

On the Generation of Functional Test Programs for the Cache Replacement Logic

W. J. Perez H.^{1,2}, D. Ravotto³, E. Sanchez³, M. Sonza Reorda³, A. Tonda³

¹ Universidad del Valle
Grupo de Bionanoelectrónica
Cali, Colombia
wjperzh@univalle.edu.co

² Universidad Pedagógica y
Tecnológica de Colombia,
Grupo Gira
Sogamoso, Colombia
wjperzh@uptc.edu.co

³ Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
{danilo.ravotto, ernesto.sanchez,
matteo.sonzareorda, alberto.tonda}
@polito.it

Abstract— Caches are crucial components in modern processors (both stand-alone or integrated into SoCs) and their test is a challenging task, especially when addressing complex and high-frequency devices. While the test of the memory array within the cache is usually accomplished resorting to BIST circuitry implementing March test inspired solutions, testing the cache controller logic poses some specific issues, mainly stemming from its limited accessibility. One possible solution consists in letting the processor execute suitable test programs, allowing the detection of possible faults by looking at the results they produce. In this paper we face the issue of generating suitable programs for testing the replacement logic in set-associative caches that implement a deterministic replacement policy. A test program generation approach based on modeling the replacement mechanism as a Finite State Machine (FSM) is proposed. Experimental results with a cache implementing a LRU policy are provided to assess the effectiveness of the method.

I. INTRODUCTION

Today's processing systems, based on high-performing processor cores, elaborate high amounts of data by executing a very large number of instructions per second. It is essential to provide the whole elaboration system with a memory hierarchy subsystem as cheap and fast as possible, in order to obtain a good system performance and low production costs [1].

One of the most important elements within the memory system is the cache memory. In the memory hierarchy scheme, memory caches are placed very close to the processor core, and they are directly devoted to reduce the gap between fast and high-performing processors cores and slow memory devices.

Nowadays, most of the integrated circuit (IC) manufacturing cost is brought about by test and validation processes. Moreover, test methodologies do not progress at the same pace the manufacturing technology does.

The previous considerations are particularly valid regarding the first level of memory hierarchy systems, i.e., caches, since there is not a mature enough methodology able to cope with all testing issues.

Different approaches have been proposed to deal with cache testing in general. On the one side, hardware-based

approaches usually require considerable modifications to the initial design in order to support testing procedures. For example in [2], the authors propose a structural modification of the cache architecture to improve the IDDQ testing sensitivity. In [3], authors include a Memory Built-in Self-Test (MBIST) device capable of applying an improved March algorithm to L1 and L2 caches present in the processor core.

Algorithm-based approaches, on the other side, normally propose a direct transformation of March-like tests into processor instructions in order to test mainly the data part of cache memories [4], [5]. The main drawback of these methodologies is the requirement of special system features to facilitate main memory writing and reading operations while the cache memory is disabled; such enabling/disabling mechanisms may not be present in the processor normal operation mode. In [15], the authors adapt a traditional march tests to test the data part of a multi-way set-associative cache.

Even though March-like methods have been demonstrated efficient regarding memory elements in caches, these strategies do not adequately deal with the test of the control part of the cache, despite the importance of this part in the correct behavior (and expected performance) of the processor.

An interesting strategy that exploits suitable test programs without additional hardware requirements is Software-Based Self-Test (SBST) [6], which can be seen as an extension of the work on functional test of processors presented in [7]. This strategy exploits the processor ISA instructions to execute suitable test programs while the processor works in the normal mode of operation; the results produced by the program are checked with respect to the expected ones in order to detect possible faults. This approach provides some benefits with respect to hardware-based approaches: in particular, it does not require any change in the processor hardware, and it can be used in a stand-alone device as well as when the processor core is deeply embedded in a SoC and its accessibility is reduced.

Moreover, being based on a functional approach, SBST allows performing test application at speed, thus increasing the achieved defect coverage.

Finally, the SBST approach does not require high cost test equipments for its application, and can even be used for on-line test purposes.

Clearly, the success of the SBST approach strongly depends on the availability of effective test programs addressing the different parts of the processor; in the case of cache controllers this issue is particularly critical, due to the limited accessibility (both in terms of controllability and observability) we normally have on the cache controller input and output signals.

SBST strategies for cache memories have already been proposed in the literature: in [16], the authors propose a transformation of a March algorithm to executables instruction sequences with some improvements in order to take into account the control logic in the cache; however, the proposed methodology only tackles direct-mapped caches. In [8], a general SBST algorithm able to generate test programs for data cache controllers is presented, which does not depend on the mechanisms implemented by the controller. In [9] a hybrid solution for testing data and instruction cache controllers is reported: the methodology exploits an external Infrastructure Intellectual Property (I-IP) to improve the result observability.

This paper presents an algorithmic-based strategy to test the replacement logic in set-associative caches that implement a deterministic replacement policy. The methodology is suitable for post-production testing, for incoming inspection, and for the on-line test of both stand-alone processors and processor cores embedded in SoCs. The proposed algorithm can be tailored to different cache configurations, both in terms of cache size and organization, and in terms of writing strategies (i.e., write-back and write-through). The methodology is based on modeling the behavior of the replacement mechanism of the cache as an FSM. Thus, no hypothesis on the actual implementation of the replacement mechanism is necessary and the methodology considers the replacement circuitry as a “black box” where only the basic hit/miss behavior is monitored. The proposed algorithm works on the FSM model and by performing a sequence of carefully selected operations is able to fully excite the FSM model by traversing each one of its transitions at least once and then checking whether the target transition has been correctly performed. The operations performed on the FSM model are then translated into a suitable test program. A major advantage of the proposed approach is that it not only successfully applies to all possible implementations of the replacement mechanism, but it also faces all possible faults that may affect it. To experimentally evaluate the real performance of the proposed approach, we implemented it in assembly code and we gathered fault simulations results for a pipelined processor whose cache controller implements the LRU approach.

The rest of the paper is organized as follow. Section 2 introduces some vocabulary, recalls some concepts about FSMs and introduces the FSM testing. Section 3 describes the proposed approach. Section 4 presents the case study and reports some experimental results. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Testing Finite State Machine

By adopting the terminology of [17], a finite state machine (FSM) M is a quintuple $M=(I,O,S,\delta,\lambda)$ in which:

- I is a finite and nonempty set of input symbols
- O is a finite and nonempty set of output symbols
- S is a finite and nonempty set of states
- $\delta: S \times I \rightarrow S$ is the state transition function
- $\lambda: S \times I \rightarrow O$ is the output function.

An FSM in a state s in S that receives an input a in I moves to the state specified by $\delta(s, a)$ and produces the output specified by $\lambda(s, a)$.

An FSM that follows the previous definition is fully specified if for every state and for all the inputs there is a specified transition to a next state and a specified output defined by the output function. The FSM is deterministic if in every state there is a unique transition to a next state for all possible inputs. The FSM is strongly connected if for every pair of states s_i and s_j there is an input that takes the machine from s_i to s_j .

FSM testing can be performed using different strategies; however, the general problem is to deduce information from a generic machine M by observing its I/O behavior. Testing functional faults in an FSM can be done by following a conformance testing procedure, where a specification machine A and an implementation machine B (whose implementation details are not known, i.e., seen as a “black box” from which it is only possible to observe the I/O behavior) are supposed to be available. The conformance test aims at generating a test sequence to determine whether B correctly implements the specification machine A , independently on its implementation and on the possible faults. The basic skeleton of a conformance testing is described in the following two steps:

An initialization sequence is applied bringing the implementation machine, in the case it is correctly implemented, to a known initial state.

A transition tour is performed, in order to verify if every transition in the specification machine is correctly implemented in the black box device, and it reaches each time the expected state.

It is important to note that conformance testing guarantees that any fault producing a misbehavior in the FSM is detected no matter its implementation. In order to correctly determine whether the reached state after every transition is the expected one, it is possible to resort to a status check [17] strategy: a sequence of input values that produce a unique output string for each state of the FSM.

B. Cache replacement policy

A cache replacement policy, or cache algorithm, is a major design parameter of any memory hierarchy. The efficiency of the replacement policy affects both the hit rate and the access latency of a cache system. The higher the associativity of the cache, the more vital the cache algorithm becomes. A great number of replacement policies exist, and

each one is a compromise between hit rate, cost (in terms of required hardware resources) and performance: Least Recently Used (LRU), Most Recently Used (MRU), Pseudo-LRU, Least Frequently Used (LFU), and many others.

Further details of the LRU policy are presented, since the case study provided in section 4 exploits a cache implementing this replacement policy. The LRU cache algorithm aims at decreasing the cache miss ratio in set-associative caches by replacing the block that has not been used for the longest time when a cache miss occurs. Clearly, the replacement logic must also update at each cache access a correct ordering among the block elements based on their time in cache, independently if the cache access produces hit or miss.

The logic that implements the LRU algorithm, during a memory access, faces three different situations:

- The processor accesses to the most recently used block (MRU), therefore no one block is changed in cache and the current ordering is maintained
- The address requested by the processor is not stored in the cache; thus, a cache miss occurs and the least recently used block (LRU) must be substituted with the requested one, guaranteeing the correct reordering of the complete set of blocks
- The block requested by the processor is currently stored in the cache. Then, the accessed block becomes the most recently used (MRU), requiring a reordering of the complete set of blocks, while no block is substituted.

In literature, different implementations have been proposed for devising this replacement mechanism; the most representative ones exploit counters, stack architectures, priority queues, and reference tables [17].

III. PROPOSED APPROACH FOR TEST PROGRAM GENERATION

Herein, a deterministic algorithm able to generate sequences of instructions for testing the replacement mechanism of the cache controller is presented and discussed. As mentioned before, the physical implementation of a replacement mechanism can be accomplished resorting to different structures such as counters, stack, priority queues, etc.; it is essential, then, to define a high level strategy able to address the whole spectrum of implementations, without considering the final one. For this reason, we choose to model the behavior of the replacement mechanism of the cache controller as an FSM, and then, we propose the implementation of an algorithm composed of a series of well selected FSM transitions able to thoroughly excite the replacement mechanism, based only on the modeled FSM. It is also important to note that considering the real implementation of the replacement mechanism as a “black box”, we can only observe an essential element of its behavior, i.e., the cache hit/miss signal. Finally, the obtained transition sequence is carefully converted to a suitable test program, coping with the specific characteristics of the processor core and its cache.

A. Testing Issues

Once again, it is important to highlight that this paper does not focus on the test of the data part of caches (as in [11]), but only on the replacement mechanism of the cache controller. The adoption of SBST strategies implies the generation of specially crafted programs that perform several memory operations carefully selecting the effective address of every memory access.

Checking whether the operations performed by the test program produced the expected results (thus revealing the presence of a faulty circuit) is made more complex by the fact that many faults in a cache controller do not manifest themselves causing the processor to produce wrong results, but simply slowing down its performance. Therefore, the test procedure requires the availability of some mechanism able to verify whether a given access (or a sequence of them) produces a cache hit or miss as expected. This can be achieved, for example, by measuring the time required by the processor to execute a given piece of code [8], or resorting to a hybrid technique, as described in [9]. For the purpose of this paper, we assume that one of such mechanisms is available. In the rest of the paper we will refer to this mechanism as *cache monitor*.

B. Replacement mechanism modelling

In this paragraph we first present the procedure for building the FSM model of the replacement mechanism for only one cache set; clearly, the cache behavior is the same for every cache set belonging to it.

Following the definition of a generic FSM M reported in paragraph 2.1, our FSM has the following sets of input symbols, output symbols and states:

- The set I of input symbols consists of all the addresses accessed by the processor. The number of possible input symbols is strictly dependent on the size of the main memory. Although the set of input symbols is very large, for our purpose we consider only two groups of symbols. The first one contains the n addresses currently stored in the n -ways of the cache set: each one of them, when accessed, produces a “hit”. The second one includes all the remaining addresses (apart from those currently stored in the cache), each one of which produces a “miss” when accessed and activates a transition towards a new state, which is the same no matter the address chosen.
- The set of output symbols O is composed of the two symbols *hit* and *miss*, which are the only output that can be produced and observed by the cache controller logic for our purpose: the output of our FSM is the hit/miss signal in the cache controller that is checked using the cache monitor implemented.
- The set of states S is built considering that at each time instant the behavior of the replacement module in the cache controller depends on the current order of the ways according to the replacement policy. Therefore, considering a generic n -way cache and denoting each way as w_1, w_2, \dots, w_n , each possible state corresponds to a

permutation of w_1, w_2, \dots, w_n , where the rightmost way is the first one that will be replaced, while the leftmost is the last one. In an n -way set-associative cache, each state has $n+1$ outgoing transitions: n transitions represent a memory access to one of the n blocks stored in the cache set, therefore producing a hit (and possibly changing the ordering of ways). The remaining outgoing transition represents an access to a memory block not memorized in the cache, therefore producing a miss.

The FSM corresponding to the replacement logic in a cache controller is fully specified: in a given state and upon every input there is a distinct next state. It is also deterministic, since at a state and upon an input, the FSM follows a unique transition. Finally, the machine is strongly connected, since it is possible to demonstrate that given a couple of states s_i and s_j , it is always possible to find a sequence of memory accesses that lead the FSM from s_i to s_j .

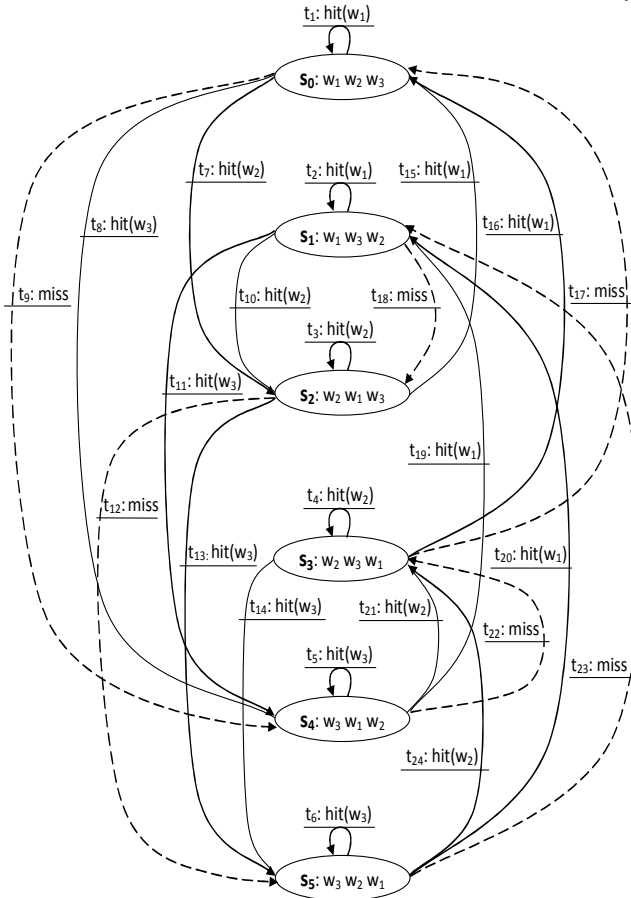


Figure 1. FSM model for a 3-way set associative cache

The generation of the FSM model is straightforward and follows the rules of a deterministic replacement policy. Moreover, the generation can be easily automated, thus limiting the human effort required.

To provide the reader with an example, we reported in Figure 1 the FSM of a 3-way set associative LRU cache. The generation of the model follows the rules of the LRU replacement strategy reported in section 2.2. The FSM has 6

states (s_0, s_1, \dots, s_5) and every state is determined by the current order of the ways (w_1, w_2, w_3) within the set. According to the previously introduced notation and the behavior of the LRU replacement policy, the leftmost way in the set is the most recently used (MRU), while the rightmost one is the least recently used (LRU). For this model, the extracted FSM counts on 24 transitions (t_1, t_2, \dots, t_{24}). In the figure, every transition is labeled with the accessed way (in the case of a hit) and the output it triggers (hit or miss).

C. Replacement mechanism testing

In this sub-section we outline the algorithm for generating the test sequence for a generic n -way set associative cache that implements the LRU policy. The same approach can be followed for developing the test algorithm for cache controllers implementing different replacement mechanisms.

Our objective is to generate a test sequence able to test the replacement mechanism of a cache, starting from the FSM that defines the behavior of the replacement mechanism. For this purpose, we generate a test for the FSM model following the conformance testing approach outlined in section 2.1 and adapting it to the replacement mechanism of the cache controller. In our conformance testing problem, we have the complete information of a specification machine, the FSM model of the replacement mechanism, and an implementation machine that is a black box that only allows us to observe its I/O behavior by reading the hit/miss signal, this machine is the actual hardware. Thus, we generate a sequence of memory accesses able to guarantee that all the transitions in the specification machine are traversed at least one time. At the end of every transition, we verify the reached state using a special sequence of memory accesses that is able to check whether the reached state is the expected one. This special sequence of transitions is described in the following.

The problem of generating a transition tours is known in literature as the *Chinese Postman Problem* (CPP) [13], and it aims at finding the shortest tour such that every edge is traversed at least once. To generate the sequence of addresses needed to traverse all the transitions in the FSM we use the Floyd-Warshall algorithm described in [14]. In order to verify the correctness of the reached states, we propose a particular sequence of memory accesses that, exploiting the cache monitor, is able to indirectly identify the current state of the FSM. As detailed later, during the application of the identifying sequence, the current state of the machine changes, but at the end of this sequence, the FSM is led back to the original state. Assuming that w_1, w_2, \dots, w_n represent the ways of a generic n -way set associative cache (with w_1 the MRU and w_n the LRU) and A_1, A_2, \dots, A_n the addresses of the blocks stored in each way, the proposed sequence works as follow:

- Firstly, we perform an access to an address (A_{n+1}) not present in the cache; the purpose of this operation is to remove the block with address A_n placed in the way w_n from the cache.

- Secondly, we access to the address A_n which has just been removed from the cache and check if the access causes a miss (as expected) or a hit. Clearly, if the access causes a hit a fault affects the cache.
- Third, supposing that the previous operation performed correctly, and removed from the cache the address A_{n-1} stored in the way w_{n-1} , then, we access again to the just removed block, in this case the one with address A_{n-1} , provoking again a miss condition. If this is not the case, the reached state is not the correct one. We repeat this operation for all the remaining ways, by accessing the block that should have been removed at the previous access and checking the cache miss behavior. These access transitions should all provoke miss.
- Finally, we perform $n-1$ accesses to the addresses stored in cache, provoking the same number of hit results, in order to lead back the FSM to the original state.

Figure 2 shows the behavior of the cache when applying the sequence of cache accesses described above for a 3-way set associative cache. The state to be verified is w_1, w_2, w_3 (with w_3 the LRU) and the ways contain respectively the addresses A_1, A_2, A_3 . In the figure MA denotes a memory access to the address specified between the parentheses.

Current State	<table><tr><td>A₁</td><td>A₂</td><td>A₃</td></tr><tr><td>w₁</td><td>w₂</td><td>w₃</td></tr></table>	A₁	A₂	A₃	w₁	w₂	w₃	
A₁	A₂	A₃						
w₁	w₂	w₃						
MA (A₄)	<table><tr><td>A₄</td><td>A₁</td><td>A₂</td></tr><tr><td>w₃</td><td>w₁</td><td>w₂</td></tr></table>	A₄	A₁	A₂	w₃	w₁	w₂	Miss
A₄	A₁	A₂						
w₃	w₁	w₂						
MA (A₃)	<table><tr><td>A₃</td><td>A₄</td><td>A₁</td></tr><tr><td>w₂</td><td>w₃</td><td>w₁</td></tr></table>	A₃	A₄	A₁	w₂	w₃	w₁	Miss
A₃	A₄	A₁						
w₂	w₃	w₁						
MA (A₂)	<table><tr><td>A₂</td><td>A₃</td><td>A₄</td></tr><tr><td>w₁</td><td>w₂</td><td>w₃</td></tr></table>	A₂	A₃	A₄	w₁	w₂	w₃	Miss
A₂	A₃	A₄						
w₁	w₂	w₃						
MA (A₁)	<table><tr><td>A₁</td><td>A₂</td><td>A₃</td></tr><tr><td>w₃</td><td>w₁</td><td>w₂</td></tr></table>	A₁	A₂	A₃	w₃	w₁	w₂	Miss
A₁	A₂	A₃						
w₃	w₁	w₂						
MA (A₃)	<table><tr><td>A₃</td><td>A₁</td><td>A₂</td></tr><tr><td>w₂</td><td>w₃</td><td>w₁</td></tr></table>	A₃	A₁	A₂	w₂	w₃	w₁	Hit
A₃	A₁	A₂						
w₂	w₃	w₁						
MA (A₂)	<table><tr><td>A₂</td><td>A₃</td><td>A₁</td></tr><tr><td>w₁</td><td>w₂</td><td>w₃</td></tr></table>	A₂	A₃	A₁	w₁	w₂	w₃	Hit
A₂	A₃	A₁						
w₁	w₂	w₃						

Figure 2. Example for a 3-way cache

From figure 2 the reader should note that the last state is equal to the first state (w_1, w_2, w_3) but the addresses are stored in different ways. This does not prevent the continuation of the FSM testing since our algorithm takes care of storing the

current association between ways and addresses in order to correctly perform the status check.

On the other side, figure 3 shows the behavior of the cache when a faulty transition occurs and the reached state is not the expected one. Suppose that the target state of a transition is w_1, w_2, w_3 (with w_3 the LRU) but for any reason the implementation machine wrongly goes to the state w_2, w_1, w_3 . As in the previous figure, MA denotes a memory access to the address specified between the parentheses. This example clearly shows that during the third memory access “MA(A₂)” the cache behavior is not the expected one, since in the second memory access, the address evicted from the cache is A_1 instead of A_2 .

Wrong State	<table><tr><td>A₂</td><td>A₁</td><td>A₃</td></tr><tr><td>w₂</td><td>w₁</td><td>w₃</td></tr></table>	A₂	A₁	A₃	w₂	w₁	w₃	
A₂	A₁	A₃						
w₂	w₁	w₃						
MA (A₄)	<table><tr><td>A₄</td><td>A₂</td><td>A₁</td></tr><tr><td>w₃</td><td>w₂</td><td>w₁</td></tr></table>	A₄	A₂	A₁	w₃	w₂	w₁	Miss OK
A₄	A₂	A₁						
w₃	w₂	w₁						
MA (A₃)	<table><tr><td>A₃</td><td>A₄</td><td>A₂</td></tr><tr><td>w₁</td><td>w₃</td><td>w₂</td></tr></table>	A₃	A₄	A₂	w₁	w₃	w₂	Miss OK
A₃	A₄	A₂						
w₁	w₃	w₂						
MA (A₂)	<table><tr><td>A₂</td><td>A₃</td><td>A₄</td></tr><tr><td>w₂</td><td>w₁</td><td>w₃</td></tr></table>	A₂	A₃	A₄	w₂	w₁	w₃	Hit KO
A₂	A₃	A₄						
w₂	w₁	w₃						

Figure 3. Example for a 3-way cache with a wrong transition

IV. CASE STUDY

The effectiveness of our approach has been experimentally evaluated on the data cache controller of the LEON2 processor [12]. The LEON2 processor is a 32 bit processor conform to SPARC V8 architecture with a 5 stages integer pipeline. The LEON2 processor has separate instruction and data caches that implement a highly configurable multi-set mechanism. It is possible to define the number of ways (from 1 to 4), the set size (1 – 64 Kbyte/set) and the line size (16 – 32 bytes per line). The data cache implements the write through policy, with write no-allocate on a write miss. The replacement policies implemented are Random, LRU and Least Recently Replaced (LRR).

To prove the effectiveness of the proposed algorithm at gate level, we synthesized the data cache controller of the LEON2 microprocessor using a 3-way set associative model and a LRU replacement mechanism. The data cache controller has been synthesized using a generic home-developed technologic library. Table 1 shows details about the synthesized gate-level descriptions of the cache controller, detailing the size of the block implementing the LRU replacement mechanism

TABLE I. LEON2 CACHE CONTROLLER GATE-LEVEL INFORMATION

	Gates	S@ faults
Cache controller	8,023	26,148
LRU algorithm	3,754	11,637

A tool was developed to automatically generate the FSM associated to a generic n -way set associative LRU cache. The Floyd–Warshall algorithm was then applied to obtain the input sequence needed to activate every transition in the FSM and to activate the status check of each state after the firing of a new transition. Both functions were implemented in about 600 lines of code in Java language. Afterwards, a 300-lines Perl program translated the input sequence into assembly code for the LEON2 microprocessor core. The assembly program generated for the 3-way set-associative LRU cache counts 1,940 lines of code and its machine version occupies 7,708 bytes, requiring about 571 K clock cycles to be executed.

Targeting the stuck-at fault the proposed methodology achieved 100% stuck-at testable fault coverage for the replacement logic of the data cache controller of the LEON2 microprocessor.

In order to provide the reader with a reference value, we have also applied a March C- algorithm to the data part of the same cache; the implemented program counts 612 lines of code, its machine version occupies 2,516 bytes, requiring about 217 K clock cycles to be executed. Fault simulation results on the same replacement logic circuit, using the program that implements the March C- algorithm, show a stuck at fault coverage of about 89%. The previous number clearly shows that a March test is not enough to achieve good results on the replacement logic, thus demonstrating the need for specific solutions for its test: the approach proposed in this paper achieves a full fault coverage, which does correspond to 100% testable fault coverage, as mentioned above.

V. CONCLUSIONS

In this paper we proposed a generic testing method for the circuitry implementing the replacement mechanism of a cache controller. The methodology is based on the FSM model of this circuitry and exploits it to produce a sequence of carefully selected memory operations able to fully excite the replacement mechanism and to make its behavior visible from the outside. The sequence of operations generated for the FSM model are then translated into a test program compliant with the processor and cache implemented. Being based on an abstract model of the addressed circuitry, the approach is completely independent on its actual implementation, and is able to detect any kind of fault forcing the circuitry to behave differently than expected.

The effectiveness of the proposed methodology has been experimentally evaluated on a 3-way set associative cache that implements the LRU replacement policy. Fault simulation results clearly show that the proposed methodology is able to fully test the circuitry implementing the replacement mechanism.

We are currently working on reducing the complexity of the proposed algorithm in order to tackle high associativity caches, while avoiding an exponential growth in the number of steps required to adequately cover the FSM.

REFERENCES

- [1] John L. Henessey & David A. Patterson, "Computer Architecture", 3th edition, Morgan Kaufmann publishers. 2003.
- [2] S. Bhunia, Li Hai, K. Roy, "A high performance IDDQ testable cache for scaled CMOS technologies", IEEE Asian Test Symposium, 2002, pp. 157-162.
- [3] P. J. Tan, Le Tung, Mantri Prasad, J. Westfall, "Testing of UltraSPARC T1 Microprocessor and its Challenges", IEEE International Test Conference, 2006, pp. 1-10.
- [4] Sultan M. Al-Harbi, Sandeep K. Gupta, "A Methodology for Transforming Memory Tests for In-System Testing of Direct Mapped Cache Tags", 16th IEEE VLSI Test Symposium (VTS '98), pp. 394-400.
- [5] J. Sosnowski, "In system of cache memories", Proc. IEEE International Test Conference, 1995, ITC pp. 384-383.
- [6] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis, "Software-Based Self-Testing of embedded processors", IEEE Transactions on Computers, Vol 54, issue 4, pp 461 – 475, April 2005.
- [7] S.Thatte and J.Abraham, "Test Generation for Microprocessors", IEEE Transactions on Computers, vol. 29, no. 6, pp. 429-441, June 1980.
- [8] W. J. Perez H., J. Velasco Medina, D. Ravotto, E. Sanchez, M. Sonza Reorda, "Software-Based Self-Test Strategy for Data Cache Memories Embedded in SoCs", Proc. 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS), 2008.
- [9] W. J. Perez, J. Velasco-Medina, D. Ravotto, E. Sanchez, M. Sonza Reorda, "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs", Proc. 14th IEEE International On-Line Testing Symposium, 2008.
- [10] A. J. Van de Goor, "Testing Semiconductor Memories, Theory and Practice", John Wiley & Sons, 1991.
- [11] A. J. Van De Goor, "Using March Tests to Test SRAMs", IEEE Design & Test, Vol. 10, issue 1, 1993, pp. 8 – 14.
- [12] Aeroflex Gaisler AB, <http://www.gaisler.com/>
- [13] J. Edmonds, E.L. Johnson, "Matching, Euler tours and the chinese postman", Mathematical Programming, vol. 5, pp.88-124, 1973.
- [14] R. W. Floyd, "Algorithm 97: Shortest path", Communications of the ACM, v.5 n.6, p.345, June 1962.
- [15] S. Alpe, S. Di Carlo, P. Prinetto, A. Savino, "Applying March Tests to K-Way Set-Associative Cache Memories," *European Test, 2008 13th*, vol., no., pp.77-83, 25-29 May 2008
- [16] Yi-Cheng Lin, Yi-Ying Tsai, Kuen-Jong Lee, Cheng-Wei Yen, Chung-Ho Chen, "A Software-Based Test Methodology for Direct-Mapped Data Cache," *Asian Test Symposium, 2008. ATS '08. 17th*, vol., no., pp.363-368, 24-27 Nov. 2008
- [17] D. Lee, M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol.84, no.8, pp.1090-1123, Aug. 1996
- [18] Jim Handy, "The cache memory book", 2th edition, Morgan Kaufmann publishers, 1998.