

# Functional Self-Testing for Bus-Based Symmetric Multiprocessors

A. Apostolakis<sup>1</sup>    D. Gizopoulos<sup>1</sup>  
Department of Informatics  
University of Piraeus, Greece  
{andapo | dgizop | mpsarak}@unipi.gr

M. Psarakis<sup>1</sup>    A. Paschalis<sup>2</sup>  
Department of Informatics & Telecom.  
University of Athens, Greece  
paschali@di.uoa.gr

## Abstract

*Functional, instruction-based self-testing of microprocessors has recently emerged as an effective alternative or supplement to other testing approaches, and is progressively adopted by major microprocessor manufacturers. In this paper, we study, for first time, the applicability of functional self-testing on bus-based symmetric multiprocessors (SMP) and the exploitation of SMPs parallelism during testing. We focus on the impact of the memory system architecture and the cache coherency mechanisms on the execution of self-test programs on the processor cores. We propose a generic self-test routines scheduling algorithm aiming at the reduction of the total test application time for the SMP by reducing both bus contention and data cache coherency invalidation. We demonstrate the proposed solutions with detailed experiments in two-core and four-core SMP benchmarks based on a RISC processor core.*

## 1. Introduction

The industry trends for improving microprocessors performance during the last decades mainly aimed at clock frequency increase and transistor size shrinkage. Since transistor scaling causes an exponential increase in leakage power this direction seems to have a dead end in MOS transistor technology. Major microprocessor vendors recently introduced new generations that integrate multiple processor cores on a single die, which although operate at lower frequencies are able to deliver higher computing power exploiting thread-level parallelism.

Manufacturing testing faces new challenges when applied to multiprocessor chips. The extension of successful uniprocessor testing approaches to the multiprocessor level is not straightforward. Such an emerging testing approach that actually forms an integral part of the manufacturing test process for uniprocessors, is *Functional Instruction-based (or Software-based) Self-Testing (SBST)*. Functional self-testing or SBST approaches are

enjoying revitalization in the last years and have been studied by many research and development groups [1]-[7]. During functional self-testing, the processors execute self-test programs from the on-chip cache in the normal operating mode. Its use does not aim to replace the other structural testing approaches but to augment them to reach higher test quality. It minimizes the need for high-cost functional testers, enables at-speed testing and can be re-used for on-line testing.

Multiprocessor testing approaches have been recently described in [8]-[10]. Bayraktaroglu *et al.* [8] proposed a *cache resident* test methodology for testing cores of multiprocessor chips. Test programs, test data and test responses are stored into the L2 cache shared among the processor cores. The authors applied the proposed approach to an UltraSPARC T1 multiprocessor which integrates eight SPARC V9 cores. Riley *et al.* [9] demonstrated the testability features of the Cell processor that includes a 64-bit dual-thread processor of the PowerPC architecture and eight synergistic processor elements all connected via an element interconnect bus. Tripp *et al.* [10] introduced the Drive-Only Functional Testing (DOFT) approach to reduce microprocessor test costs. According to this approach, the ATE drives only the functional stimulus to the processor whereas the functional responses are either compared to expected responses or accumulated into a signature inside the processor. Using DOFT the cores of a multiprocessor can be tested in parallel reducing test application time.

One key issue in multiprocessors testing is the management of test application time [10]. Assuming that all processor cores have to execute the same test programs (known to be effective from the uniprocessor case), if the inherent parallelism of the multiprocessor is not sufficiently exploited and the processors execute the test programs sequentially, the total test application time will quickly scale with the number of processor cores. This can be avoided by the parallel execution of test programs. In this case, the memory system hierarchy may be the bottleneck to test application time since the concurrent access requests from several processors to shared memory positions (test instructions and data) will lead to excessive memory access delays and processor stalls.

---

Work supported by the European Union and the Greek General Secretariat for Research and Technology, PENED research grant 03ED229.

In this paper, we study the application of functional instruction-based (or software-based) self-testing on *bus-based symmetric multiprocessors* (SMPs) focusing on the reduction of test application time. A systematic experimental analysis of a multiprocessor architecture with private L1 caches and a shared L2 cache showed that as multiple cores share the same system bus and memory system, bus contention and cache coherency mechanism slow down the self-test process and limit parallelism during test application. To minimize CPUs idle intervals, the test routines must be scheduled so as to reduce the impact of shared resources use (bus and shared L2 cache). The contribution of this paper is twofold. First, it ports the functional instruction-based self-testing of uniprocessors to symmetric multiprocessors. Second, it develops an efficient test routines scheduling methodology for test application time reduction. The scheduling is based on execution time and memory references statistics of the self-test routines of the uniprocessor and thus it is effectively implemented without time-consuming simulations of the multiprocessor. We demonstrate the proposed methodology in two-core and four-core SMPs based on a popular RISC processor core, OpenRISC 1200 [11] extensively used in SBST research for uniprocessors. We present experimental results (fault coverage, test application time, test program size) for the entire multiprocessor logic (the processor cores, the cache controller and cache coherency unit logic) that show the effectiveness of the proposed methodology.

## 2. Multiprocessor Architecture

We focus on the most popular architecture of multiprocessor systems: symmetric multiprocessors (SMP) [12]. SMPs have a single, shared, main memory that has a symmetric relationship to all processors and a uniform access time from any processor; this architecture is often called uniform memory access (UMA) multiprocessor.

Figure 1 shows the block diagram of a typical SMP architecture. In this architecture each processor core has a private L1 cache while a single L2 cache is shared among all processor cores. The cache coherency unit of each processor maintains the consistency between the contents of the private L1 caches and the shared L2 cache. The cache coherency units implement the *snooping* protocol, the most popular cache coherency protocol. Snooping protocol monitors the system bus to examine the caches status, in particular to determine whether or not the local L1 caches have a copy of a block that is requested on the bus. When a write operation is performed in a location that a local L1 cache has a copy of, then the cache controller must invalidate the local data of that location. This is the well-known *write-invalidate* cache coherency protocol [12]. A bus arbiter allocates the bus to the processors (and thus the shared L2 cache) in a typical round robin fashion. Write buffers in each L1

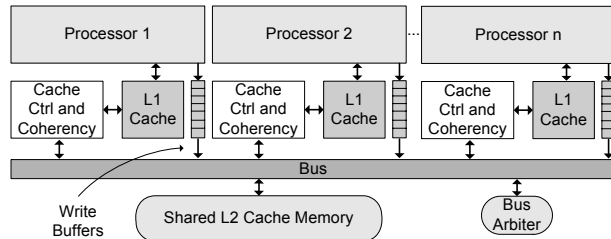


Fig. 1: Symmetric multiprocessor architecture

cache temporarily hold data waiting to be written to the shared L2 cache speeding up the memory subsystem.

## 3. Functional Self-Testing of SMPs

In the functional self-testing approaches for uniprocessors, test programs and data are downloaded into instruction and data caches, respectively, using low-speed, low-cost testers. Subsequently, the test programs are executed at the processor's actual speed and test responses are stored back in the cache; finally responses are uploaded into the tester memory. Since today's microprocessors have a fair amount of on-chip cache, the execution of self-test programs from the on-chip cache is considered as the best practice [7], [8] for test time reduction avoiding external (main) memory access cycles (necessary for manufacturing testing).

In this section, we systematically deal with the porting of the popular functional self-testing approach for uniprocessors into symmetric multiprocessors with shared resources. We first identify the testability challenges and problems of the straightforward application of functional self-test routines to SMPs, in terms of total test application time and total memory footprint. During our evaluations, we assume that:

- Before test application, the self-test routines are downloaded into the on-chip L2 cache via a cache access protocol. We adopt this solution instead of using the individual local L1 caches for test program and data storage because: (a) L1 caches are always much smaller than L2 caches and may not be sufficient for self-test program and data storage, (b) self-test program execution from L1 caches requires  $n$  separate downloads of the program (thus excessive download time) while execution from the shared L2 cache requires only one download, and (c) execution of self-test programs from the L2 cache allows testing of the bus arbiter logic. The same approach using self-test execution from shared L2 caches is also adopted in [8] for the functional testing of Sun's UltraSPARC T1.
- Self-test code is developed so that no external memory access cycles are initiated. As in [7], [8], we assume that in a manufacturing testing setup, the processors do not have access to external memory.

Uniprocessor or Multiprocessor and Self-test approach	Program download time	Program execution time	Responses upload time	Memory footprint	Faulty core identification capability
Uniprocessor	$D$	$E$	$U$	$S$	n.a.
Multiprocessor SC-SE	$D$	$n \times E$	$n \times U$	$S$	yes
Multiprocessor SC-PE	$D + D_{acc}$	$E + E_{acc} + E_{bus} + E_{inv}$	$U$	$S + S_{acc}$	no
Multiprocessor MC-PE	$n \times D$	$E + E_{bus}$	$n \times U$	$n \times S$	yes

**Tab. 1: Comparison of alternative functional self-test approaches.** ( $n$  = number of processors,  $E_{acc}$ =extra execution time due to responses accumulation,  $E_{bus}$ =extra execution time due to bus contention,  $E_{inv}$ =extra execution time due to cache invalidation,  $D_{acc}$ =accumulation code download time,  $S_{acc}$ =accumulation code size).

### 3.1 SMP Functional Self-Test Alternatives

Functional self-testing in an SMP architecture as in Figure 1 can be performed in three different ways:

#### *Single Copy–Sequential Execution (SC-SE)*

A single copy of the self-test program is stored in L2 cache and one of the processor cores executes it at a time. The download time for the test program is equal to the uniprocessor case (single download). During self-test only one processor possesses the bus at a time. Test responses of each core are stored in the L2 cache and uploaded to the tester before the next processor executes the self-test routines. This scheme eliminates bus contention and cache coherency invalidation simply because there is no parallel execution at all.

#### *Single Copy–Parallel Execution (SC-PE)*

As in the previous case, a single copy of the self-test program is stored in the shared L2 cache but the processor cores execute it in parallel. To enable the parallel execution of the same code by multiple processors, the test routines must be modified to allow the storage of test responses from different cores to the same data variables (because all processors execute the *same copy* of the program). This can be achieved by *accumulating* the test responses of the different processors, i.e. each processor does not write its response to the response variables, but rather it accumulates its response with the previous responses by other processors.

The download time is equal to the uniprocessor case plus a small extra time to download the code modifications due to responses accumulation. In this approach, the processors compete for the system bus to use the L2 cache and bus contention severely slows down the test execution. Moreover, cache coherency invalidation adds time overhead due to the shared memory addresses that the processors access to accumulate the test responses. The upload time for the test signatures is equal to the uniprocessor case, but the uploaded responses are the accumulated test responses of all processors. Therefore, faulty processor identification is not possible.

#### *Multiple Copies–Parallel Execution (MC-PE)*

Unlike the two previous approaches, multiple copies of the self-test program are stored in L2 cache and each processor executes its own copy in parallel with the other

cores. The L2 cache is divided in  $n$  different regions and each region stores the self-test program and test responses for one processor. The download time of the test programs and the upload time for test responses are each  $n$  times the corresponding download/upload times of the uniprocessor. Self-test application can be seriously slowed down due to bus contention because all processors use the system bus to access the shared L2 cache. Faulty processor identification is possible since each processor stores its test responses to different L2 cache regions. Table 1 summarizes the characteristics of the three SMP functional self-test alternatives.

## 4. Proposed Functional Self-Test Approach

It is clear to this point that the two major issues for the effective application of a functional self-test approach to an SMP architecture are the time overheads due to data cache invalidation and bus contention. We propose two generic solutions to reduce them.

### 4.1 Data Cache Invalidation Reduction

A generic solution for the reduction of data cache invalidations requires the test responses of the processors to be stored in *different* regions of the shared L2 cache. In this case, none processor’s cache controller will invalidate any word of its local L1 cache during testing. This is the case in the MC-PE approach where multiple program copies are executed in parallel from different L2 cache regions. The main drawback of this approach is the multiple time-consuming downloads of the self-test program. What we propose to alleviate this problem is that the self-test program is downloaded *only once* and a *test scheduler* controls the collection of test responses in different regions of L2 cache (an improvement over the SC-PE scheme of the previous subsection).

The *test scheduler* is a very small code portion that is added to the self-test program. It is assigned the task of scheduling the self-test routines among the processor cores by appropriately setting the *Base Address Pointer (BAP)* for each of the test routines based on the *processor’s unique core id* in the SMP. The BAP is used by the test routines to index the location of the data cache where the test responses are stored. The proposed test scheduler uses the id of the processor that executes the test program and assigns a different value to each BAP,

so that each processor stores its responses in a different region. The test scheduler algorithm is the following:

```

# Processor Identification
Read core_id
# Change the Base Address Pointer (BAP)
and Schedule the Self Test Routine
#
For i=1 to Number of Self Test Routines
Begin
  BAP ← L2_addr[core_id]
  Call Self_Test_Routine(i, BAP)
End

```

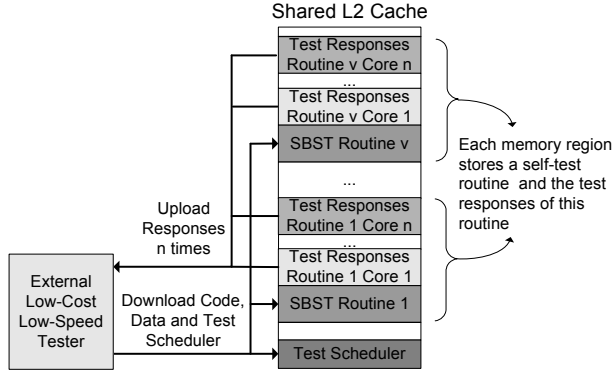


Fig. 2: Shared L2 cache and its allocation

The shared L2 cache is divided into  $v+1$  regions (where  $v$  is the number of routines in the self-test program) as shown in Figure 2. The first small region is used for the storage of the test scheduler code. The next  $v$  regions are further divided into  $n+1$  subregions: the first subregion stores the  $v$ -th self-test routine and the next  $n$  subregions store the corresponding test responses of the  $n$  processors.

The download time is equal to the download time of the uniprocessor plus the small extra time for downloading the test scheduler. The test application time is equal to the test application time of the uniprocessor plus the overhead from bus contention. The cache invalidation time overhead ( $E_{inv}$  in Table 1) is completely eliminated since the processors write on separate data cache regions. Finally, the upload time is  $n$  times the upload time of the uniprocessor.

## 4.2 Bus Contention Reduction

Our second aim is to reduce bus contention among the processors and the key idea is the processors to execute the self-test routines in a *different order* so that they access the shared bus in different times reducing bus conflicts. We propose a methodology that schedules the self-test routines among the processors so that the *mean number* of processors requesting to access the shared L2 cache through the common bus is *minimized*. Our methodology is based on high-level self-test programs statistics easily available from the uniprocessor case: the *total number* of L2 cache accesses that each self-test routine requires and its *test execution time* in the uniprocessor.

The processors' requests to access the shared L2 cache can be modelled as a *queuing system* [13]. We assume a finite number of processors and that every processor enters the queue when it submits a bus request. The basic terminology to study the queuing system is:

$n$ : number of processors in the SMP

$v$ : number of routines in the self-test program

$P_k$ : probability that  $k$  processors ( $0 \leq k \leq n$ ) are into the queue waiting to be served by the L2 cache (to access the bus)

$\lambda_k$ : mean requests arrival rate from the processors when  $k$  processors are already in the queue (state  $k$ )

$\mu_k$ : mean service rate of the shared L2 cache when  $k$  processors are in the queue (state  $k$ )

$\bar{\mu}$ : mean service time (mean access time of the shared L2 cache in clock cycles for a read or write access)

$\lambda$ : mean requests arrival rate from the processors

$\mu$ : mean service rate of the shared L2 cache

$L_q$ : mean number of processors in the queue

As mentioned the proposed solution schedules the test routines in the processors aiming to minimize the number of processors waiting to access the bus, i.e minimizing  $L_q$ . We approximate the variable requests arrival rates  $\lambda_k$  in the queuing system, assuming that the mean arrival rate  $\lambda$  follows a Poisson distribution. The mean service rate is a constant  $\mu$  and the mean service time of the shared L2 cache is:

$$\bar{\mu} = \frac{(R_r \cdot S_r) + (R_w \cdot S_w)}{R_r + R_w} \quad (1)$$

where  $R_r$  and  $R_w$  are the total number of read and write requests of the self-test routines respectively, when these are executed in the uniprocessor.  $S_r$  and  $S_w$  are the memory access time in clock cycles of the shared L2 cache for a read or a write request, respectively. Assuming that the write buffers are never full, then  $R_w = 0$ , and  $\bar{\mu} = S_r$ .

According to the above assumptions, the queuing system is an M/M/1/n [13]. "M/M" denotes that the arrivals and the departures follow a Poisson distribution, "1" denotes the single server of the queuing system (the shared L2 cache), and "n" denotes the number of the processors. A Poisson distribution satisfies our modelling requirement for random occurrence of the arrivals and departures [13]. When the system is in state 0, there are no access requests and thus no processor waits in the queue. In all other states 1, 2, ..., n the number of processors in the queue is 1, 2, ..., n, respectively. The variables  $\lambda_k$ ,  $\mu_k$ ,  $P_0$  and  $L_q$  are given by the following formulas:

$$\lambda_k = \lambda(n-k), \quad 0 \leq k \leq n \quad L_q = n - \frac{\lambda + \mu}{\lambda}(1 - P_0) \quad (2)$$

$$\mu_k = \mu, \quad 0 < k \leq n \quad P_0 = \frac{1}{\sum_{k=0}^n \frac{n!}{(n-k)!} \left(\frac{\lambda}{\mu}\right)^k} \quad (3)$$

Our methodology schedules the self-test routines in the processors in the combinations that give the smallest mean value of processors in the queue. Our calculations are based on the self-test routine with the smallest test application time (in clock cycles). We denote by  $c_i$  and  $r_i$ ,  $1 \leq i \leq v$  the total execution clock cycles (in the uniprocessor) and the total bus access requests of each routine  $i$ .

For each combination of the test routines, we order them so that  $c_1 < c_2 < \dots < c_n$ , and then calculate the total bus requests rate, the total services rate and corresponding waiting probabilities in an interval of  $c_1$  clock cycles (the execution time of the smallest routine).

$$\lambda = \frac{r_1 + \sum_{i=2}^n r_i \cdot c_1}{n} \quad (4) \quad \mu = \frac{c_1}{\mu} \quad (5)$$

The algorithm that schedules the  $v$  routines to the  $n$  processors is the following:

- Step 1: Compute the mean service time ( $\bar{\mu}$ ) according to equation (1) and the number of combinations.
- Step 2: For every routines combination, identify the test routine with the smallest execution time.
- Step 3: Compute  $\lambda$  and  $\mu$  for every routines combination according to equations (4) and (5).
- Step 4: Compute  $P_0$  and  $L_q$  for every routines combination according to equations (2) and (3).
- Step 5: Choose the routines combination with the smallest value of  $L_q$  and schedule these routines first.
- Step 6: Identify the routine with the smallest execution time in the combination. Replace this routine with another one which when combined with the remaining routines being executed, gives the minimum the value for  $L_q$ .
- Step 7: Go back to step 6 until all the processor cores execute all  $v$  self-test routines
- Step 8: If at the end of scheduling one or more routines are scheduled in parallel for  $n$  processors, abort the current scheduling and schedule these routines first with a combination that minimizes the value of  $L_q$ . Then, go back to step 6 to schedule the remaining routines.

The scheduling algorithm can be easily automated and quickly leads to an effective scheduling of self-test routines based on known values from simulation of the routines in the uniprocessor (bus requests and execution time). We illustrate the algorithm with an example.

*Example:* Consider an SMP with  $n=3$  processors and a self-test program which consists of  $v=4$  routines. The bus access (read and write) requests and the total clock cycles of the routines (in uniprocessor) are:

Test routines	Total clocks	Read requests	Write requests	Total requests
R <sub>1</sub>	40	4	-	4
R <sub>2</sub>	60	3	2	5
R <sub>3</sub>	80	4	-	4
R <sub>4</sub>	60	6	-	6
Total	240	R <sub>r</sub> = 17	R <sub>w</sub> = 2	19

Also, we assume that the L2 cache read service time ( $S_r$ ) is 4 clocks and its write service time ( $S_w$ ) is 1 clock.

Algorithm Steps 1 – 4: We compute the values  $\lambda$ ,  $\mu$ ,  $P_0$  and  $L_q$  for all four possible routines combinations:

Combinations	min clks	$\lambda$	$\mu$	$P_0$	$L_q$
[R <sub>1</sub> ,R <sub>2</sub> ,R <sub>3</sub> ]	40	3.11	10.85	0.40	0.31
[R <sub>1</sub> ,R <sub>2</sub> ,R <sub>4</sub> ]	40	3.78	10.85	0.33	0.40
[R <sub>1</sub> ,R <sub>3</sub> ,R <sub>4</sub> ]	40	3.33	10.85	0.38	0.34
[R <sub>2</sub> ,R <sub>3</sub> ,R <sub>4</sub> ]	60	4.67	16.28	0.40	0.31

Algorithm Step 5: The combinations with the smallest  $L_q$  are [R<sub>1</sub>,R<sub>2</sub>,R<sub>3</sub>] and [R<sub>2</sub>,R<sub>3</sub>,R<sub>4</sub>]. We start with [R<sub>1</sub>,R<sub>2</sub>,R<sub>3</sub>].

Algorithm Step 6: The smallest routine is R<sub>1</sub>. Processors 2 and 3 are still executing routines R<sub>2</sub> and R<sub>3</sub>, respectively. We look for a combination that includes routines R<sub>2</sub> and R<sub>3</sub>, replacing routine R<sub>1</sub> with another, and with the minimum value of  $L_q$ . This combination is [R<sub>2</sub>,R<sub>3</sub>,R<sub>4</sub>].

Algorithm Step 7: We repeat Step 6 until all processors execute all routines. Step 8 is not used in our example.

The proposed test routines scheduling according to the proposed algorithm is shown in Figure 3.

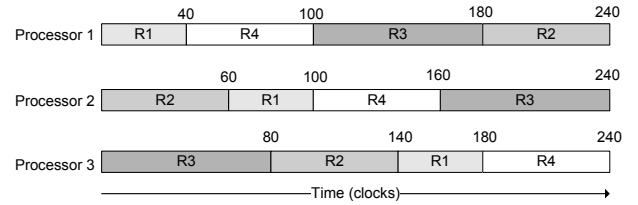


Fig. 3: Example - test scheduling

## 5. Experimental Results

We demonstrate the effectiveness of the proposed methodology on two SMP benchmarks based on a publicly available RISC processor model, the OpenRISC 1200 [11]: a two-core SMP, which we call *OpenRISC Duo* and a four-core SMP, which we call *OpenRISC Quad*. Table 2 lists the characteristics of the SMP benchmarks. They have been synthesized in a 0.18um library.

Characteristic	OpenRISC Duo	OpenRISC Quad
Processors	2	4
L1 Cache	8 KB	8 KB
Write Policy	Write Through	Write Through
Block Size	16-Bytes	16-Bytes
Shared L2 cache	128 KB	256 KB
Bus	Wishbone	Wishbone
Coherency	Snooping	Snooping
Gate Count	80897	162189
Fault Count	189584	379900

Tab. 2: SMP benchmarks characteristics

Tables 3 and 4 show fault simulation results for the OpenRISC Duo and Quad, respectively. For the OpenRISC core we used the self-test routines proposed in [5]. Detailed results are provided for the components of each individual core (processors, cache controllers, write buffers) and the shared logic (bus arbiter). Fault coverage slightly differs among the cores of the SMPs because the self-test routines are executed in different order in each core. Our methodology achieves more than 91% total stuck-at fault coverage for both SMP benchmarks.

Module	Faults	Fault Coverage (%)		
		Core1	Core2	Total
Processors	2 × 83660	90.79	90.81	90.80
I-Cache Ctrls	2 × 1238	91.63	91.57	91.60
D-Cache Ctrls	2 × 1698	90.21	90.18	90.20
Write Buffers	2 × 7590	94.15	94.10	94.13
Bus Arbiter	1212	90.35		90.35
<b>Total</b>	<b>189584</b>	<b>91.06</b>		

Tab. 3: Fault simulation for the OpenRISC Duo

Module	Faults	Fault Coverage (%)				
		Core1	Core2	Core3	Core4	Total
Processors	4 × 83660	90.78	90.79	90.82	90.80	90.80
I-Cache Ctrls	4 × 1238	91.59	91.62	91.43	91.69	91.58
D-Cache Ctrls	4 × 1698	90.16	90.03	90.16	90.23	90.15
Write Buffers	4 × 7590	94.12	93.65	94.22	94.31	94.08
Bus Arbiter	3156	90.43			90.43	
<b>Total</b>	<b>379900</b>	<b>91.05</b>				

Tab. 4: Fault simulation for the OpenRISC Quad

Self-test approaches	Code size (words)		Test responses (words)	
	Duo	Quad	Duo	Quad
SC-SE	3271	3271	3582	7164
SC-PE	3543	3543	1791	1791
MC-PE	6566	13084	3582	7164
<b>Proposed</b>	<b>3327</b>	<b>3283</b>	<b>3582</b>	<b>7164</b>

Tab. 5: Comparison of memory size

Table 5 compares the proposed approach with the other functional self-testing alternatives in terms of memory footprint (test code, data and responses size). The superiority of the proposed approach against MC-PE is obvious. Compared with the two SC approaches, the proposed approach slightly increases the test program size compared to the SC-SE case because of the test scheduler and needs more memory than the SC-PE (requires smaller test program but generates much more test responses). Note that there is a difference in test responses between the SC-PE and all other approaches because the former accumulates the test responses of a core for a given test routine with the corresponding test responses of the other cores. The drawback of the SC-PE approach is that it lacks fault identification capability.

Figure 4 compares the self-test execution times (in clock cycles) of the different alternatives. The superiority of the proposed approach against SC-SE is obvious.

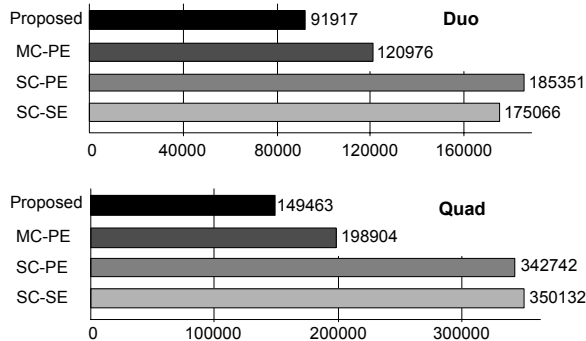


Fig. 4: Comparison of test execution times (cycles)

The effectiveness of the proposed approach is demonstrated when we compare it against the PE approaches. A detailed analysis of the results showed that in the case of SC-PE the portion of the execution time reduction that is due to cache invalidations is about 70% and the remaining 30% is due to bus contention. In the case of MC-PE, the improvement is only due to bus contention reduction since cache invalidations do not occur. Note that in the MC-PE approach a major drawback is the excessive time for the download of the multiple program copies.

## 6. Conclusions

In this paper we propose an optimization methodology for the application of functional self-testing on bus-based symmetric multiprocessors (SMPs) aiming the efficient exploitation of the parallelism of these architectures. We demonstrate the effectiveness of functional self-testing approaches (previously applied to uniprocessors) to SMP architectures and propose a methodology that focuses on the reduction of the test application time by minimizing the time overheads due to cache coherency invalidation problems and intense bus contention among the processors. The benefits of the methodology are clearly demonstrated through detailed experiments on two-core and four-core SMP benchmarks.

## References

- [1] F.Corno, M.Sonza Reorda, G.Squillero, M.Violante, "On the Test of Microprocessor IP Cores", in *Proc. Design Automation & Test in Europe*, pp.209 – 213, 2001.
- [2] L.Chen, S.Ravi, A.Raghunathan, S.Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", in *Proc. IEEE/ACM Design Automation Conf.*, pp. 548 – 553, 2003.
- [3] N.Kranitis, A.Paschalis, D.Gizopoulos, G.Xenoulis, "Software-based self-testing of embedded processors", *IEEE Trans. on Computers*, vol. 54, Issue 4, pp. 461 – 475, 2005.
- [4] S.Gurumurthy, S.Vasudevan and J.Abraham, "Automated Mapping of Pre-Computed Module-Level Test Sequences to Processor Instructions", *Proc. IEEE Intl. Test Conf.*, paper 12.3, 2005.
- [5] M.Psarakis, D.Gizopoulos, M.Hatzimihail, A.Paschalis, A.Raghunathan, S.Ravi, "Systematic Software-Based Self-Testing of Pipelined Processors", in *Proc. IEEE/ACM Design Automation Conf.*, pp. 393-398, 2006.
- [6] V. Singh, M. Inoue, K. Saluja, H. Fujiwara, "Instruction-Based Self-Testing of Delay Faults in Pipelined Processors" *IEEE Trans. on VLSI Systems*, vol.14, no.11, pp. 1203-1215, 2006.
- [7] P.Parvathala, K.Maneparambil, W.Lindsay, "FRITS—A Microprocessor Functional BIST Method", in *Proc. IEEE Intl. Test Conf.*, pp. 590 – 598, 2002.
- [8] I.Bayraktaroglu, J.Hunt, D.Watkins, "Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues", in *Proc. IEEE Intl. Test. Conf.*, paper 27.2, 2006.
- [9] M.Riley, L.Bushard, N.Chelstrom, N.Kiryu, S.Ferguson, "Testability Features of the First-Generation Cell Processor", in *Proc. IEEE Intl. Test. Conf.*, paper 6.1, 2005.
- [10] M. Tripp, S. Picano, B. Schnarch, "Drive Only At Speed Functional Testing; on of the techniques Intel is using to control test costs", in *Proc. IEEE Intl. Test. Conf.*, paper 6.3, 2005.
- [11] OpenCores Projects [www.opencores.org/projects/](http://www.opencores.org/projects/)
- [12] J.L.Hennessy, D.A.Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Elsevier, 2006.
- [13] R.B. Cooper, *Introduction to Queuing Theory*, Elsevier 1981.