lichen, dey}@ece.ucsd.edu

ABSTRACT
Software-based self-test (SBST) is emerging as a promising technology for enabling at-speed test of high-speed microprocessors using low-cost testers. We explore the fault diagnosis capability of SBST, in which functional information can be used to guide and facilitate the generation of diagnostic tests. By using a large number of carefully constructed diagnostic test programs, the fault universe can be divided into fine-grained partitions, each corresponding to a unique pass/fail pattern. We evaluate the quality of diagnosis by constructing diagnostic-tree-based fault dictionaries. We demonstrate the feasibility of the proposed method by applying it to a processor example. Experimental results show its potential as an effective method for diagnosing larger processors.

Categories and Subject Descriptors
B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance.

General Terms
Algorithms, Measurement, Reliability, Experimentation.

Keywords
Microprocessor, self-test, instruction, diagnostics.

1. INTRODUCTION
As microprocessor speed continues to rise beyond the gigahertz range, there is a growing need for at-speed testing. Traditionally, functional testing has been used in industry for applying at-speed tests, in which high-speed, high-pin-count functional testers are used to apply billions of architectural validation (AV) vectors to the device-under-test. As the clock speed, I/O pin count, and length of the AV tests continue to increase, functional testers will become prohibitively expensive, making functional testing economically infeasible. An alternative solution is at-speed scan. Unlike functional testing, scan cannot duplicate the exact operational environment in the normal operational mode, such as the cross-die temperature variation. The difference between the test environment and the normal operational environment may lead to over-testing or under-testing in speed-binning, neither of which is desirable.

In attempt to achieve low-cost and accurate at-speed testing, software-based self-test (SBST) [1]-[3] has emerged as a promising technology for testing performance-critical ICs such as microprocessors. Prior to self-test, test programs are loaded to the on-chip memory by a low-cost tester with low-speed and low-pin-count. The processor then performs self-test by executing the test programs. Upon completion of the test, the test responses or test response signatures can be unloaded from the on-chip memory, again using a low-cost tester. The use of low-cost testers makes SBST an economical solution for at-speed testing, while the application of tests in the normal operational mode enforces the test accuracy.

An additional benefit of SBST is that it can be applied in the absence of scan chains, making it a suitable choice for designs that cannot tolerate scan-induced performance overhead on their critical paths (e.g., high-end microprocessors). The absence of scan chains, however, poses a significant challenge for fault diagnosis.

It is a known fact that the complexity of diagnostic test generation is much higher than that of detection test generation. Though deterministic methods for generating diagnostic tests are available for combinational circuits [4], sequential circuits are much too complex to be handled by the same approach. As a result, currently, the existence of scan chains is a must in order to allow diagnosis of large sequential circuits [5].

Recently, there have been several proposals on generating diagnostic tests for sequential circuits by modifying existing detection tests. In [6], genetic algorithm is used to improve the diagnostic capability of detection tests. In [7], diagnostic tests are generated by eliminating certain vectors in the existing detection test sequence in order to un-detect certain faults. A prerequisite for both methods is a high-coverage detection test set for the sequential circuit under test, as neither method is able to distinguish faults that cannot be detected by the given detection tests. Hence, the success of these methods depends on the success of sequential test generation techniques. Currently, many efforts have been invested in making sequential Automatic Test Pattern Generation (ATPG) techniques feasible for large sequential circuits. However, the applicability of sequential ATPG on complex sequential circuits such as processors is yet to be proven.

As an alternative to sequential ATPG, SBST has been shown to have the potential of successfully generating tests for a particular type of sequential circuits – microprocessors. If properly modified, these tests could possibly achieve a high diagnostic capability. In addition, SBST has a high potential in diagnosis, as functional information is an invaluable resource for guiding and facilitating the generation of diagnostic tests. Our goal in this work is to explore the diagnostic potential of SBST by developing methods for software-based diagnosis.

Previous research in fault diagnosis can be grouped under three categories: diagnostic test generation [4][6][7], diagnosis using realistic fault models [5][8], and fault dictionary management [9][10]. We focused on the generation of diagnostic tests and made the following assumptions during the development of our method. First, it is known that by analyzing test response values rather than pass/fail results, the same diagnostic resolution can be achieved with a smaller test set [6]. For simplicity, we choose to analyze only pass/fail results. This assumption can be eliminated with more sophisticated fault simulation tools. Secondly, accurate diagnosis depends on the use of realistic fault models [8]. Thus, the use of stuck-at fault model may lead to inaccurate diagnosis for bridge faults. On the other hand, it has been reported in [6] that there is a strong correlation between the diagnostic measures for stuck-at faults and for bridge faults. Thus, for the purpose of diagnostic test generation, it is acceptable to evaluate the quality of the diagnostic tests using the single stuck-at fault model.

This work is supported by MARCO/DARPA Gigascale Silicon Research Center (GSRC).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.
Copyright 2002 ACM 1-58113-461-4/02/0006…$5.00.

The rest of the paper is organized as follows. In Section 2, we introduce our methodology for software-based diagnosis, which is based on the careful construction of a large number of diagnostic test programs. We evaluate the quality of diagnosis by constructing diagnostic-tree-based fault dictionaries. In Section 3, we demonstrate the proposed method on a processor core. In Section 4, we draw conclusions and discuss future research directions.

2. METHODOLOGY

We describe our methodology for software-based diagnosis by drawing analogy to traditional fault diagnosis methods, which are based on the application of test patterns.

2.1 Generating diagnostic test programs

In traditional fault diagnosis methods, the basic unit of test is a *test pattern*. We refer to them as *test pattern-based diagnosis*. Given a set of test patterns, two faults can be distinguished if they exhibit different behaviors at the primary outputs (POs).

In the SBST method introduced in [1], test patterns are applied to internal modules of a processor using test programs composed of processor instructions. The test response at the POs of the internal module cannot be checked *immediately* after the application of the test pattern. Rather, the test program captures the module's response and stores it to the on-chip memory for later analysis. Thus, in software-based diagnosis, the basic unit of test is a *test program*. We perform diagnosis by analyzing the test responses corresponding to these test programs, which become available in the on-chip memory after the execution of the test program.

Compared with test pattern-based diagnosis, software-based diagnosis faces new challenges in achieving a high diagnostic resolution, as a test program typically detects more faults than a test pattern.

In test pattern-based diagnosis, a solution to combat low diagnostic resolution is to construct new test patterns that can cause two previously undistinguishable faults to behave differently. If the test patterns are to be generated deterministically, structural analysis needs to be performed to find the right patterns. Otherwise, a large number of fault simulations are needed in order to search for the desired test patterns. Both options are expensive for large sequential circuits.

In software-based diagnosis, with the aid of explicit functional information, it is easier to generate test programs to distinguish certain faults without structural analysis or fault simulation. For example, faults on the data bus may be distinguished by loading and storing specific values; faults in the instruction decode unit may be distinguished by executing different instructions. It should be pointed out that due to the complex structure of a processor, it is impossible to foresee all faults detected by a test program. Thus, given an observed test response, the fault candidates should be determined by fault simulation rather than by speculation. Nonetheless, knowledge of functional information can facilitate the search for the right test program.

In order to achieve a high diagnostic resolution, we propose the use of a large number of carefully constructed diagnostic test programs. Each programs is designed to cover as few faults as possible, while the union of all test programs covers as many faults as possible. To achieve these goals, several principles need to be followed in the construction of these test programs:

1. Reduce the *variety* of instructions in each test program. If possible, use one type of instruction only.
2. Reduce the *number* of instructions in each test program. Each test program contains only the essential instructions for (a) applying one specific test pattern to an internal module of the processor, and (b) observing test responses off the target module by storing them to memory.
3. Create multiple copies of the same test program, which are varied to observe test responses on *different outputs* from the target module. This is to differentiate faults that cause errors to propagate to different nodes.

We illustrate the above principles by comparing detection test programs with diagnostic test programs. The detection test program described in [1] includes subroutines targeting at individual sub-modules of a processor. For example, a subroutine may be targeted at testing the adder logic in the ALU. To increase code density, a loop may be used to execute the ADD instruction with an array of operand values. Thus, the subroutine includes not only the ADD instruction, but also instructions needed for constructing the loop, such as branch instructions, SUB (subtraction) instruction, etc. The resulting detection test program is not suitable for diagnosis due to its large coverage caused by a large variety of instructions and a large number of ADD tests performed. To construct a diagnostic test program that detects as few faults as possible, we follow the principle of using only the essential instructions for applying one ADD test to the ALU. An example test program can be found in [12], where only three types of instructions are used, and only one ADD test is applied. Construction of diagnostic test programs for a processor core following the above three principles can be found in Section 3.1.

2.2 Analysis of diagnostic results using diagnostic tree

Having constructed the diagnostic test programs, the next step is to evaluate the resulting diagnostic resolution, or, how many faults remain indistinguishable from each other.

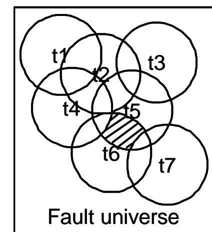

Figure 1. Partitioning the fault universe

Groups of indistinguishable faults can be identified with the construction of a fault dictionary. A fault dictionary entry has two fields, with the first being a unique pass/fail pattern and the second being the group of indistinguishable faults that leads to the pass/fail pattern. Figure 1 shows an abstract view of the dictionary, where each circle represents faults detected by a test program (*t1* through *t7*) and each partition in the fault universe corresponds to a group of indistinguishable faults. E.g., the shaded area corresponds to faults causing both *t5* and *t6* to fail but others to pass.

Through fault simulation, we determine the set of faults detected by each test program. By performing set operations, we determine the fault candidates corresponding to any pass/fail pattern. The same method has been used for test pattern-based diagnosis.

Based on the fact that many pass/fail patterns are impossible due to the correlation among test programs, it is efficient to enumerate only the *possible* pass/fail patterns. Thus, we choose to build the fault dictionary using a diagnostic tree-based approach, in which impossible pass/fail patterns can be detected as early as possible.

Diagnostic tree has been developed through numerous works and is regarded as one of the most space-efficient ways for storing a fault dictionary [9][10]. Here we illustrate the organization of the diagnostic tree we have used and several methods for limiting the tree size.

Given the detection list shown in Figure 2(a), Figure 2(b) shows the corresponding diagnostic tree.

Each tree node is associated with two elements: a fault list and the number of faults in the list. The root node contains the list of all faults. Each level in the tree is associated with a test *T*. The left/right child of a node *N* corresponds to faults in *N* causing the *T* to fail/pass. A fault list no long needs to be divided if it (a) contains no faults, or (b) contains only one fault. In case (a), an impossible pass/fail pattern is detected (e.g., (*t0*,*t1*)=(F,F)). In case (b), we found a pass/fail pattern which can uniquely identify a single fault (e.g., (*t0*,*t1*)=(P,F)). Clearly, only collapsed faults should be considered in order to facilitate the timely termination

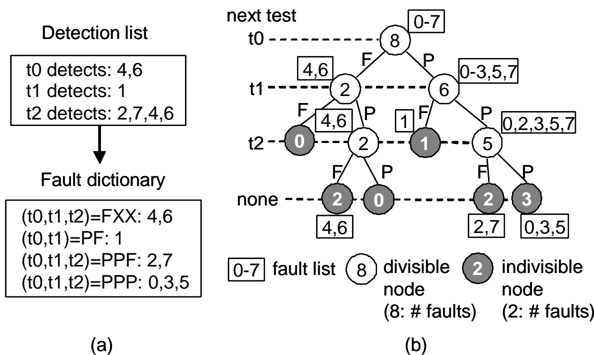

Figure 2. Diagnostic Tree

of tree branches. Each non-zero leaf node in the diagnostic tree corresponds to a group of indistinguishable faults. Their pass/fail response can be determined by tracing back to the root node.

We built the diagnostic tree from the root up using preorder traversal. The pseudo code can be found in [12].

In a deep diagnostic tree, many impossible pass/fail patterns can be detected near the root. Thus, the tree-size is much smaller than that of a full binary tree. In fact, the tree size is bounded by $F\cdot(T+1)$ rather than 2^{T+1}-1, where F is the number of collapsed faults (i.e., the maximum number of leaf nodes in the tree), and T is the number of test programs (i.e, T+1 is the depth of the tree).

The tree size can be further reduced by eliminating *redundant test programs*, which we defined as test programs that cannot further divide the existing fault partitions. Consider the diagnostic tree shown in Figure 2. Let *t3* be a test program detecting faults {1,4,6}. Assuming *t0*, *t1*, and *t2* have been applied, *t3* is a redundant test program.

2.3 CAD framework for evaluating diagnostic test programs

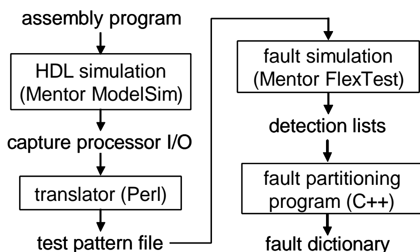

Figure 3. Evaluating diagnostic test programs

To evaluate the quality of a given set of diagnostic test programs, we used the CAD framework shown in Figure 3. To create a detection list for each test program, we first simulate the execution of the test program using an HDL simulation tool. We then capture the I/O signals at the boundary of the processor and translate them into the test pattern format readable by a fault simulator. The detection list of the test program is produced by a full-processor fault simulation. To generate the fault dictionary from the detection lists, we developed a fault-partitioning program based on the diagnostic tree-based method described in Section 2.2. The fault-partitioning program also identifies redundant test programs so as to eliminate them from the final set of diagnostic test programs.

3. EXPERIMENTAL RESULTS

We demonstrate the proposed software-based diagnostic method by constructing diagnostic test programs for a processor core named PARWAN [11]. PARWAN is an 8-bit accumulator-based microprocessor capable of executing 23 different instructions. The synthesized version of PARWAN contains 1785 equivalent NAND gates and 53 flip-flops. The total number of faults is 3940. The number of collapsed faults is 1912. The quality of the diagnostic test programs is evaluated in terms of the size of indistinguishable fault groups in the fault universe.

3.1 Composition of diagnostic test programs

Based on the three principles described in Section 2.1, we constructed 3609 diagnostic test programs for PARWAN. Detailed contents of the test programs can be found in [12]. The composition of the test programs is shown in Table 1. In order to achieve a high diagnostic resolution, each test program aims at applying a specific test pattern to an internal module of the processor using a specific instruction. Thus, we categorize the test programs under module names and instruction names.

Table 1. Diagnostic test programs for PARWAN

Module	Test program types	# test programs
ALU	LDA test	128x5
	ADD test	64x5
	AND test	64x5
	SUB test	64x5
	NOT test	128x5
SHU (Shifter)	ASL test	128x5
	ASR test	128x5
PC	JMP test	64
CTRL	CTRL test	25
	Total	3609

With a three-bit control input, the ALU has six functional modes, corresponding to six instructions: LDA (load from accumulator), STA (store to accumulator), ADD (addition), AND (logic AND), SUB (subtraction), and NOT (inversion). Thus, six different types of test patterns can be applied to the ALU. Note that it is implied that the STA instruction must be used in every test program to store the test response to memory. Thus, we used five types of test programs, corresponding to the other five instructions. In each test program, with control inputs fixed by the instruction type, we apply pseudo random patterns to the data inputs using pseudo random operand values. E.g., 128 operand values are used for LDA tests. We then create 5 copies of each test program, each observing a subset of outputs from the ALU. One test program uses the STA instruction for observing the data output, while the other four combines STA instruction and branch instructions to observe the four status outputs, including *v* (overflow), *c* (carry), *z* (zero), and *n* (negative).

Similarly, two different types of test programs are constructed for the Shifter (SHU), consisting of ASL (arithmetic shift left) tests and ASR (arithmetic shift right) tests.

For the Program Counter (PC), test patterns are applied using the jump instruction (JMP). Each test program jumps from one specific memory location to another. To facilitate the observation of the test response, the test programs are designed in a way such that a particular value is stored to memory *only if* the jump is executed properly.

Finally, 25 test programs are used to partition faults within the Controller (CTRL). For the 19 non-branch instructions, one test program is used to exercise each instruction, with the exception of the STA instruction. For the 4 branch instructions, two test programs are used to exercise each instruction, including a branch-taken case and a branch-not-taken case. By analyzing the state transition diagram, we have concluded that any additional test program will not further partition the faults in the Controller.

3.2 Evaluation of diagnostic test programs

We use the CAD framework shown in Figure 3 to evaluate the quality of the diagnostic test programs. On a Sun workstation with a 300MHz Ultra-SPARC-IIi processor and 128MB of physical memory, the typical CPU time for running HDL simulation and fault simulation for a test program in Table 1 is 1.8 sec and 10 sec, respectively. With 3609 test programs, the total CPU time required is 11.83 hours.

Table 2 shows the quality of the diagnostic test programs. We first divide the test programs into sub groups. We eliminate redundant test programs from each sub groups by running the

fault-partitioning program. We then combine all irredundant test programs and used the fault-partitioning program again to derive the final fault dictionary.

In Table 2, Column 1 shows the names of the sub groups, with "ALL" denoting the union of all irredundant tests extracted from all sub groups. For each group of test programs, Column 2 shows the number of test programs. Column 3 shows the number of irredundant test programs. Column 4 shows the number of indistinguishable fault groups in the fault universe. Column 5/6 shows the number of faults that failed/passed all test programs. Column 7 shows the average size of all fault groups, *excluding* the all-F and all-P groups. Column 8 shows the number of non-zero nodes in the diagnostic tree. Column 9 shows the CPU time required for running the fault-partitioning program.

Table 2. Quality of diagnostic test programs

	# tests	# irre.	# groups	all-F	all-P	ave group size	# tree nodes	CPU time [sec]
ADD	320	72	261	560	813	2.081	28633	22.7
AND	320	47	109	544	981	3.617	9487	8.4
SUB	320	55	231	575	819	2.262	29695	23.8
LDA	640	37	68	524	1081	4.652	14258	14.2
NOT	640	41	108	540	1002	3.491	21165	19.9
ASL	640	58	146	464	955	3.424	34075	29.1
ASR	640	59	126	472	979	3.718	28045	24.5
JMP	64	37	122	654	981	2.308	3323	2.7
CTRL	25	25	174	472	621	4.762	1378	1.0
ALL	**431**	**259**	**746**	**339**	**307**	**1.702**	**57830**	**46.1**

Out of 3609 test programs, only 259 are irredundant. Note that this is different from the total number of tests in the ALL group, as there are redundancies among irredundant tests extracted from different sub groups. The small number of tests implies a small diagnostic tree, as well as short test application time. Note that the set of irredundant test programs depends on the order in which test programs are added to the diagnostic tree.

Given the fault dictionary resulting from all irredundant test programs, we extracted the group size distribution shown in Table 3. For a group size *N*, frequency *F* denotes the number of groups that contain *N* faults. Multiplying *N* by *F*, we obtained the fault distribution shown in Figure 4. Note that excluding the all-F and all-P groups, the majority of the faults belong to groups of small sizes. The average size of these groups, as shown in Table 3, is 1.702.

Table 3. Group size distribution

Group size	1	2	3	4	5	6	7	8	9
Frequency	525	125	54	10	5	10	3	2	4

Groups of size 13, 16, 22, 28, 307, 339: one each
Groups of size 26: 2

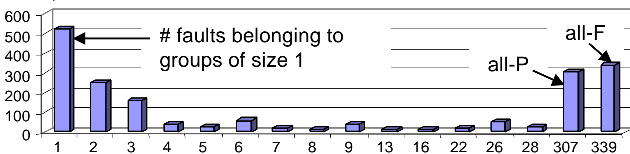

Figure 4. Fault distribution

The all-P group contains faults that cannot be detected by any of the diagnostic test programs. The 307 collapsed faults translate to 382 uncollapsed faults, resulting in a fault coverage of 90.63% out of 3940 faults. This is close to the fault coverage reported in [1] (91.42%), which is that of a detection test program prepared based on the SBST method. As we have discussed in the introduction, the number of diagnosable faults is limited by the number of detectable faults. As reported in [1], on PARWAN, SBST achieved higher fault coverage than sequential ATPG (46.82%) and even full-scan without test points (82.18%).

The all-F group contains faults that are detected by all test programs. A rough inspection of the fault list shows that these faults are located on functionally critical nodes in the processor, such as certain nodes in the instruction decode unit and tri-state buffers controlling the busses. These faults can be further divided by inspecting test response values in addition to pass/fail results. However, it is quite possible that a significant fraction of the all-F faults result in the same empty test response, as their critical location cause the processor's inability to execute *any* instruction. This fundamental limitation indicates that a hybrid approach may be needed for these hard-to-diagnose faults. Nonetheless, an overwhelming majority of faults in a processor (datapath faults and even a larger number of faults in control logic) can be handled by software-based diagnosis, which has the *unique* advantage of enabling low-cost at-speed test.

4. CONCLUSIONS AND FUTURE DIRECTIONS

Though a promising technology for enabling at-speed test with low-cost testers, SBST poses a great challenge for fault diagnosis. To address this problem, we have developed a method for enabling software-based diagnosis, in which a large number of carefully constructed diagnostic test programs are used to divide the fault universe into fine partitions, each corresponding to a unique pass/fail pattern. To demonstrate its feasibility, we applied the proposed method to a processor core. Experimental results show its potential as an effective method for diagnosing larger processors.

The fault simulation time required for deriving the detection lists for all diagnostic test programs is a major concern when scaling the proposed method to large industrial processors. Though the simulation time can be reduced with mixed-level fault simulation and distributed computing, more efficient test generation methods need to be developed in order to reduce the number of diagnostic test programs to be fault-simulated. For example, more deterministic test programs can be used to diagnose programmer-visible faults (e.g., faults on data busses and in register banks) and datapath components with regular structures. In addition, hierarchical approach may be used to reduce the problem size, that is, the number of faults to be considered in one fault simulation.

5. REFERENCES

[1] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Computer-Aided Design*, vol.20, no.3, March 2001, pp. 369-380.

[2] W.-C. Lai, A. Krstic, and K.-T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," *Proc. 18th VLSI Test Symp.*, Montreal, Canada, May 2000, pp. 15-20.

[3] S. Almukhaizim, P. Petrov, and A. Orailoglu, "Low-cost, software-based self-test methodologies for performance faults in processor control subsystems," *Proc. IEEE 2001 Custom Integrated Circuits Conference*, San Diego, CA, May 2001, pp. 263-6.

[4] T. Grüning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: A fast diagnostic test pattern generator for combinational circuits," *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 194-197.

[5] S. Venkataraman and S.B. Drummonds, "Poirot: applications of a logic fault diagnosis tool," *IEEE Design & Test of Computers*, vol.18, no.1, Jan.-Feb. 2001. pp. 19-30.

[6] X. Yu, J. Wu, and E.M. Rudnick, "Diagnostic test generation for sequential circuits," *Proc. Int. Test Conf. 2000*, Atlantic City, NJ, Oct. 2000, pp. 225-34.

[7] I. Pomeranz and S.M. Reddy, "A diagnostic test generation procedure based on test elimination by vector omission for synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol.19, no.5, May 2000, pp. 589-600.

[8] J. Wu and E.M Rudnick, "Bridge fault diagnosis using stuck-at fault simulation," *IEEE Trans. Computer-Aided Design*, vol.19, no.4, April 2000, pp. 489-495.

[9] V. Boppana, I. Hartanto, and K. Fuchs, "Full fault dictionary storage based on labeled tree encoding," *Proc. 14th VLSI Test Symp.*, Princeton, NJ, April 1996, pp. 174-9.

[10] B. Chess and T. Larrabee, "Creating small fault dictionaries," *IEEE Trans. Computer-Aided Design*, vol.18, no.3, March 1999, pp. 346-356.

[11] Z. Navabi, *VHDL: Analysis and modeling of digital systems*, New York, McGraw-Hill, 1993.

[12] http://esdat.ucsd.edu/~lichen/dac2002, additional information related to this paper.