Functional test generation for the pLRU replacement mechanism of embedded cache memories

Functional Test Generation for the pLRU Replacement Mechanism of Embedded Cache Memories

W.J. Perez H.[2-3], E. Sanchez[1], M. Sonza Reorda[1], A. Tonda[1], J. Velasco Medina[2]
[1] Politecnico di Torino, Torino, Italy
[2] Universidad del Valle, Cali, Colombia
[3] Universidad Pedagógica y Tecnológica de Colombia, Sogamoso, Colombia

Abstract

Testing cache memories is a challenging task, especially when targeting complex and high-frequency devices such as modern processors. While the memory array in a cache is usually tested exploiting BIST circuits that implement March-based solutions, there is no established methodology to tackle the cache controller logic, mainly due to its limited accessibility. One possible approach is Software-Based Self Testing (SBST): however, devising test programs able to thoroughly excite the replacement logic and made the results observable is not trivial. A test program generation approach, based on a Finite State Machine (FSM) model of the replacement mechanism, is proposed in this paper. The effectiveness of the method is assessed on a case study considering a data cache implementing the pLRU replacement policy.

1. Introduction

Modern processing systems depend on high performance processor cores, able to process high amounts of data and execute a very large number of instructions per second. To achieve the performance requested by the specifications, it is crucial to support the system with a cheap and fast memory hierarchy organization [1].

Among the different memory levels existing in a processing system, one of the most important is the cache one. In the memory hierarchy the cache is placed near the processor core, and it is dedicated to reducing the gap between high-performing processors and other memory devices, usually much slower.

In recent years, the costs of test and validation processes have been constantly growing with respect to the total integrated circuit (IC) manufacturing cost: in particular, there is not yet a mature and comprehensive methodology able to cope with all testing issues regarding the first level of memory hierarchy systems, i.e., caches.

However, research activities on cache testing are thriving, and different solutions have been proposed in this area. In [2], for example, a structural modification of the cache architecture to enhance the IDDQ testing sensitivity is proposed, while in [3] the authors present a Memory Built-In Self Test (MBIST) device, able to apply a modified March algorithm to both L1 and L2 caches. All these hardware-based approaches, however, necessitate of considerable modifications of the initial design.

Another interesting solution is to translate March-like tests into sequences of processor instructions [4], [5], [14], [22]. This methodology does not require altering the original design of the device, but it still needs special system features to ease main memory writing and reading operations while the cache memory is disabled; such mechanisms may not be present in the normal operation mode of microprocessor-based systems.

March-like approaches have proven efficient to test memory elements in caches, but they do not consider thoroughly the control part of the cache memory, which is crucial to obtain the correct behavior and the expected performance from the processor.

Software-Based Self-Test (SBST) [6] is a methodology that exploits the Instruction Set Architecture (ISA) of the targeted processor to run suitable test programs while the processor itself operates in normal mode: data generated by the programs are then checked for conformity with the expected results to detect possible misbehaviors. SBST does not require additional hardware and it can be applied both to stand-alone devices and to Systems-on-Chip (SoCs), where the processor and other cores included into the device are deeply embedded and their accessibility is limited. SBST can be seen as a extension of the work on functional tests for processors first presented in [7].

The success of a SBST-based strategy is strongly dependent on the availability of suitable test programs, able to address the different parts of the device. The limited accessibility of cache controllers' input and output signals, in terms of both controllability and observability, makes the creation of effective test programs especially critical.

In literature, it is possible to find a certain number of SBST approaches that address cache memory testing: in [18], for example, the conversion of a March algorithm to executable instruction sequences is proposed. The algorithm is improved to take into account the control logic of the cache, but the methodology is applicable to direct-mapped caches, only. A general SBST algorithm to generate test programs for data cache controllers is presented in [10], this time independently on the controller type. In [11], the authors report a hybrid

solution to test data and instruction cache controllers: the approach makes use of an external Infrastructure Intellectual Property (I-IP) core to increase the observability of the results.

This paper focuses on the replacement policy adopted in set-associative data cache memories and is based on the work presented in [7], that clearly shows how 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 is based on building a FSM model of the replacement circuit, which is then exploited by a traversing algorithm able to obtain a compact set of memory accesses suitable to test the targeted replacement logic. However, when that algorithm is applied to cache controllers implementing the pLRU mechanism, the length of the generated sequence tends to become excessively large. A major contribution of this paper is that we exploit some specific peculiarities of the pLRU replacement logic [8] to produce a more compact test sequence. The obtained results are suitable for post-production testing, incoming inspection and on-line tests of both stand-alone processors and processor cores embedded in SoCs: it can also be applied to different cache configurations, with regards to cache size, organization and writing strategies (i.e., write-back or write-through). Moreover, being based on a SBST approach, the method is suitable to detect delay faults, and does not require any change in the targeted hardware.

The rest of the paper is organized as follows. Section 2 introduces some vocabulary, sums up the basic concepts of FSMs and introduces the issue of FSM testing. Section 3 describes the proposed approach. Section 4 presents the case study and reports experimental results. Section 5 concludes the paper.

2. Background

2.1. Finite state machines testing

A finite state machine (FSM) M is defined in [20] as a quintuple M=(I,O,S,δ,λ) 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
- δ: S × I → S is the state transition function
- λ: S × I → O is the output function.

A 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 $\lambda(s, a)$.

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 [20]. A conformance test aims at generating a test sequence to determine whether the black box implementation of the modeled *FSM* behaves exactly as the specification machine, independently of its implementation and on the possible faults.

In a conformance testing:

- An initialization sequence is applied, bringing the implementation machine 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.

2.2. Cache Replacement Policies

A cache replacement policy is a major design parameter of set-associative cache memories. The efficiency of the replacement policy affects both the hit rate and the access latency of a cache system. In the worst case, a malfunctioning cache replacement mechanism can produce a bypass of the cache memory and severely degrade the system performance. 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. The most common replacement mechanisms are: Least Recently Used (LRU), Most Recently Used (MRU), pseudo-LRU (pLRU), Least Frequently Used (LFU), First In First Out (FIFO), FIFO-Clock, and many others.

2.3. Pseudo-LRU

The pseudo-LRU replacement policy is a hardware efficient approximation of a LRU algorithm where the *ages* of the ways in a cache set are not linearly ordered, but arranged in a binary tree (see Fig. 1). This tree has n leaves (one for each cache way in the set) where n is a power of two, and $n-1$ nodes. Each node is represented by a single bit, thus using $n-1$ bits, called *history bits*. Referring to Fig. 1, the least recently used way corresponds to the binary state of two history bits at different levels in the tree, in this case a_0b_0 or a_0b_1. In the example $a_0=0$ and $b_0=0$; therefore, the LRU line is that stored in the way w_0.

The main advantage of the pLRU policy is that its implementation requires less hardware than the LRU policy, but it inherently involves a loss of information about how historically have been accessed the ways in the set. For example, in a *4-way* LRU implementation we need a minimum of 8 bits (*2x4*) to store the "*access order*" of the ways in the set, while for the same case a pLRU implementation requires only 3 history bits. However, the main drawback of pLRU implementations is that when a miss arises it cannot guarantee that the cache always substitutes the least recently used line. This behavior is considered an anomalous behavior that does not represent in any case a functional bug, but an intrinsic weakness of the replacement policy.

The access order for every way (AWO_x) in the set depends on the values of the history bits a_0, b_0, and b_1, then, it is possible to formally defined it by:

$$AOW_0 = a_0 + b_0$$
$$AOW_1 = a_0 + \overline{b_0}$$

$$AOW_2 = \overline{a_0} + b_1$$
$$AOW_3 = \overline{a_0} + \overline{b_1}$$

Thus, when a memory access generates a miss, the current way whose access order (AOW_x) equals 0 is evicted from the line. In the figure, if $a_0b_0b_1 = 000$, the way that leaves the line is W_0 since $AOW_0 = 0$.

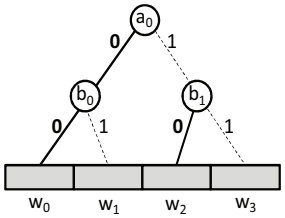

Fig 1. pLRU 4-way. The *"access order"* of the ways in the set is represented by the history bits a_0, b_0 and b_1. In this example, if $a_0b_0b_1$ = 000 the least recently used way is w_0 and therefore it will be evicted in the next miss access.

If we consider a hit/miss access to the cache memory, it is possible to state that the new value on the history bits is obtained by toggling from 0 to 1 the 0 values belonging to the accessed way represented by the AOW_x equations.

Assuming again that $a_0b_0b_1 = 000$, and supposing that a hit access is performed on W_0; then, the new value adopted by the history bits *is* $a_0b_0b_1 = 110$. Supposing that in this case, a new hit access is performed on W_1, the final history bits value is $a_0b_0b_1 = 100$, since $\overline{b_0}$ is toggled from 0 to 1.

Another important issue to consider when tackling the pLRU replacement policy is that it is susceptible to timing anomalies known as *domino effects*, as reported in [8]. That paper shows that there are cyclic sequences of accesses in pLRU caches that could cause that a data stored in a way never to be removed from the cache.

3. Proposed approach

The proposed approach described in this paper is based on the work previously introduced in [7]. In a few words, herein we improve the method presented in that paper by including an additional step able to handle cache anomalies present in the circuitry of the replacement mechanisms when implementing the pLRU algorithm. It is important to mention that we do not face the test of the memory elements present in the cache controller (i.e., the tag bits), but only the replacement logic.

The original algorithm in [7] is able to deterministically generate sequences of memory accesses for testing the LRU cache replacement algorithm available in cache controllers.

The cache controller functions, and specifically the replacement mechanisms, can be implemented in different ways, resorting to data structures such as a memory to store the history bits, priority queues, and others. However, we adopt a high-level strategy able to cope with different cache implementations, and thus independent on the actual implementation ("black box" approach).

Bearing in mind the complexity of the cache replacement mechanisms, we model their behavior using a FSM. Such a representation allows us to scale well considering different cache implementations, and avoiding an explosion on the considered cases without losing essential information regarding the implemented replacement algorithm.

In this paper, we propose an algorithm that initialize the FSM in a well known state by traversing some initial transitions or recurring to a flush instruction. Then, the algorithm traverses all the FSM transitions checking at every time if the reached transition corresponds to the expected one. in this way, the resulting transitions sequence is able to thoroughly excite the pLRU replacement mechanism, based only the knowledge of its behavior. The only cache information we assume observable is the cache hit/miss signal. The proposed algorithm includes supplementary memory accesses that increase observability without recurring to additional hardware. 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 great advantage of the presented methodology is that it faces all possible faults that could affect the cache replacement logic, while at the same time it is sufficiently general to apply to all possible implementations of the targeted replacement policy. The complete process is described in the following.

3.1. Additional testing considerations

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 [10], or resorting to a hybrid technique, as described in [11], or resorting to performance counters, as suggested in [12] and [22]. 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.

3.2. pLRU modeling

In the approach of [7], the formal representation of a FSM state depends on the internal order of the ways according to the normal evolution of the device. Therefore, considering a generic *n-way* cache and denoting each way as $w_0,w_1,..,w_{n-1}$, each possible state corresponds to a permutation of $w_0,w_1,..,w_{n-1}$, where the rightmost way is the first one that will be replaced, while the leftmost is the last one. Differently, in this paper we introduce a new FSM representation by stating a

different concept of state. In this case, we use the history bits available in the pLRU cache replacement logic in order to define our FSM states, avoiding the explosion of states and transitions in cache memories with a larger number of ways.

Considering the previous ideas, the FSM *M* 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 former contains the *n* addresses stored in the generic time in the *n-ways* of the cache set: each one of them, when accessed, produces a "hit" and activates a transition to another FSM state. The latter includes all the remaining addresses, 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 observed 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 the reached value of the history bits present in the pLRU replacement logic. Therefore, considering a generic *n-way* cache, being *n* a power of 2, there are *n-1* history bits, and consequently, 2^{n-1} states. In a *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.

Based on the previous considerations, it is possible to state that a *n-way* cache that implements the pLRU replacement mechanism, counts with 2^{n-1} states, and $(n+1)(2^{n-1})$ transitions.

Considering the characteristics of the resulting FSM, it is possible to see that the derived FSM 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 2 shows the FSM derived for the pLRU replacement logic of a 4-way cache: the FSM counts with 8 states and 40 transitions. It is possible to observe that the new FSM representation positively impacts the number of both states and transitions, since using the previous FSM representation, these values would amount to 24 and 125 respectively [7].

3.3. FSM traversing algorithm

The traversing algorithm firstly initializes the FSM in a well known state, then generates a sequence of memory accesses able to guarantee that all the transitions in the modeled FSM are traversed at least once, and at the end of every transition, the reached state is verified. However, considering the observability issues previously exposed, it is necessary to include a special sequence of memory accesses able to check whether the reached state is the expected one. This verification sequence is referred to as *status check*.

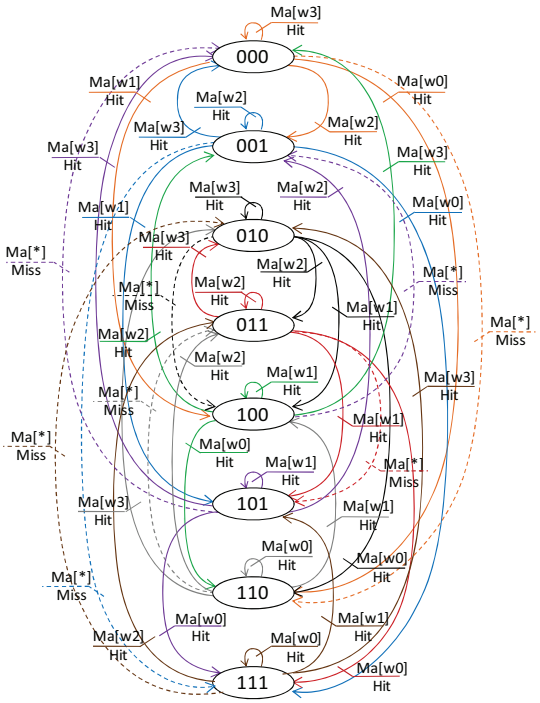

Fig 2. FSM model for the pLRU replacement logic of a 4-way cache

The initialization and generation of a sequence of transitions able to go through all the FSM transitions do not deserve special consideration since these tasks are relatively straightforward and easy to implement. In literature, the problem of generating a transition tour is known as the *Chinese Postman Problem* (CPP) [16]; and is possible to solve it by applying the Floyd–Warshall algorithm described in [17]. Since the size of the considered FSM is relatively small, finding an exact solution for this problem is rather easy.

In order to address the generation of the status check we must consider the pLRU functional behavior and identify the required sequence of memory accesses. For this purpose it is possible to exploit the concepts described in [7], but pLRU peculiarities introduce the necessity to calculate at every transition the real access order of the data present in the cache.

The access order for every way in a pLRU cache is determined by the history bits that change depending on the accessed way. In Table 1, ways ordering created by a pLRU algorithm and a LRU algorithm in a 4-way cache are shown.

Table 1. pLRU and LRU comparison example for a 4-way cache

LRU	pLRU	Memory access	$a_0b_0b_1$
$w_0w_1w_2w_3$	$w_0w_1w_2w_3$	Hit(w_0)	000
$w_0w_1w_2w_3$	$w_0w_1w_2w_3$	Hit(w_2)	001
$w_2w_0w_1w_3$	$w_3w_0w_1w_2$	Hit(w_1)	101
$w_1w_2w_0w_3$	$w_1w_2w_3w_0$	Miss	000
W3	**W0**	*Evicted way*	

LRU and *pLRU* columns in Table 1 show the order established by each of two replacement logics after a defined memory access. The left most way (w_x) is the newest, while the rightmost one is the oldest. The *Memory access* column shows the access performed at each step, while the last column shows history bits values for pLRU implementation.

Let us suppose that both cache controllers start from the same initial conditions. Then, after 3 different hit accesses, it is possible to see that a miss may remove from memories a different data depending on the cache implementation: *W3* in the LRU case, *W0* in the pLRU implementation.

In order to correctly apply the status check sequence, we use an algorithm that models the pLRU policy and is able to generate the next FSM state using as input the current state, the hit/miss expected value, and the accessed way. Assuming that $w_0,w_1,..,w_{n-1}$ represent the ways of a generic *n-way* set associative cache, and $A_0,A_1,....A_{n-1}$ the addresses of the blocks stored in each way; additionally, let us identify the oldest way in the memory with A_{old}, and w_{old}, that could not correspond to A_{n-1} w_{n-1}, due to the pLRU replacement logic. The proposed status check sequence works as follows:

- Firstly, we perform an access to an address (A_n) not present in the cache; the purpose of this operation is to remove the oldest block in memory A_{old}.
- Secondly, we access to the address A_{old} 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 oldest value, then, we access again to the just removed block, 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 *n-1* 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 some access to the addresses still in cache, provoking the same number of hit results, in order to lead back the FSM to the original state. This number of accesses ranges from *1* to *n-1*, and is calculated using the same function devised for modeling the pLRU policy.

It is important to note that in order to actually remove a block from the cache memory, and then check its absence from the cache, we must clearly know at every iteration which one is the oldest value present in the cache.

4. Case study

The effectiveness of the proposed methodology has been evaluated on the cache controller of the data cache available in the Leon3 processor core [15]. Leon3 is a 32-bit RISC processor compliant with the Sparc V8 architecture. It implements a 7-stage pipeline with separate instruction and data caches. The cache system supports associative caches counting up to 4 ways per cache line, different cache size possibilities, and the replacement policy is selected considering LRU, LRR or random strategies.

The basic Leon3 processor core is made up of pipeline, cache controllers and AMBA AHB interface (see Fig 3). The full vhdl source code is available under the GNU GPL license and distributed as part of the GRLIB IP Library.

In order to assess the proposed methodology, we implemented a pLRU replacement policy using the same structures and design parameters present in the Leon3. For this purpose, we only modified the vhdl files related to the data cache controller. Analyzing the GRLIB IP library, the vhdl file directly involved in the data cache controller description is *dcache.vhd*, which uses the *libcache.vhd* and *libiu.vhd* libraries, while the top level file is *cache.vhd*. These files contain 1,138, 667, 245, and 134 code lines respectively. On the other hand, the data cache memory is described in the *cachemem.vhd* file and counts 453 lines of code.

The main modifications were conducted in the *dcache.vhd* and *libcache.vhd* files in order to add the pLRU replacement policy. 98 new code lines are added, and 5 existing code lines are modified in the *dcache.vhd* file; in addition, 43 new code lines were included to the *libcache.vhd* file.

The entire functional behavior of the processor core was simulated at RT level using *ModelSim SE 6.4b* in order to verify and validate the correct implementation of the pLRU replacement mechanism for 2 and 4-ways. Some test sequences were manually devised, in order to check the correctness of the implementation. Interestingly, we also developed the test sequences (read/write operations) described by [8] that actually highlight the pLRU domino effects reported in our implementation.

In any case, it is important to note that the 2-way pLRU case is too simple and becomes the same that 2-way LRU case. Therefore, for the purposes of this work we are interested only in the 4-way pLRU case.

The data cache controller implementing a 4-way pLRU replacement mechanism was synthesized using *Synopsys Design Vision* targeting a homemade technology library. The synthesized version of the whole cache controller counts 8,624 equivalent gates, while the circuitry implementing our pLRU implementation requires 4,045 out of the 8,624 gates. The fault simulation campaign was performed on *Tmax* v. B-2008.09-SP3 by Synopsys. The number of stuck-at faults for the entire cache controller is 22,901; 10,223 of them correspond to the pLRU replacement circuitry.

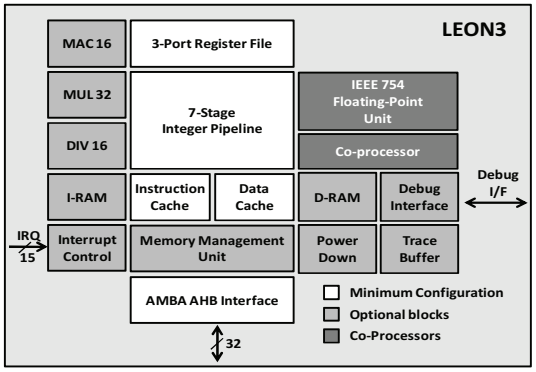

Fig 3. Leon3 processor core architecture

5. Experimental results

We developed a new tool in Java counting about 1,000 lines that is able to generate the traversing algorithm described before. The sequence of memory accesses is then translated to C language exploiting a Perl script that contains about 500 lines. The C program is then compiled using MKPROM for BCC v1.0.31b to be run in the Leon3 microprocessor core. The devised sequence counts 660 memory accesses that includes all the 40 FSM transitions present in the first line history bits of the cache memory, an average of 6 more access to perform the status check after every transition, and finally 380 additional accesses to toggle every history bit of the rest of the cache lines.

The complete program sizes 15,6 Kbytes, and takes about 293 K clock cycles to be executed. By applying the obtained algorithm, we verified that the sequence achieves 100% fault coverage on the considered list of stuck-at faults on the pLRU replacement logic available in the targeted cache controller.

6. Conclusions

In this paper we extend our previous work that tackled test program generation for the replacement policy available on cache controllers. The methodology is based on modeling the circuitry implementing the replacement policy as a FSM, which is then exploited to automatically generate a sequence of memory accesses that thoroughly excite the cache controller replacement policy. The proposed approach is able to make the behavior of the replacement policy observable from the outside, by just checking whether the circuitry generates the correct sequence of hit/miss operations.

The proposed approach is evaluated on a 4-way set associative cache controller that implements the pLRU replacement policy, on the Leon3 processor core. Fault simulation results show that the proposed methodology is able to fully test the circuitry.

When compared with [7], the methodology proposed here not only reduces the number of accesses to memory required by the original approach, but also avoids the explosion of states and transitions on larger cache implementations, thus significantly reducing the length of the final sequence.

7. References

[1] John L, Henessey & David A, Pattterson, "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] Perez H, W.J.; Ravotto, D.; Sanchez, E.; Reorda, M.S.; Tonda, A.; , "On the Generation of Functional Test Programs for the Cache Replacement Logic", Asian Test Symposium, 2009. ATS '09. , vol., no., pp.418-423, 23-26 Nov. 2009

[8] Berg, Christhop. PLRU Cache domino effects. ECRTS 2006. 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. http://drops.dagstuhl.de/opus/volltexte/2006/672.

[9] S.Thatte and J.Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429–441, June 1980.

[10] 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.

[11] 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.

[12] M. Hatzimihail, M. Psarakis, D. Gizopoulos, A. Paschalis, “A methodology for detecting performance faults in microprocessors via performance monitoring hardware”, Proc. IEEE International Test Conference, 2007, pp. 1-10

[13] A. J. Van de Goor, “Testing Semiconductor Memories, Theory and Practice”, John Wiley & Sons, 1991.

[14] A. J. Van De Goor, “Using March Tests to Test SRAMs", IEEE Design & Test, Vol. 10, issue 1, 1993, pp. 8 – 14.

[15] Aeroflex Gaisler, http:// http://www.gaisler.com/cms/

[16] J. Edmonds, E.L. Johnson, “Matching, Euler tours and the Chinese postman”, Mathematical Programming, vol. 5, pp.88-124, 1973.

[17] R. W. Floyd, “Algorithm 97: Shortest path”, Communications of the ACM, v.5 n.6, p.345, June 1962.

[18] 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

[19] 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," *17th IEEE Asian Test Symposium, 2008.*, pp.363-368

[20] 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

[21] Jim Handy, “The cache memory book”, 2th edition, Morgan Kaufmann publishers, 1998.

[22] J. Sosnowski, Improving software based self testing for cache memories, Proc. of IEEE 2nd International Design and Test Workshop, pp.49-54, 2007, IEEE Comp. Soc.