On the Functional Test of the Cache Coherency Logic in Multi-core Systems

J. Perez Acle
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

R. Cantoro, E. Sanchez, M. Sonza Reorda
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Abstract

Multi-core systems are becoming particularly common, due to the high performance they can deliver. Their performance strongly depends on the availability of effective cache controllers, able to guarantee (among others) the coherence of the caches of the different cores.

This paper proposes a method for the test of the cache coherence logic existing within each core in a multi-core system, resorting to a functional approach; this means that the method is based on the generation of a suitable test program, to be run in a coordinated manner on the cores composing the system. The method is able to detect hardware defects affecting this logic. The method was validated on a LEON3 multicore system.

1.Introduction

Multi-core systems are increasingly popular in the applications where high performance is required, due to the interesting mix of performance, flexibility and power they offer. However, the complexity of the devices implementing such multi-core systems, combined with the increased sensitivity to faults of new technologies, asks for new techniques able to effectively detect possible faults affecting their hardware structure, both at the end of the manufacturing process, and during the operational life (*in-field test*).

A common solution lies on resorting to Design for Testability (DfT) techniques, such as scan test or Logic BIST. However, these techniques may sometimes be inadequate: firstly, these solutions may often not be exploited during the operational life, for example because they require an external tester (not available for *in-field* testing); secondly, because IP producers tend not to disclose details about the DfT architectures, avoiding to impair IP protection; thirdly, because sometimes DfT is inadequate to achieve sufficient defect coverage, that can only be obtained by running the test in the same operating conditions (e.g., in terms of speed) and configuration of the application. For all these reasons, a functional test approach based on developing suitable test programs to be executed by each core and on observing the results produced is a suitable solution. This approach is also known as Software-Based Self-Test (SBST) [1].

Caches are one of the most critical components within multi-core systems, since their behavior can seriously affect the performance of the whole system. Previous papers [4][5][6] already described how their data and tag parts can be effectively tested resorting to a SBST approach. In some cases, their test can be made easier by exploiting the special instructions provided by some Instruction Set Architectures to directly access their content [15].

Additionally, multi-core systems require proper coherence protocols, able to guarantee that the content of the caches of the different cores is always up-to-date, so that each time a processor accesses a piece of data, it always accesses a correct and coherent value. Validation of cache-coherent multiprocessors is a challenging task, often performed through extensive simulation of randomly generated sequences of operations [2][7]. On the other side, it is also crucial to check whether any hardware defect affects the cache coherence logic. In [7] we focused on the test of the coherence logic of a cache controller implementing the MESI protocol. In that paper, we only considered the logic corresponding to the Finite State Machine implementing the protocol, neglecting the rest of the involved control circuitry.

The purpose of this paper is to propose a method to generate a proper test program to be run on a multi-core system in order to check whether the circuitry implementing the cache coherence protocol is affected by any hardware fault. The test program is derived from the functional specifications of the circuitry under evaluation only, and can therefore be reused on any circuit implementing the same coherence protocol. Interestingly, since the proposed technique is based on a test program, it is well suited to be adopted by system companies for both Incoming Inspection [3], and in-field test (since it is possible to activate the test at any time during the operational phase).

In order to practically validate our approach and to better quantify its cost in terms of memory occupation and execution time, some experimental results were gathered using a multi-core system based on the LEON3 processor [9].

2.Background

Nowadays, multi-core systems usually include multi-level caches. Each cache has a corresponding cache controller, implementing not only the functions required

978-1-4799-8332-2/15/$31.00 ©2015 IEEE

to properly operate the cache by itself, but also to guarantee the coherence of the shared data allocated at a given time on the different processors' caches.

In particular, the Cache Coherence Logic (CCL) mainly aims at avoiding the case in which two copies of the same memory block allocated in two different caches do not contain the same values. To avoid this problem, several mechanisms exist. One of the most popular, which is considered in this paper, is based on spying the addresses flowing on the bus (*snooping*), so that a block in a cache is invalidated if the value of the same block has been changed in another processor cache. Hence, a key role in the cache coherence logic is played by the Validity Bit (VB) associated to each cache line. The VB is substituted by several bits when the adopted cache coherence protocol is more complex, like in the case of the MESI protocol [10].

In this paper, we consider a Cache Coherence Logic implementing a simpler protocol, such as the one adopted for the data cache of the LEON3 processor core [9]. In such a case a Validity Bit is associated to every cache line; in addition, the cache implements the write-through, no allocate mechanism. If the processor is used in a multi-core configuration, the cache coherence logic continuously snoops the bus transfers: if another processor executes a write operation on a block which is also stored in the local memory, the block is invalidated (i.e., VB is forced to 0) thus forcing every further access to the block to access the memory. VB is forced to 1 each time a new block is uploaded into the cache memory.

Based on the above discussion, the CCL is mainly composed of the following elements:

- the VBs (one for each cache line)
- a set of comparators, whose inputs are the external address bus and the tag fields associated to the cache set corresponding to the address currently on the bus
- some control logic, able for example to interact with the bus and understand when to sample the address value during a memory write operation.

In the next section we will propose an algorithm able to test these three pieces of circuitry by executing a proper test program and checking the system behavior.

3.Proposed approach

It is described here the algorithm proposed to test the cache coherence logic in a multi-core system; for sake of simplicity the usage of the algorithm in a dual-core system where each core includes a direct mapped cache is initially discussed: These assumptions will be removed in the second part of this section. It is also assumed that a previous test has been run, able to test the cache itself including the cache controller circuitry not corresponding to the coherence logic.

The algorithm targets stuck-at faults. The test of the targeted logic requires first exciting each fault, and then observing the fault effects. We will deal with the two issues separately. It is important to note that the basic function of the CCL is to invalidate a given cache line when another processor executes a write operation on the memory block it stores, i.e., to properly modify the value of the corresponding VB. Hence, observing the effect of any fault in the CCL once it has been excited means observing the value of the corresponding VB.

In order to perform this set of operations, in the following the different memory operations involved in the CCL testing are described. They require the use of two processor cores: P0 and P1. P0 is the target processor core, whose CCL we want to test, whereas P1 is a support processor intended to execute operations that invalidate the data in the P0 cache module.

3.1 Excitation phase

We first need to check whether any stuck-at fault exists, affecting the VBs.

In the following, the required operations developed for this purpose are detailed. Every step details the processor required to execute the listed operations and the expected behavior in the targeted cache:

0. P0 - cache flush; all validity bits are initialized to 0;
1. P0 - upload each cache line with a known block (thus turning all VBs to 1); for every line a read operation is performed and a cache miss is expected;
2. P0 - access the block which was uploaded in each line in the previous step, checking that a hit is triggered; if not, the corresponding VB is affected by a stuck-at-0;
3. P1 - invalidate the P0 cache (thus turning all VBs in P0 to 0);
4. P0 - access the block which was uploaded in each line, checking that a miss is triggered; if not, the corresponding VB is affected by a stuck-at-1. In the absence of faults, all VBs turn back to 1, and a cache miss is expected.

Each of the above steps (apart from step 0) consist of n read or write operations, being n the number of cache lines, each accessing to a memory location which is supposed to be stored in a different cache line. Details about the rules to compute these addresses can be found in [4]. Hence, the above steps require $4n$ instructions, plus the cache invalidation instruction (*flush* in the case of the LEON3 assembly code).

Secondly, we need to check whether the CCL is affected by any fault. In a direct mapped cache, the CCL is basically composed of a comparator; each time a processor core accesses the memory, this comparator compares the address on the bus with the content of the

tag field of the corresponding cache line. For testing the comparator we can exploit the algorithm proposed in [9]. Such an algorithm specifies a set of *2m+2* input vectors that should be applied to the *2m* comparator inputs (as shown in Fig. 1), guaranteeing that they allow achieving full stuck-at fault coverage, independently of the specific comparator implementation.

Applying each of these test patterns to a comparator in the CCL requires the following steps:

1. P0 - upload in a suitable cache line a memory block, so that the value of the tag field matches the required value (input B); this can be achieved by executing a read access to a suitable location in memory;
2. P1 - execute a write operation on the block uploaded at the previous step; this implies that the required value is written to the bus, and thus applied to the A input of the comparator.

Depending on the test vector, the comparator is expected to produce a match or mismatch; correspondingly, the related VB in P0's cache is forced to 0 or left at 1.

The algorithm can be easily extended to a core including a *k* ways set associative cache. In this case the CCL includes *k* comparators, and the algorithm should be repeated to test each of them.

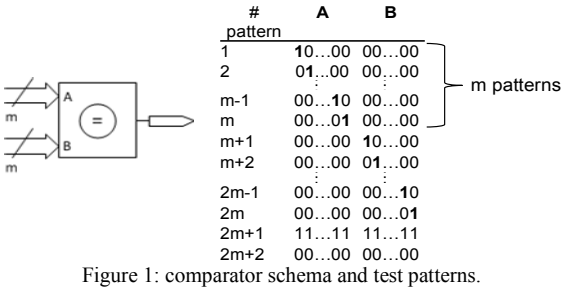

Figure 1: comparator schema and test patterns.

3.2 Observation phase

The above algorithms allow forcing a known VB to 0 or 1. In order to observe whether the target VB holds the expected value or not, the test program should execute an access to the block stored in the corresponding cache line. If this triggers a hit, it means that the VB holds the value 1, otherwise (miss) the VB holds the value 0. Most of the faults affecting the CCL can be labeled as *performance faults*, i.e., they do not affect the correctness of the result produced by the test program, but rather its performance [14].

In order to observe whether a given memory access triggers a hit or miss we can adopt different techniques, based on the hardware mechanisms available:

- *performance counters* existing in the processor, devoted to count the number of hit and miss situations triggered by a given program [11];
- an *internal timer*, devoted to measure the test program execution time
- the *debug infrastructure* usually provided by the processor, able to trace and communicate to the outside the bus activity for a given period [12]
- an *ad hoc module* added to the system and in charge of monitoring the bus activity [13].

3.3 Analytical performance analysis

The main component of the test time required by the algorithm is related to memory accesses. Their number can be estimated as follows.

Some of the transfers needed for the execution of the test are cache hits and consequently internal to a core, while others are cache misses and will compete with the other processors to access the bus. In both cases the amount of necessary data transfers depends on the number of cores *n*, the number of tag bits *ntb* (which affects the size of the required comparator), and the number of lines *nl* in the cache, assuming it is direct mapped. The number of missed read and write transfers (i.e., *misses*) required by the complete test are:

$$N_{miss} = n * (nl * (2\ rd + 1\ wr) + ntb * 2\ wr + 4\ rd + 2\ wr)$$

The internal transfers (*hits*) on each core are:

$$N_{hit} = (nl + 2) * rd$$

3.4 Optimizing the test in multiple-core systems

In the previous sub-sections we described how to test the possible stuck-at faults affecting the CCL of a target processor core, using a second core as a support.

In the case of an *n*-core system some parallelism can be exploited using each processor to support the test of the following one. Processor P0 plays the role of support processor for P1, P1 for P2, and so on.

The total duration of the algorithm will depend on the performance of the bus. In an ideal scenario where there is no bus contention, the test duration will remain constant, equal to *T* the duration of the algorithm for testing a single core. On the other side, if the memory bus access is saturated the execution time will be dominated by the memory accesses and will increase linearly with the number of processors *nT*.

4.Experimental results

The approach effectiveness has been experimentally evaluated in a multi-core system based on the freely available LEON3 processor by Aeroflex Gaisler [10].

A multi-core system was implemented including a minimum set of memory cores, plus a configurable number of LEON3 processors. Every processor core is

instantiated with separate data and instruction caches. The configuration for the data caches used in our test was 1 way (direct mapped), 1Kbyte/way, 16 bytes/line. As mentioned, the data cache implements the write-through policy, with write no-allocate on a write miss.

The test program on each processor requires the execution of 317 assembly instructions for the VB test and 412 instructions for the Comparator test part, plus some loop instructions for synchronization. For every processor core, the compiled test program requires about 1KB of code memory.

The execution times for both parts of the test (VB and Comparator) are reported in Fig. 2 for systems with 2 to 8 processors. VB test corresponds to the solid line, and Comparator test to the dashed one. All values are expressed in number of clock cycles. These values show that the effects of bus and memory contention do impact more significantly on the VB test, where the execution time grows more quickly with the number of cores. In the Comparator test the execution time grows more slowly, due to the higher amount of parallelization that our algorithm achieves.

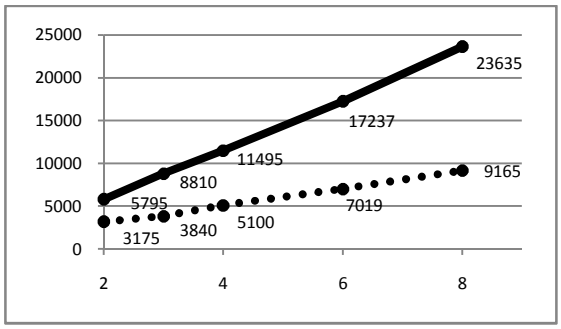

Figure 2: Test programs execution time (clock cycles) vs. number of cores.

Finally, in order to validate the correctness of the proposed algorithm we analyzed the LEON3 data cache controller RTL source code to identify the parts of it which implement the snoop mechanism. The whole system was then simulated using the Mentor Graphics ModelSim tool to check that the algorithm behavior is the expected one.

5.Conclusions

This paper proposes a method to detect possible faults affecting the hardware implementing the cache coherence logic integrated in each cache controller of a multi-core system.

The proposed approach is based on a functional approach, i.e., on the execution of a carefully written test program executed by different cores in a coordinated manner. The method achieves by construction a full fault coverage of the static faults in the addressed logic, and is suitable to be used both during end-of-manufacturing test and for in-field test (e.g., when safety-critical systems are considered).

We experimentally evaluated the proposed approach on a system integrating a variable number of LEON3 cores, showing its cost in terms of execution time, which grows linearly (and slowly) with the number of cores.

References

[1] M. Psarakis, et al., "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3, pp. 4-19, May-June 2010

[2] B. O'Krafka, et al., "MPTG: a portable test generator for cache-coherent multiprocessors", 14th IEEE Annual International Phoenix Conference on Computers and Communications, 1995, pp. 38-44

[3] M.L. Bushnell, V.D. Agrawal, "Essential of Electronic Testing", Kluwer Academic Publishers, 2000

[4] S. Di Carlo, et al, "Software-Based Self-Test of Set-Associative Cache Memories", IEEE Transactions on Computers, vol. 60 n. 7, pp. 1030-1044

[5] M. Riga, et al., "On the functional test of L2 caches", 18th IEEE International On-line Testing Symposium (IOLTS), 2012, pp. 84-90

[6] W. J. Perez, et al., "Software-Based Self-Test Strategy for Data Cache Memories Embedded in SoCs", 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008., pp.1-6

[7] E. Sanchez, M. Sonza Reorda, "On the functional test of MESI controllers", 12th IEEE Latin American Test Workshop (LATW), 2011

[8] X. Qin, P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012

[9] H. Grigoryan, et al., "Generic BIST architecture for testing of content addressable memories", 17th IEEE International On-Line Testing Symposium (IOLTS), 2011, pp. 86-91

[10] http://www.gaisler.com/index.php/products/processors/leon3

[11] IA-32 Intel Architecture Software Developers Manual

[12] M. Hatzimihail, et al., "A Methodology for Detecting Performance Faults in Microprocessors via Performance Monitoring Hardware", IEEE International Test Conference, 2007, paper 29.3

[13] M. Grosso, et al., "An on-line fault detection technique based on embedded debug features", Proc. IEEE International On-Line Testing Symposium, 2010, pp. 167–172

[14] W. J. Perez, et al., "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs", IEEE International On-Line Testing Symposium, 2008, pp. 143-148

[15] T.-Y. Hsieh, et al., "Tolerance of Performance Degrading Faults for Effective Yield Improvement", IEEE International Test Conference, 2009, Lecture 3.1

[16] G. Theodorou, et al., "Software-Based Self-Test for Small Caches in Microprocessors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2014