MT-SBST: Self-Test Optimization in Multithreaded Multicore Architectures

N. Foutris[1] M. Psarakis[1] D. Gizopoulos[1] A. Apostolakis[1]
X. Vera[2] A. Gonzalez[2]

[1] University of Piraeus, Department of Informatics, Greece
{nfoutr | mpsarak | dgizop | andapo}@unipi.gr
[2] Intel Barcelona Research Center, Intel Labs-UPC, Barcelona, Spain
{xavier.vera | antonio.gonzalez}@intel.com

Abstract

Instruction-based or software-based self-testing (SBST) is a scalable functional testing paradigm that has gained increasing acceptance in testing of single-threaded uniprocessors. Recent computer architecture trends towards chip multiprocessing and multithreading have raised new challenges in the test process. In this paper, we present a novel self-test optimization strategy for multithreaded, multicore microprocessor architectures and apply it to both manufacturing testing (execution from on-chip cache memory) and post-silicon validation (execution from main memory) setups. The proposed self-test program execution optimization aims to: (a) take maximum advantage of the available execution parallelism provided by multiple threads and multiple cores, (b) preserve the high fault coverage that single-thread execution provides for the processor components, and (c) enhance the fault coverage of the thread-specific control logic of the multithreaded multiprocessor. The proposed multithreaded (MT) SBST methodology generates an efficient multithreaded version of the test program and schedules the resulting test threads into the hardware threads of the processor to reduce the overall test execution time and on the same time to increase the overall fault coverage. We demonstrate our methodology in the OpenSPARC T1 processor model which integrates eight CPU cores, each one supporting four hardware threads. MT-SBST methodology and scheduling algorithm significantly speeds up self-test time at both the core level (3.6 times) and the processor level (6.0 times) against single-threaded execution, while at the same time it improves the overall fault coverage. Compared with straightforward multithreaded execution, it reduces the self-test time at both the core level and the processor level by 33% and 20%, respectively. Overall, MT-SBST reaches more than 91% stuck-at fault coverage for the functional units and 88% for the entire chip multiprocessor, a total of more than 1.5M logic gates.

1. Introduction

The physical limits of semiconductor microelectronics have become a major concern in manufacturing technology. The diminishing gains in processor's performance due to the increasing gap between processor and memory speed (*memory wall*), the absence of enough parallelism in single instruction streams (*ILP wall*) and the escalation in power consumption (*power wall*) motivate computer architects to look at different directions for next processor generations.

Current microprocessor industry trend is towards the development of chip multiprocessors (CMP) and chip multithreaded (CMT) architectures which although operate at lower frequencies are able to deliver higher performance exploiting thread-level (*intra-core*) or processor-level (*inter-core*) execution parallelism. However, to cope with this industry trend, the test technology community has to explore the effective porting from the uniprocessor era to the multiprocessor era (CMP and CMT architectures) of all test and validation techniques, that have been recently devised to deal with the emerging reliability problems of modern microprocessors [1], [2]. The main objective of this porting of test techniques to multithreaded multiprocessors should be the exploitation of the execution parallelism of the new processor architectures to avoid excessive scaling of the overall test time. As a consequence, it will reduce test cost and improve time-to-market without degrading the effectiveness of the test techniques in terms of their fault detection capabilities (fault coverage).

Software-Based Self-Testing (SBST) [3]–[15] is a testing method that has gained increasing acceptance with major microprocessor vendors and today forms an integral part of the manufacturing test flow of single-threaded processors. The SBST key idea is to exploit the instruction set architecture and on-chip programmable resources to execute effective self-test programs. The use of SBST methodologies contributes to the reduction of yield loss, while its non-intrusive nature does not require any processor hardware modification. In addition, at-speed testing ability enables screening of timing defects that do not manifest themselves at lower frequencies [12].

The effective application of SBST to multithreaded multicore architectures poses significant challenges: (i) porting of existing test programs from the single-threaded, unicore case to efficiently test *all* the individual cores; (ii) providing sufficient fault coverage for the *thread-specific control logic*, which is a significant portion of the control logic in the multithreaded architectures; (iii) exploitation of *thread-level* and *core-level parallelism* to reduce test execution time; and (iv) avoiding the scaling of test program *memory footprint* with the number of cores.

Software-based approaches for the manufacturing testing of CMP and CMT architectures have been proposed in [13], [14], and [15]. Bayraktaroglu *et al*. [13] proposed the conversion of existing legacy tests, either hand-written or randomly-generated to test the multithreaded cores of the CMT architecture of UltraSPARC T1. They described how a software-based *cache-resident* test methodology can be utilized during the manufacturing test flow of a

978-1-4244-7207-9/10/$26.00 ©2010 IEEE

commercial multicore chip, UltraSPARC T1, and applied by a low-cost external tester. In [13], the CPU cores of the CMT architecture execute the test program *sequentially* while the other cores are disabled; this scheme eliminates the need for replicating the test program for each processor core but it does not exploit either the core-level parallelism or the thread-level parallelism of the architecture, thus, it does not satisfy one of the main objectives of a multithreaded SBST methodology. Apostolakis *et al*. [14] considered the application of SBST to bus-based CMP architectures consisting of multiple single-threaded cores of OpenRISC 1200 processor. A scheduling methodology has been proposed for the test routines to exploit core-level parallelism and minimize the time overheads coming from the memory subsystem in order to reduce the total test execution time. The work of [14] exploits only the core-level execution parallelism of a single-threaded CMP architecture. A first approach on the application of SBST in the CMT architecture of OpenSPARC T1 processor for manufacturing testing only, was proposed in [15] where thread-level parallelism is exploited to reduce self-test execution time. In [15], the impact of the test program scheduling in the fault coverage of the thread-specific control logic and the shared functional units of the OpenSPARC T1 processor were not taken into consideration. Overall, none of the above approaches considers the case of self-test program execution from main memory (as in a post-silicon validation setup), where the cache-residence limitation does not apply and a main memory subsystem is available to store the test program.

In this paper, we present, for first time, a complete *multithreaded software-based self-testing* (MT-SBST) methodology that targets *both* the optimization of test execution time *and* the improvement of the fault coverage of the thread-specific control logic. First, we assess the impact of test routine scheduling in the fault coverage of hard-to-test control structures: the *thread-switch logic* inside the processor cores and the *thread-specific control logic* of the shared components outside the processor cores. Subsequently, we propose a multithread scheduling algorithm that achieves a very efficient balance between self-test program execution time and fault coverage of the thread-specific control logic. The algorithm is *solely* based on easy-to-obtain runtime statistics of the single-threaded execution of the self-test program. In particular, our proposed MT-SBST methodology performs the following:

- Test program *development* for all the functional units of a CMT multiprocessor architecture.
- Test program *profiling,* without multiple time-consuming simulations, from single-threaded unicore execution.
- Assessment of the *impact* of the multithreaded execution of test program on the fault coverage of the thread-specific control logic.
- Test program scheduling to take advantage of *thread-level parallelism* and speedup execution of its test routines for the on-core functional units, and *core-level parallelism* to speedup the execution of its test routines for the off-core shared functional units. At the same time, our scheduling improves the fault coverage for those structures.

We fully apply the proposed methodology in a complex publicly available CMT processor architecture, OpenSPARC T1 [16] consisting of 8 cores and 32 threads. Our experimental results show that the proposed multithread scheduling algorithm significantly speeds up the execution time of test program at both the core-level (up to 3.6X) and the processor-level (up to 6.0X) compared to the single-threaded execution. Furthermore, compared to straightforward multithreaded execution of the test program the proposed multithread schedule reduces test execution time at the core-level and the processor-level by more than 33% and 20%, respectively. On top of these significant speed improvements, and despite its much shorter execution time, the proposed MT-SBST schedule improves the fault coverage of the thread switch logic of each core by about 10% compared to the straightforward multithreaded version. Overall, our methodology guarantees high stuck-at fault coverage levels: more than 91% for the functional units (all integer functional units of the eight cores and the off-core shared floating point unit) and more than 88% for the logic of the entire processor (including the functional units, the thread switch logic and the interconnection networking, which all together count about 1.5M logic gates).

The rest of the paper is organized as follows: Section 2 provides an overview of SBST, for single threaded and multithreaded architectures. Section 3 provides a detailed analysis of the proposed MT-SBST methodology and Section 4 presents the experimental results. Finally, Section 5 concludes the paper.

2. Software-Based Self-Testing Overview

2.1 SBST of single-threaded processors

The basic concept of software-based self-testing (SBST) for a single-threaded uniprocessor is described in detail in [12], along with its position in the testing process. A test program is executed by the processor at normal mode of operation. The test instruction sequences usually load test patterns from memory (or generate them internally) and apply operations to excite faults in hardware components. Fault propagation is performed executing instructions that store test responses into data memory, from which they can be uploaded and evaluated by an external (low-cost) tester. A key task of an SBST methodology is the generation of test instruction sequences that can effectively test the processor modules and reach high fault coverage. Several recent works have proposed efficient test program generation methodologies that target different modules of single-threaded microprocessor cores, such as

integer functional units [4], [5], [6], pipeline control logic [7], [8], speculative mechanisms [9] and floating-point units [11]. Today, SBST forms an integral part of the manufacturing test flow [3], [13] of top-end processors, and its role is complementary to other traditional testing methods, either structural like scan-based test or BIST, or functional using external testers [12]. SBST improves the overall test quality without requiring any hardware modification or extra test equipment.

SBST is also a potentially effective solution for post-silicon validation. Execution of verification tests in early silicon prototypes is orders of magnitude faster than simulation-based verification tests and this enables designers to apply more comprehensive tests within a limited time period [17]. However, developing testbenches for post-silicon validation is a tedious task since it suffers from limited internal node observability compared to the full signal observability that a pre-silicon, simulation-based environment offers. This problem is exacerbated in multithreaded, multicore architectures because of their more complex control logic (for thread scheduling and synchronization) and memory subsystems (cache coherence mechanisms) [18]. Therefore, utilization of self-test programs from manufacturing testing as a starting point or using SBST methodologies to enhance the controllability and observability of functional verification tests (i.e. legacy tests) [19] could be a very efficient solution for the generation of effective post-silicon validation tests.

Consequently, SBST can be a key part of an efficient flow for manufacturing testing and post-silicon validation stages, and in this paper we study both cases. The SBST experimental setup for these two stages differs in the storage device where the self-test program resides. In a manufacturing testing setup, test code and data are downloaded into on-chip caches (instruction and data) by an external tester and the test responses are also stored into the on-chip data cache [13]; after test program execution, the tester uploads the test responses to compare them with the golden signatures. This *cache-resident* setup eliminates the need for high-cost functional testers and speeds up the execution of self-test program. However, it imposes a restriction in the development of self-test program, that no cache misses occur during its execution. On the contrary, in a post-silicon validation setup of prototype chips, no such limitation exists, since a main memory subsystem is usually available for full system-level testing. The test code and data are stored in main memory and this also allows the execution of the much larger programs used in post-silicon validation. In this paper, we utilize the same self-test programs for manufacturing testing and post-silicon validation, and study how the two different setups affect the multithreaded execution of self-test programs. Of course, in the case of post-silicon validation setup, the main memory transactions induce longer waiting intervals in the execution of self-test programs compared to the manufacturing testing setup. The proposed test scheduling algorithm efficiently exploits the waiting intervals to speedup execution of self-test threads in both cases and generates different schedules.

It should be noted that the total test application time, both in manufacturing testing and post-silicon validation, consists of the test program download time, the test program execution time and the test responses upload time, which are affected by the underlying architecture, the cache access interface bandwidth, and the test routine structure. Our methodology primarily focuses on test program execution time optimization, but decent gains are also obtained in download time due to the single copy of test program. Further reduction of upload and download time could be achieved using test program compression and test response compaction techniques.

2.2 Multithreaded (MT) SBST Preliminaries and Experimental Setup

For the application of SBST in a multithreaded multicore architecture, we assume the following setup:

- A test program consists of a set of test routines that target all the *private* functional units of *each* processor core (i.e. functional units in the execution pipeline of each core such as ALU, multiplier, divider and shifter) and the off-core *shared* functional units (i.e. a floating-point unit that all cores share).
- A *single copy* of the test program (test code and data) is stored in memory (either on-chip cache or main memory depending on the setup) instead of separate copies for each core; this reduces the storage requirements and avoids the scaling of test program memory footprint with the number of cores. All processor cores have to execute the same test program to detect faults in their private units while the self-test program for the shared units must be executed once (by one or more cores).
- Each processor core generates a set of *separate* test responses; this assumption enables faulty core diagnosis. Diagnosis capability is important for both manufacturing testing and post-silicon validation since it allows the binning of partially "good" chips.

In order to reduce the execution time in an MT-SBST approach, we need to take advantage of both the available thread-level and core-level parallelism, visualized in Figure 1. Let assume four test routines for the functional units FU_1, FU_2, FU_3, and FU_4 of the processor core (these routines must be executed by each core) and one test routine for a shared functional unit (this routine must be executed once). Exploitation of core-level parallelism enables the parallel execution of the test routines FU_1, FU_2, FU_3 and FU_4 by all n processor cores and speeds up the execution of the shared-FU test routine. If execution parallelism is not exploited, the overall test execution time will scale with the number of processor cores (8 in T1 multiprocessor). Instead of having a single core to execute

the shared-FU routine (top of Figure 1), the routine is split into *n* subroutines which can be executed in parallel (middle of Figure 1). We can schedule in a different way the test routines in the *n* cores to achieve the optimum utilization of the common memory subsystem and the interconnection network [14]. Next, we exploit thread-level parallelism to speedup the execution of the test routines in each core; assuming that each core supports four hardware threads in an interleaved multithreading fashion, all four threads are used to execute the test routines as shown in Figure 1 (bottom). The overlap of the idle intervals of one thread (i.e. due to a long latency operation or a cache miss) by another active thread is the key point for the efficient parallelization of test routines.

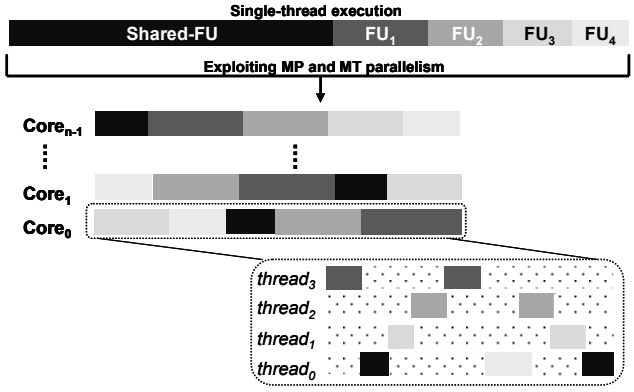

Figure 1: Exploiting MP and MT parallelism in the execution of the test program

3. Proposed MT-SBST Methodology

When normal applications are developed for a multithreaded architecture the main focus is the maximization of the application throughput and the processor resources utilization. The tuning of the application workload depends on its specific characteristics. In this paper, we aim to tune self-test program to the characteristics of multithreading technology to achieve the maximum speedup, that – as our experiments reveal – a naïve, straightforward multithreading schedule is not able to reach. Even in the case of a small number of cores and threads per core, the exhaustive search of the test program scheduling is infeasible and a high-level test scheduling algorithm based on simple single-thread runtime statistics is required.

The main objectives of the proposed methodology are: (a) to develop test routines for the functional units of the processor; (b) to assess the test program execution characteristics for its efficient tuning towards a multithreaded architecture; (c) to analyze how the multithreaded execution of the test program affects the fault coverage of the thread-specific control logic (which is not explicitly targeted by the test routines for the functional units); and (d) to propose an efficient scheduling algorithm which reduces test program execution time *without degrading* its effectiveness in terms of fault coverage for the related logic. Overall, the main goal of our methodology is to achieve the *best tradeoff point between self-test time reduction and self-test effectiveness for the thread-specific control logic*. The steps of the methodology are summarized in Figure 2 and individually analyzed in the following subsections.

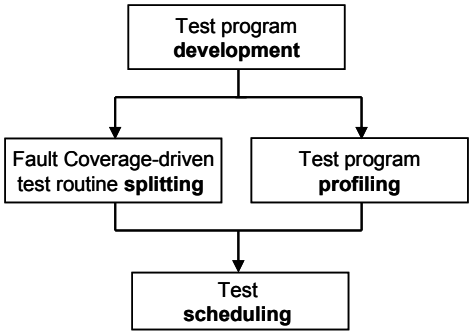

Figure 2: Proposed MT-SBST methodology

3.1 Test program development

Our demonstration vehicle is the open-source CMT processor model, OpenSPARC T1, which integrates eight 64-bit SPARC V9 processor cores, each supporting four hardware threads [16]. Figure 3 shows the organization of the OpenSPARC T1 processor. Each CPU core implements a six-stage, single-issue execution pipeline and has 16KB L1 instruction cache and 8KB L1 data cache. An on-chip unified 3MB L2 cache divided in four banks is shared among all CPU cores. A crossbar switch handles communication between the CPU cores and the shared memory while at the same time it provides access to a shared floating-point subsystem. OpenSPARC T1 implements fine-grain multithreading: it switches among the available threads at every cycle giving priority to the least recently executed thread.

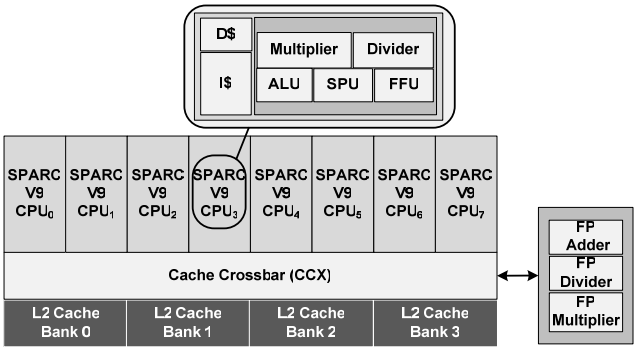

Figure 3: OpenSPARC T1 organization

The first step of the proposed methodology is the development of test routines that target *all* the private functional units of each SPARC V9 core: ALU, shifter, integer multiplier, integer divider, stream processing unit (SPU), and floating-point frontend unit (FFU). The test routines for these six functional units must be executed by all processor cores. We also develop separate test routines for the components of the off-core shared floating-point

unit (FPU – FP adder, multiplier, divider) of OpenSPARC T1 each of which must be executed only once.

For a few functional units, like the shifter and the multiplier we adopted proven effective optimized test sets from previous SBST approaches [6], [8] and tuned them to the functional units of SPARC V9 core. For the other modules, we either developed customized test routines (like in the cases of the ALU and the divider) or enhanced the regression tests of the modules (like in the cases of FFU and SPU) included into OpenSPARC T1 verification suite. This first step of self-test program development does not affect the operation of the subsequent steps. Thus *any* self-test program for the individual integer and floating-point units can be used.

Table 1 summarizes the characteristics of the functional units of the SPARC V9 core and the corresponding test routines. Second column presents the gate count of the functional units and third column gives the fault coverage obtained in a single-thread execution (results are for stuck-at fault model using Synopsys' TetraMAX).

Functional units	Gate count (K gates)	Fault coverage (stuck-at %)	Single-thread execution time (K cycles)	
			Manufacturing testing	Post-silicon validation
Shifter	5.9	97.5	14.4	45.2
ALU	6.2	92.7	32.5	61.4
Divider	11.4	97.3	54.5	78.4
Multiplier	54.2	96.4	8.6	17.7
FFU	16.6	72.1	9.9	18.3
SPU	18.5	86.9	33.1	45.4
Total	**112.8**	**91.2**	**153.0**	**266.4**

Table 1: Private functional units and corresponding test routines of SPARC V9 core

The two rightmost columns show the test routine execution time in a single thread for: (a) manufacturing testing (execution from on-chip shared L2 cache) and (b) post-silicon validation (execution from main memory). The execution time of the test routines depends on: the number of test patterns, the latency of the corresponding instructions and the development style (which affects the instruction-level parallelism of the routines – loops, etc). Our test program achieves more than 91% fault coverage in total for all the functional units, the highest structural fault coverage that has been reported by a software-based testing approach on a real open-source industrial processor such as OpenSPARC T1.

In Table 2 we present the effectiveness of the FPU routine in terms of stuck-at fault coverage only for the execution pipelines (adder, multiplier, divider) of the shared floating-point unit. We deal with the control part of the FPU later. The developed FPU routine achieves more than 92% stuck-at fault coverage for this complex functional unit. The total execution time of FPU routine is 2.6M clock cycles when executed from on-chip shared L2 cache (manufacturing testing) and 2.9M clock cycles when executed from main memory (post-silicon validation).

Modules	Gate count (K gates)	Fault coverage (stuck-at %)	Single-thread execution time (K cycles)	
			Manufacturing testing	Post-silicon validation
FP Add.	33.7	91.7	1300.1	1450.2
FP Mult.	60.1	92.9	520.4	580.5
FP Div.	13.6	91.0	780.2	870.2
Total	**107.4**	**92.3**	**2600.7**	**2900.9**

Table 2: Modules of the shared off-core floating point unit and corresponding test routines

The fault coverage of the individual functional units remains the same when the corresponding test routines are executed in a multithreaded fashion. However, this is not the case for the control logic, either the thread-specific control logic of the core or the shared FPU control logic. In Section 3.3 we analyze how the multithreaded execution affects the fault coverage of these control modules. We aim to propose a multithreaded execution of our test routines that although reduces the total test execution time it does not reduce the fault coverage on this control logic. We discuss our scheduling algorithm in Section 3.4.

3.2 Test program profiling

The second step of the methodology is the high-level profiling of the single-thread version of the test program that allows us to quickly assess its scaling characteristics to a multithreaded environment. All test routines are executed in a single hardware thread of one SPARC V9 core that has exclusive access to the core resources while the other three threads are parked (i.e. exclusive single-thread performance). Figure 4 shows the exclusive single-thread performance of all test routines for the two different SBST setups (note that routines DIV, FFU and SPU have been divided into two subroutines; at the end of this subsection we explain why we chose to split these routines). Each bar represents the fractions of time the state machine of the hardware thread, executing the corresponding test routine, stays in one of the five possible states: *ready*, *run*, *wait*, *speculative ready*, and *speculative run*. The SPARC V9 core switches among the available threads at every cycle (i.e. fine-grain multithreading). A thread can be scheduled (is available) when it is in one of the following states: ready, speculative ready, run, and speculative run[1]. On the other hand, a thread enters the wait state due to one of the following reasons: I-cache fill, store buffer full, long latency operation, and resource conflict (i.e. simultaneous requests to a shared resource). Therefore, when executing the test routines in a single-thread, the core enters a wait state when the thread is unavailable. To collect runtime statistics for the thread state we used the functionality of the thread monitor unit of SPARC V9 core.

[1] A thread speculates a load operation as a cache hit before the actual request is granted from the memory subsystem.

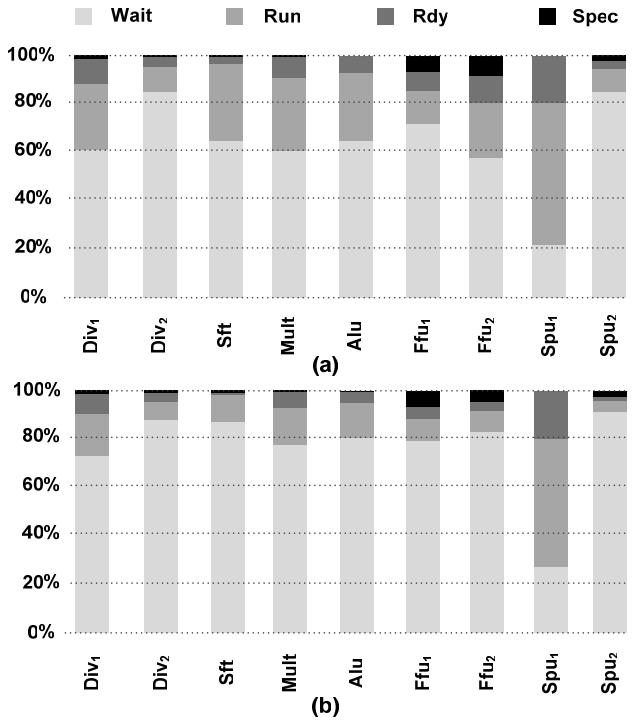

Figure 4: Test program profiling: exclusive single-thread execution for (a) manufacturing testing (b) post-silicon validation

Test program profiling shows that the total core utilization is very low since the core is waiting for long time intervals because the thread is unavailable. In the case of manufacturing testing (Figure 4a) the thread is in wait state for the 62% of the total execution time of the test program. In the case of post-silicon validation (Figure 4b) the time that the thread waits increases due to the longer penalty of L2 cache misses and accesses to main memory; the thread is in wait state more than 77% of the total execution time. Hence, the test program profiling stage designates the ability for performance gains when routines are scheduled in an optimized multithreaded fashion.

We further analyzed the profiles of the test routines to identify different execution phases, such as CPU-bound or memory-bound intervals, within a test routine execution and then we split it into more than one subroutines based on these phases. This splitting procedure enables us to schedule more efficiently the test routines into the hardware threads achieving better exploitation of thread-level parallelism (TLP). In our study, routines Div, FFU and SPU, present different runtime statistics at different execution phases and are split into two subroutines each, Div_1 (24.2 K) and Div_2 (30.3 K), FFU_1 (9.4 K) and FFU_2 (0.5 K) and SPU_1 (23.8 K) and SPU_2 (9.3 K), respectively (execution from L2 cache in clock cycles).

3.3 Coverage-driven test routine splitting

We study the effect of multithreaded execution of the test routines on the fault coverage of the on-core (thread-switch logic) and off-core (shared FPU) control logic.

On-core control logic (thread-switch logic). Thread-switch logic fault coverage increases with the activity of the four thread state machines. Thus, to increase the fault coverage of the thread-switch logic, we should avoid decreasing the number of state transitions of the thread state machines by forcing the four threads to enter more times in the wait state. However, this target contradicts with the test execution time reduction goal since increasing the number of resource conflicts (i.e. simultaneous requests to a shared resource) will adversely affect the exploitation of CMT.

We consider two routines from our basic core test program that can cause resource conflicts due to their long latency operations: multiplier and divider routines. We performed a set of fast, high-level experiments to quantify the speedup achieved if we split these test routines into two or four time-balanced subroutines and schedule two or four hardware threads to execute them in parallel. In Table 3, we compare the time of the single-threaded execution versus the multithreaded execution for these two routines for execution from L2 cache and main memory.

Testing setup	Routines	1 thread	2 threads		4 threads	
		ET (A) K cycles	ET (B) K cycles	Speedup (A/B)	ET (C) K cycles	Speedup (A/C)
Manuf. testing	Multiplier	8.6	5.7	1.5	5.4	1.6
	Divider	54.5	37.1	1.5	35.9	1.5
Post-si. validation	Multiplier	17.7	9.3	1.9	8.3	2.1
	Divider	78.4	43.5	1.8	39.3	2.0

Table 3: Single-threaded execution vs. multithreaded execution (ET: execution time)

The experimental results show that the two-threaded execution achieves significant speedup over the single-threaded execution which ranges between 1.5X and 1.9X. However, the speedup saturates at two threads since using more than two threads only slightly reduces execution time. Therefore, to improve the fault coverage of the thread-switch logic during the multithreaded execution we split the long-latency routines into subroutines that generate resource conflicts when executed in multithreaded mode. However, to achieve the best tradeoff between execution time reduction and fault coverage of the thread-switch logic the number of subroutines must not exceed the number of threads at which the speedup saturates. The output of this step is a number of sets each one containing the appropriate number of subroutines that must be executed in parallel to cause resource conflicts. In the case of the multiplier and the divider two sets are created: $\{Div_1, Div_2\}$ and $\{Mult_1, Mult_2\}$.

Off-core control logic (shared FPU). We exploit core-level parallelism to execute the test routines for the off-core shared FPU. In order to determine an efficient multicore, multithreaded execution of FPU test routine we study how the execution time and the fault coverage scale with the number of cores and threads that execute the test routines. Thus, we split FPU test routine into 4, 8, 16, and

32 subroutines and schedule them to different number of processor cores: 1, 4 or 8 cores each running 1 or 4 threads. At this point it should be noted that partitioning the test routines in arbitrary number of subroutines with 'almost' equivalent execution times, depends on code style that have been adopted in the development of test routines (e.g. load-apply-store routines, such FPU routine, allow this partitioning). Table 4 presents total execution time and combined stuck-at fault coverage of the two FPU control submodules: *FP input* that multiplexes the FPU requests from multiple cores and *FP output* that arbitrates the results of FP pipelines for the single FPU-crossbar connection. Also, Table 4 presents results for both the execution from L2 cache (manufacturing testing) and main memory (post-silicon validation). Our experiments demonstrate that the fault coverage is affected by the execution of FPU test routine by multiple cores and multiple threads. This happens because the FPU control modules carry *thread and core id specific information*. The results suggest that the most efficient FPU routine schedule in terms of speedup and fault coverage in both setups is 8 cores each running 4 threads: a total of 32 hardware threads executing in parallel 32 different FPU time-balanced subroutines. Thus, in our proposed test scheduling the FPU test subroutines *are executed in parallel by all processor cores* – separately from basic core test routines – occupying all 32 threads of the CMT architecture.

Schedule	1 thread		4 threads	
	ET (K cycles)	FC (%)	ET (K cycles)	FC (%)
Manufacturing Testing				
1 core	2600.7	61.9	1400.1	62.7
4 cores	920.1	89.9	490.3	91.0
8 cores	519.2	90.9	437.4	91.6
Post-silicon validation				
1 core	2900.9	62.3	1600.7	63.2
4 cores	1000.5	90.5	517.8	91.2
8 cores	563.1	91.2	460.5	92.3

Table 4: Multicore, multithreaded execution of FPU test routine (ET: execution time, FC: fault coverage of FPU control logic)

3.4 Test scheduling algorithm

We propose an algorithm that schedules a set of on-core components test routines $\{R_1, R_2, \ldots, R_N\}$ into *k* hardware threads targeting the best tradeoff between test execution time and fault coverage. The proposed algorithm is presented in Figure 5.

The first part of the algorithm partitions test routines into two groups: G_L which contains routines having waiting time fraction (WT) less than the average waiting time fraction (WT_{avg}) of all test routines and G_H which contains routines having WT more than WT_{avg}. Then, the two groups are sorted in descending order according to the execution time (ET) of their routines (WT, WT_{avg} and ET values are calculated during test program profiling).

The second part of the algorithm picks test routines from the two groups and iteratively assigns them into threads. The long test routines (with the higher ET) are scheduled first in order to produce a time-balanced scheduling. When a routine that belongs to a *resource conflict group* (RCG) (an RCG contains routines that perform concurrent requests to a shared resource) is selected then all the other elements of the group are scheduled in parallel. If there are routines that can not be scheduled in parallel due to resource limitations they are not selected in the current loop iteration. For instance, routines SPU_1 and SPU_2 can not be executed in parallel since the co-processor implementing the SPU operations supports one outstanding SPU operation per core.

The algorithm satisfies two scheduling criteria: (a) routines that generate resource conflicts (belong to a resource conflict group, RCG) are executed in parallel; and (b) at any time the set of currently executed routines (CXR) contains equal number of low-WT and high-WT test routines. The first criterion aims to improve the fault coverage of the thread-specific control logic and the second criterion aims to overlap the "long" waiting intervals of the half routines with the "running" intervals of the other half routines. The algorithm output is *k* sets SR_{th1}, SR_{th2}, … SR_{thk} that contain the routines scheduled to each thread.

4. Experimental Results

We applied the proposed scheduling algorithm to the test routines of functional units of OpenSPARC T1 for the two different SBST setups: manufacturing testing and post-silicon validation. For the purposes of our evaluation, we also set up a naïve (straightforward) multithreading schedule that assigns routines with the same characteristics to the same thread, i.e. routines using the multiplier (SPU and Mult), divider routines (Div), short latency operations (ALU and Sft) and floating-point operations (FFU and FPU). Both the naïve and the proposed multithreading schedules are based upon the same requirement: to avoid, as much as possible, resource conflicts that degrade test program performance. Therefore, naïve approach constitutes a fair alternative of the proposed approach.

We first analyze core-level thread scheduling without considering testing of the off-core shared FPU. The generated test routine schedules for the two SBST setups and the naïve scheduling approach are shown in Table 5. Each column includes the test routines scheduled in each thread of the core. Notice that the proposed schedules for the two SBST setups are different which is due to the different results of the test program profiling stage.

```
1.  Inputs:   k: number of threads
2.            Basic core test routines: S = {R1, R2, …, RN}
3.            Single-threaded test program profiling results:
4.                  ETi: execution time of routine Ri
5.                  WTi: waiting time fraction of routine Ri
6.                  WTavg: average waiting time fraction of all test routines
7.                  Groups of routines causing resource conflicts: RCG1, RCG2, …, RCGM
8.  Restrictions: Routines cannot be executed concurrently due to limited resources (i.e. SPU1, SPU2)
9.
10. Output:   Sets of scheduled test routines in k threads: {SRth1, SRth2, … SRthk}
11.
12. // Partition routines into two groups:GL (routines with low WT fraction) and GH (routines with high WT fraction)
13. for i = 1, 2, …, N do
14.       if WTi < WTavg  insert Ri to GL ;
15.       else insert Ri to GH ;
16. end for
17.
18. Sort GL and GH in descending order according to ETi
19.
20. ETth1, ETth2, … ETthk = 0 ;        // Accumulated execution times of routines assigned to threads 1…k
21. SRth1, SRth2, … SRthk = Ø ;        // Set of routines scheduled to threads 1…k
22. CXR = Ø ;                          // Set of currently executed routines by all k threads
23.
24. while (GL, GH not empty) do
25.       select thread j with shortest ETthj;
26.       remove the last routine of SRthj from CXR ;        // The last routine has been completed
27.
28.       // Picks up a routine from GH or GL and assigns it to thread j
29.       if (GH empty) OR (# of routines in CXR with low WT < # of routines in CXR with high WT) then
30.             select the longest routine Ri from GL that does not have restriction with any routine of CXR ;
31.             remove Ri from GL ;
32.       end if
33.       if (GL empty) OR (# of routines in CXR with low WT ≥ # of routines in CXR with high WT) then
34.             select the longest routine Ri from GH that does not have restriction with any routine of CXR ;
35.             remove Ri from GH ;
36.       end if
37.       insert Ri to SRthj ;
38.       insert Ri to CXR ;
39.
40.       // Schedules in parallel all routines having resource conflicts with Ri
41.       if Ri belongs to an RCGm then
42.             remove Ri from RCGm ;
43.             while (RCGm not empty) do
44.                   select thread j with shortest ETthj ;
45.                   remove the last routine of SRthj from CXR ;
46.                   select next longest routine Ri from RCGm ;
47.                   remove Ri from RCGm;
48.                   remove Ri from its group (GL or GH) ;
49.                   insert Ri to SRthj ;
50.                   insert Ri to CXR ;
51.             end while
52.       end if
53. end while
```

Figure 5: The proposed core scheduling algorithm

Naïve scheduling	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*
	SPU_1 SPU_2 $Mult_1$ $Mult_2$	Div_1 Div_2	ALU Sft	FFU_1 FFU_2
Proposed scheduling	**Manufacturing Testing**			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	ALU	Div_1 $Mult_2$ FFU_1	Div_2 SPU_2	SPU_1 $Mult_1$ FFU_2 Sft
	Post-silicon validation			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	ALU FFU_2	Div_1 SPU_2 $Mult_2$	Div_2 FFU_1 $Mult_1$	SPU_1 Sft

Table 5: Schedules of core test routines

In Table 6 we compare the proposed multithreaded scheduling with the single-threaded and naïve scheduling approaches in terms of execution time and stuck-at fault coverage of the thread-switch logic (recall that the coverage for the functional units is in all cases more than 91% – see Table 1 – since the coverage does not depend on the multithreaded execution). The speedup of the multithreaded approach is calculated against the test execution time of the single-threaded execution. The speedup obtained by the proposed multithreaded scheduling is up to 3.6X, very close to the ideal theoretical 4X speedup, which means that it exploits the TLP very efficiently, using only easy-to-obtain runtime statistics from the single-threaded execution and avoiding time consuming simulations. Compared with the naive scheduling (that achieves a speedup only up to 2.5X), our methodology reduces the test time by 33%.

	Single threaded		**Naïve scheduling**		**Proposed scheduling**	
	Manuf.	Post-si	Manuf.	Post-si	Manuf.	Post-si
Execution time (K cycles)	153.0	266.4	69.2	107.5	46.1	73.6
Speedup	–	–	2.2	2.5	3.3	3.6
FC (%)	32.6	33.5	67.6	68.3	75.5	77.2

Table 6: Comparison of core level scheduling approaches (FC: Fault coverage of thread switch logic)

Furthermore, the proposed scheduling does not reduce the fault coverage of thread-specific control logic of the core but on the contrary (due to the elaborate routines scheduling algorithm) it improves it up to about 10% compared with the naïve scheduling, thus achieving an excellent tradeoff between speedup and fault coverage.

From this point onward, we include the testing of the control part of the off-core shared FPU (recall that the coverage for the FP adder, multiplier and divider is more than 92% – see Table 2) in our scheduling. In naïve scheduling the FPU routine is split into 8 subroutines ($FPU_{i/8}$) which are executed by thread 3 of each core shown in bold in Table 7. In our approach the FPU test routine is split into 32 time-balanced subroutines ($FPU_{i/32}$) which are executed by all four threads of each core *before* the basic core test routines: all 32 threads of the architecture are occupied to execute in parallel the FPU subroutines. Note that Table 7 presents only the schedules of processor core 0 for the naïve approach and our proposed approach for manufacturing testing and post-silicon validation. The schedules for all processor cores are similarly produced scheduling the corresponding FPU subroutines before the core test routines.

Naïve scheduling	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*
	SPU_1 SPU_2 $Mult_1$ $Mult_2$	Div_1 Div_2	ALU Sft	FFU_1 FFU_2 **$FPU_{1/8}$**
Proposed Scheduling	**Manufacturing Testing**			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	$FPU_{1/32}$ ALU	**$FPU_{2/32}$** Div_1 $Mult_2$ FFU_1	**$FPU_{3/32}$** Div_2 SPU_2	**$FPU_{4/32}$** SPU_1 $Mult_1$ FFU_2 Sft
	Post-silicon Validation			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	$FPU_{1/32}$ ALU FFU_2	**$FPU_{2/32}$** Div_1 SPU_2 $Mult_2$	**$FPU_{3/32}$** Div_2 FFU_1 $Mult_1$	**$FPU_{4/32}$** SPU_1 Sft

Table 7: Schedules of test routines at processor level

Table 8 summarizes test execution time of single-threaded, naïve scheduling and proposed scheduling approaches and the speedup achieved by the multithreaded approaches over the single-threaded one. Compared with the naïve scheduling, the proposed scheduling reduces the test execution time of the entire processor by up to 20%.

	Single threaded		**Naïve scheduling**		**Proposed scheduling**	
	Manuf.	Post-si	Manuf.	Post-si	Manuf.	Post-si
Execution time (K cycles)	2753.7	3167.3	592.4	662.6	492.5	531.1
Speedup	–	–	4.6	4.8	5.6	6.0

Table 8: Comparison of scheduling approaches incl. FPU

Finally, Table 9 presents the fault coverage for the complete targeted logic (about 1.5M gates of logic) of the OpenSPARC T1, which includes all the integer functional units and the on-core control logic (thread-switch logic and integer pipeline control logic) of all eight CPU cores, the off-core shared FPU (including the execution units and the thread-specific control logic) and also the interconnection network (this is not explicitly targeted by

test routines). The total fault coverage for all functional units (both integer and floating-point) is 91.3%, while the total fault coverage for the entire processor is 88.6%. Despite its shorter execution time, the proposed approach achieves higher fault coverage compared with naïve.

Components		Gate count (K gates)	Fault coverage (stuck-at %)		
			Single threaded	Naïve Scheduling	Proposed scheduling
Core (x8)	IFUs	8 × 112.8	91.2	91.2	91.2
	CCL	8 × 28.4	62.2	71.8	82.8
Off-core	FPU	115.8	90.0	92.0	92.3
	INN	259.5	14.9	79.9	82.7
Total (FUs)		**1018.2**	**91.0**	**91.2**	**91.3**
Total (Processor)		**1504.9**	**73.6**	**86.3**	**88.6**

Table 9: Fault coverage (IFUs: Integer Functional Units, FPU: Floating-Point Unit, CCL: Core Control Logic, INN: INterconnection Network, FUs: Functional Units of processor)

5. Conclusions

In this paper, we present the application of SBST in multithreaded, multicore architectures. The proposed MT-SBST methodology leverages the existing thread-level parallelism (TLP) for test optimization. We analyze the impact of multithreaded test execution on fault coverage and propose a flexible methodology to speedup test execution time by exploiting execution parallelism without reducing the fault coverage of the control logic (but on the contrary improving it). Comprehensive experiments on OpenSPARC T1 demonstrate that our methodology speeds up the test time of a 4-threaded core by 3.3 and 3.6 times for manufacturing testing and post-silicon validation, respectively. Compared with a straightforward multithreaded scheduling the proposed methodology achieves significant time reduction, 33% at the core-level and 20% at the processor-level. Overall, our methodology guarantees high fault coverage, more than 91% fault coverage for the functional units and more than 88% for the entire OpenSPARC T1 processor logic (more than 1.5M gates of logic).

References

[1] S.Bockar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", *IEEE Micro*, vol. 25, no. 6, pp. 10-16, Nov. 2005.

[2] C.Constantinescu, "Trends and Challenges in VLSI Circuit Reliability", *IEEE Micro*, vol. 23, no. 4, pp. 14-19, July 2003.

[3] P.K.Parvathala, K.Maneparambil, W.Lindsay, "FRITS–A Microprocessor Functional BIST Method", *IEEE International Test Conference* (ITC), pp. 590 – 598, 2002.

[4] L.Chen, S.Ravi, A.Raghunathan, S.Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", *IEEE/ACM Design Automation Conference* (DAC), pp. 548-553, 2003.

[5] F.Corno, E.Sanchez, M.Sonza Reorda, G.Squillero, "Automatic Test Program Generation – a Case Study", *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 102–109, 2004.

[6] A.Paschalis and D.Gizopoulos, "Effective Software-Based Self-Test Strategies for On-line Periodic Testing of Embedded Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, 2005, pp. 88-99.

[7] S.Gurumurthy, S.Vasudevan and J.Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor", *IEEE International Test Conference* (ITC), paper 27.3, 2006.

[8] D.Gizopoulos, M.Psarakis, M.Hatzimihail, M.Maniatakos, A.Paschalis, A.Raghunathan, S.Ravi, "Systematic Software-Based Self-Test for Pipelined Processors", *IEEE Transactions on VLSI Systems*, vol.16, no. 11, pp 1441-1453, Nov. 2008.

[9] M.Hatzimihail, M.Psarakis, D.Gizopoulos, A.Paschalis, "A Methodology for Detecting Performance Faults in Microprocessor Speculative Execution Units via Hardware Performance Monitoring", *IEEE International Test Conference* (ITC), paper 29.3, 2007.

[10] L.Lingappan, N.K.Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors", *IEEE Transactions on VLSI Systems*, vol. 15, no. 5, pp. 518-530, May 2007.

[11] G.Xenoulis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Instruction-Based Online Periodic Self-Testing of Microprocessors with Floating-Point Units", *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no.2, pp. 124-134, 2009.

[12] M.Psarakis, D.Gizopulos, E.Sanchez, M.Sonza Reorda, "Microprocessor Software-Based Self-Testing", *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4-18 May/June 2010.

[13] I.Bayraktaroglu, J.Hunt, D.Watkins, "Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues", *IEEE Internatioal Test Conference* (ITC), paper 27.2, 2006.

[14] A.Apostolakis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors," *IEEE Transactions on Computers*, vol. 58, no. 12, 2009, pp. 1682-1694.

[15] A.Apostolakis, M.Psarakis, D.Gizopoulos, A.Paschalis, I.Parulkar, "Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors", *IEEE European Test Symposium* (ETS), pp. 33-38, 2009.

[16] OpenSPARC T1 Microarchitecture Specification, Sun Microsystems Inc., Aug. 2006.

[17] H.Rotithor, "Postsilicon Validation Methodology for Microprocessors", *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 77-88, October/December 2000..

[18] A.DeOrio, I.Wagner, V.Bertacoo, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs", *IEEE International Symposium on High Performance Computer Architecture* (HPCA), pp. 405-416, Feb. 2009.

[19] O.Guzey, Li-C.Wang, J.Bhara, "Enhancing signal controllability in functional test-benches through automatic constrain extraction", IEEE International Test Conference (ITC), paper 19.2, 2007.