Unambiguous I-Cache Testing Using Software-Based Self-Testing Methodology

Ching-Wen Lin and Chung-Ho Chen
Institute of Computer and Communication Engineering, National Cheng Kung University,
Tainan, Taiwan, R.O.C.
kliolin@mail.ee.ncku.edu.tw and chchen@mail.ncku.edu.tw

***Abstract*—We propose an unambiguous instruction cache software-based self-testing methodology that can generate a reliable result to precisely determine the test passed or not. We present testing cases that cause ambiguous cache testing results and propose five principles of test pattern selection to prevent these situations from occurring. To preserve the order of March sequence in testing an I-cache, we leverage cache bank and cache disable operations. In this way, we are able to implement any March algorithm without violating the sequence order. Finally, we present a case study for ARM v5 ISA processor that has an 8KB instruction cache. We use the March C- algorithm and achieve 100% of inter-word coverage and more than 97% of intra-word coverage evaluated by the RAMSES simulator.**

Keywords—I-cache testing; SBST; March algorithm; ARM

I. INTRODUCTION

Software-based self-testing (SBST) for processor systems is an attractive testing solution due to its non-intrusiveness and flexibility. In SBST, the testing result is usually interpreted from the execution result of the testing program. March algorithms are widely used for memory cell built-in self-testing (BIST) and SBST [1]-[3]. A typical March algorithm consists of special sequences of March elements that are memory operations applied to the memory cells by specific addressing orders. However, an I-cache cannot be tested by SBST easily because the I-cache is usually regarded as a read-only device from the viewpoint of the program execution. The first issue to resolve is how to give the testing patterns which at the same time are also valid instructions. This is a common problem for testing I-cache in SBST since it is not always possible to find the instruction encoding that satisfies a test pattern and an instruction at the same time. Despite that test patterns can be made available from instructions, simply determining testing result bases on the outcome of instruction execution might not be reliable enough. As an example, if instruction "AND R0, R0, R0" is selected as a test pattern which is read into the I-cache, the program examines the result of R0 to see if this instruction memory has passed the test or not. Assume that the faulty bits have changed the instruction to an "OR R0, R0, R0" instruction. Examining R0 in this case, we cannot tell if the fault has occurred or not. We define the testing result of this nature as ambiguous. In this paper, we present a method to prevent these ambiguous situations and obtain reliable testing result.

Another aspect in testing I-cache is how to minimize the impact of program execution on the order of the March sequence since the deviated March sequence in memory testing may result in the lower fault coverage. In testing an I-cache line by line, it is required to ensure that the last word of a target cache line should be also tested. Nevertheless, when a pipelined processor is executing the last instruction from a cache line, the next cache line will be fetched into the cache. The addressing order of the March sequence is broken at the same time. To prevent this from occurring, we leverage a common cache operation, cache disable that changes the next instruction is fetched from a non-cacheable memory reference. In this way, we are able to maintain the order of the required March sequence and obtain the expected fault coverage.

This work is supported by the National Science Council, Taiwan, under Grant NSC 102-3113-P-006-018.

Testing cache using SBST has been proposed by lots of researches. S. D. Carlo [2] has tried to overcome the difficulties of the set-associative cache testing. They deal with the problems for data cache memory successfully; however, there is no detailed discussion on testing the intra-word faults of the I-cache. The other work, G. Theondorou [3], has proposed the direct cache access instructions (DCA instruction) that can read and write the data array of the I-cache. Unfortunately, not every ISA has the DCA instructions that can fully support cache testing. In this paper, we present an unambiguous I-cache SBST methodology without DCA instructions, and we discuss three ambiguous testing models and propose five principles of test pattern selection to prevent these ambiguous situations from occurring.

The rest of this paper is organized as follows. Section II describes the implementation of the March algorithm. Section III presents our proposed unambiguous principles. Section IV provides a case study and the experimental result. Finally, we make a conclusion for this paper.

II. TRANSLATE MARCH ALGORITHM INTO SBST PROGRAM

A March test is a sequence of March elements composed of pre-defined read or write accesses to the tested memory cells by the special addressing order. The values of March read or write are called data backgrounds (DBs) or their complementary data backgrounds ($\overline{\mathrm{DBs}}$) [4]. After one March element has been applied to a memory cell, the element should then be applied to the next one according to the specified addressing order which can be ascending ($\Uparrow$), descending ($\Downarrow$), or either ($\Updownarrow$). Fig. 1 shows that the word-oriented March C- algorithm which contains six March elements with the write/read order specified from left to right. The least number of the required data backgrounds, including the complementary, to test the intra-word faults of an L-bit memory cells (word) is defined as follows:

$$\{2 * (\lceil \log_2 L \rceil + 1)\} \text{ --- Eq. (1)}$$

$$\Updownarrow(wDB);\ \Uparrow(rDB,\ w\overline{DB});\ \Uparrow(r\overline{DB},\ wDB);\ \Downarrow(rDB,\ w\overline{DB});\ \Downarrow(r\overline{DB},\ wDB);\ \Updownarrow(rDB)$$

Fig. 1. Word-oriented March C- algorithm

A. *March data backgrounds*

Our SBST concentrates on the data array of the I-cache. Before we introduce how to find March elements that are also valid instructions, we will illustrate the data array banking model with Fig. 2. The cache banking is a common technique to reduce the access time and dynamic power [5], [6]. Fig. 2(a) shows a single bank data array which consists of eight instructions, and Fig. 2(b) shows an eight-bank data array with interleaved instruction storage. Different banking models will impact the length and the content of one data background.

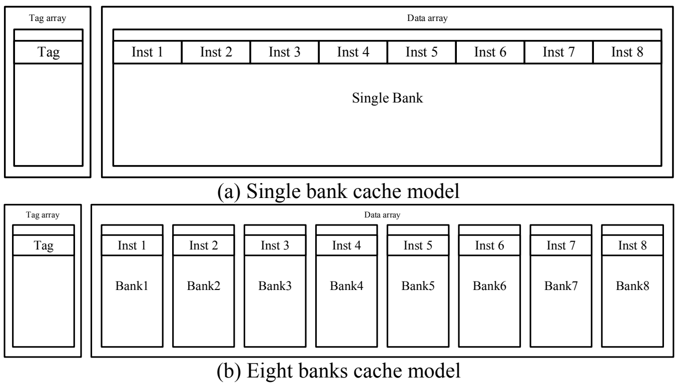

Fig. 2. Cache banking models

For the single bank model, the length of the March data background is the line size, i.e., 256 bits, and at least nine pairs of March data backgrounds is required to test the intra-line faults according to Eq. (1). This implies that there are eight instructions per March data background. For March data backgrounds and the complementary, we need to find more than 144 valid instructions. This number makes the translation difficult if not impossible. To resolve this difficulty, we choose the implementation of the eight-bank data array. For this cache banking model, the data background is 32 bits, the same length of an instruction. According to Eq. (1), if all the data backgrounds can be regarded as valid instructions, we only need to find twelve valid instructions.

B. *SBST program establishment*

We have introduced a feasible method to reduce the number of March data backgrounds. In this section, we present the translation between the March algorithm and the SBST program. The crucial requirements for our SBST program development are: (1) having a non-cacheable memory region, and (2) having a privilege instruction which can disable cache access even the memory region is cacheable. These two features can be found in modern microprocessors.

We divide the testing program into four segments: an initial segment, a sequence control segment, a test pattern segment, and a result verification segment. As illustrated in Fig. 3, these four segments have different memory attributes: cacheable or non-cacheable, which is defined by the program allocation model of our SBST. The memory management unit decides which region can be cached or not by these attributes. Fig. 3 also shows our SBST program execution flow. The sequence control segment determines which line of the test patterns to be fetched into the I-cache. The sequence of fetching the test patterns must conform to the addressing order of the specified March operations. The sequence control segment for a direct-mapped cache is straightforward by using the linear index when addressing a cache line. For an N-way set-associated cache, it is required to have additional operations to modify the replacement policy state in order to obtain the specified addressing order [2].

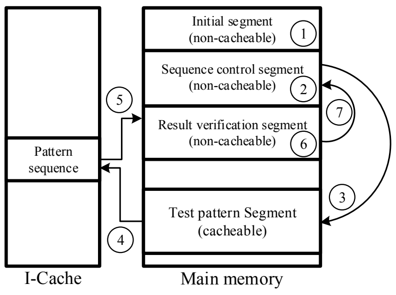
Initial segment (non-cacheable) 1
Sequence control segment (non-cacheable) 2
Result verification segment (non-cacheable) 6
Test pattern Segment (cacheable)
3
4
5
7
Pattern sequence
I-Cache
Main memory

Fig. 3. Program segments for I-Cache SBST

The key segment transition is how to jump to the result verification segment from the test pattern segment without bringing in unnecessary instructions into the I-cache. We notice that the pipeline architecture will fetch instructions until the jump instruction is actually executed. This may violate the order of the March sequence since the instructions following the jump are written into another cache line.

For example, a classical five-stage pipeline processor finishes the jump operation at the decode stage, and there is a redundant instruction fetched in the fetch stage. This redundant instruction might violate the read/write order of the March sequence and degrade the fault coverage. Considering the characteristic of the pipeline and the cache banking model, we employ the disable cache instruction to accomplish the segment jump operation without violating the specified March sequence.

Due to the pipeline architecture, there are several instructions fetched from the I-cache before the disable cache instruction is executed. We name these instructions as the "follower." The number of follower, I_F is depended on the processor architecture, i.e., the pipeline depth.

As Fig. 4(a) shows, assuming we test the cache lines in the first bank, and the first bank is also the entry point from the sequence control segment. To conform to the specified March sequence, the first requirement is that

$$I_F < (I_L - I_{BR} - 1) \text{ --- Eq. (2)}$$

where I_L is the number of instructions per line, I_{BR} is the number of instructions per row in a bank. In this example, testing the second bank uses the similar test pattern layout as in Fig. 4(a).

Fig. 4(b) shows the test pattern layout in a cache line when testing the last bank. For this situation, we must use the follower instructions to serve as the data background. Notice that the disable cache instruction is allocated at the left neighboring bank of the target bank. After the follower instructions are executed by the processor, the jump instruction to be executed is fetched from the main memory since the cache is disabled. Nevertheless, at the time of cache line filling, the jump instruction for testing last row is also brought into the cache along with the rest of the cache line. To test the last bank, the requirement is given as follows:

$$I_{BR} <= I_F \text{ --- Eq. (3)}$$

To test the entire cache line, the above two requirements must be met at the same time:

$$I_{BR} <= I_F < (I_L - I_{BR} -1) \text{ --- Eq. (4)}$$

Testing bank 3 and bank 4 in this example uses the similar test pattern layout.

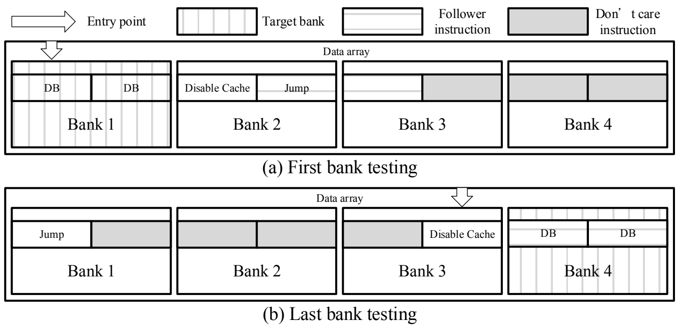

Fig. 4. The test pattern layout

III. UNAMBIGUOUS INSTRUCTION CACHE TESTING

We have described how to generate a March sequence conformed SBST program. As mentioned previously, an SBST testing result may be ambiguous if only interpreting the register value and memory content for test verification. In this section, we elaborate the causes of the ambiguity and propose the principles of test pattern selection to eliminate them.

A. *Ambiguous factors*

In SBST, any instruction that has the same encoding as a data background can be chosen as a candidate test pattern. There are various reasons for the occurrences of ambiguity, and we classify them into three categories:

Factor 1: Many instructions may produce the same result.

For example, an ADD instruction, assuming it is a data background, can be changed into a SUB instruction due to the faulty cache. Simply interpreting the destination register value cannot determine whether the test is passed or not. This situation often occurs on data-processing type of instructions, especially bit-operation instructions.

Factor 2: Many memory words may have the same values.

For example, the contents of R1 and R3 point to an individual memory word respectively, but these two memory words have the same values. In the register-indirect addressing mode, “Load R0, [R1]” produces the same result as the “Load R0, [R3]”. When either instruction is the data background, interpreting the result of R0 cannot tell which instruction is executed.

Factor 3: Repeated instructions may conceal the faults

The repeated instructions may generate ambiguous results even if they can produce unambiguous results individually. In the eight-bank model, a carefully-chosen “Load R0, [R1]” could test each bank row alone without producing the ambiguity. Nevertheless, in the four-bank model, this instruction must be repeated twice for each bank row. In this case, we cannot discover the faulty bits of the first word in the bank row since this error is concealed by the second run.

B. *Principles to eliminate the ambiguity*

We have introduced three reasons that cause the ambiguity in testing result. We propose five principles in test pattern selection. These principles serve as the base guidelines that are applicable to the ISA of modern processors.

(1) Choose a unique value for the source register.

A unique value of the source register can imply a non-reproducible result, especially for data-processing type instructions. Before applying this instruction for testing, the test program should reset the rest of registers to zero so that there is no room for ambiguity to occur.

(2) Prohibit the instructions that have undefined bits.

Some instructions have the undefined bits because of the encoding. Those bits are usually ignored at the decoder stage, so the failures of these bits could not be propagated by the normal functions of the processor. Therefore, the instructions that have undefined bits should be prohibited to be the test patterns.

(3) Clear the target memory and program status register before testing.

A clean target memory and status register can reflect the test result of the chosen test pattern. This principle is suitable for every type of instructions especially for load/store instructions. These clean operations can be finished at the sequence control segment shown in Fig. 3.

(4) Employ the encoding of an undefined instruction as the destination register value of a load instruction or the result to be stored for a store instruction.

To prevent Factor 2 from occurring, to use uncommon values for load/store instructions is a good method. The undefined instructions cannot be generated by the tool chain, so they can be exclusive in the memory.

(5) Select either post-indexed or pre-indexed addressing instructions.

To distinguish testing results of multiple instructions in a bank row, using an addressing instruction that has auto-increment mode can tell the difference of each testing result in a bank row.

We apply the above five principles in test pattern selection for testing the I-cache of the ARMv5 ISA processor.

IV. CASE STUDY: ARM V5 PIPELINE PROCESSOR

In this section, we demonstrate the unambiguous SBST test program development on the direct-mapped I-cache of an ARM v5 ISA pipelined processor. We concentrate on the data array testing of the I-cache that is physically implemented as eight 32x256 SRAM banks, and the method for testing the tag array of the I-cache can refer to the work in [1].

A. Experimental environment

We have implemented an ARM-compatible processor and its direct-mapped I-cache by Verilog. For our experiment, we simulate the processor running the testing program and log the read/write signals of the I-cache, and then we apply these signals as the input patterns to the RAMSES simulator [7] to perform the RAM fault simulation. We implement the March C-algorithm that can effectively detect all SAF, TF, AF, CFst, CFin, and CFid faults. TABLE I shows the relationship between the March data backgrounds and our testing instructions selected based on the above five principles.

According to Eq. (1), the minimal number of test patterns is 12 for our design. We are only able to find one pair of the test patterns that directly match with two valid instructions. For the rest of test patterns, we select the instructions that have the best match, i.e., minimum Hamming distance.

A data background that matches a valid instruction might not be used as a test pattern due to its ambiguity in test result. For example, the March data background "0x00000000" can be mapped directly into an ARM instruction "ANDEQ R0, R0, R0." In this case, the test result is ambiguous by examining R0. To remedy this, we modify this instruction into "ANDEQ R0, R0, R1." In this way, the data background used to test is only one-bit away from the specified one. We explore this method to select the rest of the test patterns. We also use extra test patterns to improve the final fault coverage. As the result, we choose seven pairs of instructions for our data backgrounds which are listed in TABLE I.

TABLE I. MARCH DATA BACKGROUND TRANSLATION

Minimal pattern set	Chosen DB (instruction)	Minimal pattern set (complementary)	Chosen $\overline{\text{DB}}$ (instruction)
00000000	00000001 ANDEQ R0,R0,R1	FFFFFFFF	EA7FFFFF B 0x1FFFFFC
01FFFE00	01DEEE00 BICSEQ R14,R14,R0,LSL #28	FE0001FF	E71000EE LDR R0,[R0,-R14,ROR #1]
03FC03FC	03DC03FC BICSEQ R0,R12,#0xF0000003	FC03FC03	E703EC03 STR R14,[R3,-R3,LSL #24]
E1E1E1E1	E1D1E1E1 BICS R14,R1,R1,ROR #3	1E1E1E1E	151E1E1E LDRNE R1,[R14,-#0xE1E]
66666666	66466666 STRBVS R6,[R6],-R6,ROR #12	99999999	91999989 ORRSLS R9,R9,R9,LSL #19
55555555	55555555 LDRBPL R5,[R5,-#0x555]	AAAAAAAA	AAAAAAAA BGE 0xFEAAAAA8
	EA7FFFFF B 0x1FFFFFC		E7EEEFEE STRB R14,[R14,R14,ROR #31]

B. Experimental results

We compare our results with S. D. Carlo [2] for the data array of the I-cache. In [2], they use a two-way set-associative I-cache with 32 sets and eight-word per cache line. We change the banking model into eight-bank model which is the same as our setting for the comparison. We perform the perfect March C- algorithm using their proposed patterns and use the RAMSES [7] to evaluate the fault coverage. TABLE II shows the results of the fault coverage. Our proposed method has performed much better on intra-word testing, AF, CFst, CFid. We also observe that using the test patterns selected for testing intra-word faults has performed well for the rest of the fault types. Even with the DB selection principles which limit the freedom in selecting a test pattern, we show that testing an I-cache using our proposed SBST is a viable approach for high quality testing.

TABLE II. DATA ARRAY FAULT COVERAGE LIST

		[2], 8KB	Our proposed, 8KB
	Fault count	Coverage	Coverage
SAF	131,072	100%	100%
TF	131,072	100%	100%
AF (inter-word)	534,773,760	100%	100%
AF (intra-word)	2,031,616	51.41%	100%
AF (total)	536,805,376	98.53%	100%
CFst (inter-word)	2,139,095,040	100%	100%
CFst (intra-word)	8,126,464	50%	97.83%
CFst (total)	2,147,221,504	98.48%	99.99%
CFin (inter-word)	1,069,547,520	100%	100%
CFin (intra-word)	4,063,232	100%	100%
CFin (total)	1,073,610,752	100%	100%
CFid (inter-word)	2,139,095,040	100%	100%
CFid (intra-word)	8,126,464	50%	97.05%
CFid (total)	2,147,221,504	98.48%	99.98%

TABLE III shows the breakdown of instruction distribution for our SBST program. We observe that around 50% of the instruction count is used to ensure unambiguous testing results.

TABLE III. SBST INSTRUCTION DISTRIBUTION

	Instruction count	Percentage
Initial segment	71,532	1.51 %
sequence control segment	1,355,200	28.60 %
test pattern segment	2,288,647	48.34 %
result verification segment	1,019,552	21.53 %
Total	4,734,931	100.00 %

V. CONCLUSION

This paper has addressed the problems of testing an instruction cache using SBST. We point out that a testing result can be ambiguous if only interpreting the resultant value of the test program in register or memory. We propose five principles of test pattern selection to prevent ambiguous results from occurring. To keep March sequence in the specified order, we leverage the common cache operations, i.e., disable cache, and the pipeline architecture to properly place the test patterns. Finally, we provide a case study using a pipeline processor and show that high quality I-cache SBST can be achieved for all kinds of memory fault types.

REFERENCES

[1] Y.-C. Lin, Y.-Y. Tsai, K.-J. Lee, C.-W. Yen, and C.-H. Chen, "A Software-Based Test Methodology for Direct-Mapped Data Cache," Asia Test Symp. ATS'08, pp. 363-368, 2008.

[2] S. D. Carlo, P. Prinetto, and A. Savino, "Software-Based Self-Test of Set-Associative Cache Memories," IEEE Trans. on Computer, vol. 60, no. 7, pp. 418-423, 2011.

[3] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, "Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures," IEEE Trans. on Very Large Scale Integration systems, vol. 21, no. 4, pp. 786-790, 2013.

[4] A.J. van de Goor, I.B.S. Tlili, and S. Hamdioui, "Converting March Tests for Bit-Oriented Memories into Tests for Word-Oriented Memories," in Proc. Of Int. Workshop on Memory Technology Design and Testing, pp. 46-52, 1998.

[5] T. V. Kalyan and M. Mutyam, "Word-Interleaved Cache: An Energy Efficient Data Cache Architecture," in AMC/IEEE Int. Symp. on Low Power Electronics and Design, ISLPED'08, pp. 265-270, 2008.

[6] Y. K. Cho, S. T. Jhang, and C. S. Jhon, "Selective Word Reading for High Performance and Low Power Processor," in Proc. Of the 2011 ACM Symp. on Research in Applied Computation, RACS'11, pp. 25-30, 2011.

[7] C.-F. Wu, C.-T. Huang, and C.-W. Wu, "RAMSES: a fast memory fault simulator," in Int. Symp. On Defect and Fault Tolerance in VLSI Systemsm pp. 165-173, 1999.