DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors

Michael A. Skitsas, *Student Member, IEEE,* Chrysostomos A. Nicopoulos, *Member, IEEE,* and Maria K. Michael, *Member, IEEE*

Abstract—As technology scales deep into the sub-micron regime, transistors become less reliable. Future systems are widely predicted to suffer from considerable aging and wear-out effects. This ominous threat has urged system designers to develop effective run-time testing methodologies that can monitor and assess the system's health. In this work, we investigate the potential of online software-based functional testing at the granularity of individual microprocessor core components in multi-/many-core systems. While existing techniques monolithically test the entire core, our approach aims to reduce testing time by avoiding the over-testing of under-utilized units. To facilitate fine-grained testing, we introduce DaemonGuard, a framework that enables the real-time observation of individual sub-core modules and performs on-demand *selective testing* of only the modules that have recently been stressed. Moreover, we investigate the impact of the cache hierarchy on the testing process and we develop a *cache-aware selective testing* methodology that significantly expedites the execution of memory-intensive test programs. The monitoring and test-initiation process is orchestrated by a transparent, minimally-intrusive, and lightweight operating system process that observes the utilization of individual datapath components at run-time. We perform a series of experiments using a full-system, execution-driven simulation framework running a commodity operating system, real multi-threaded workloads, and test programs. Our results indicate that *operating-system-assisted selective testing* at the *sub-core level* leads to substantial savings in testing time and very low impact on system performance. Additionally, the cache-aware testing technique is shown to be very effective in exploiting the memory hierarchy to further minimize the testing time.

1 Introduction

The era of nanoscale technology has ushered designs of unprecedented complexity and immense integration densities. Billions of transistors now populate modern multi-core microprocessor chips and the trend is only expected to grow, leading to single-chip many-core systems [1]. Diminutive feature sizes, however, put undue strain on the reliability and long-term endurance of these modern systems [2], [3], [4]. Research by both industry and academia is pointing to the alarming fact that, on top of early-life failures, future designs will be increasingly vulnerable to aging and wear-out artifacts [5], [6].

The issue of aging and gradual degradation necessitates the use of mechanisms that can enable protection against undesired system behavior by facilitating detection, mitigation, and/or recovery from faults throughout the lifetime of the system [7]. Such schemes broadly fall into two categories: (a) Concurrent methods relying on fault-tolerant mechanisms (i.e., redundancy techniques) [8], [9], [10], and (b) Non-concurrent periodic online testing [11], which aims to detect errors that are, subsequently, addressed using various techniques.

Non-concurrent periodic online testing is one methodology used traditionally for the detection of permanent faults (hard failures). Moreover, it can be used for circuit failure prediction within the cores of modern microprocessors, due to either infant mortality reasons (early-life failures), or aging-related factors [12], [13], [14]. Hardware-based schemes, typically using Built-In Self-Testing (BIST) [15], as well as software-based schemes, known as Software-Based Self-Testing (SBST), can be employed for this problem. The SBST technique [16], [17], [18], [19], [20], [21] is an emerging new paradigm in testing that avoids the use of complicated dedicated hardware for testing purposes. Instead, SBST employs the existing hardware resources of a chip to execute specific (software) programs that are designed to test the functionality of the processor. The test routines used in this technique are executed as normal programs by the CPU cores under test. As a result, the major cost of SBST is the *time* overhead incurred by the execution of the appropriate test routines on the CPU. The hardware overhead is either non-existent, or negligible, and no Instruction Set Architecture (ISA) extensions are required.

In this work, we focus on the processor cores of homogeneous multi-/many-core systems [1]. Memory testing, typically performed using memory BIST and error corrections, as well as on-chip interconnect testing, addressed by fault-tolerant routing and BIST methods

• *M.A. Skitsas and M.K. Michael are with the KIOS Research Center and the Department of Electrical and Computer Engineering (ECE), University of Cyprus, Nicosia, Cyprus.*
E-mail: {skitsas.michael, mmichael}@ucy.ac.cy
• *C.A. Nicopoulos is with the Department of Electrical and Computer Engineering (ECE), University of Cyprus, Nicosia, Cyprus.*
E-mail: nicopoulos@ucy.ac.cy

[5], [22], are beyond the scope of this work. Existing SBST mechanisms for multi-core systems [16], [17], [18] periodically test either an entire CPU *core*, or – in more aggressive scenarios – the entire CPU (i.e., all CPU cores). Naturally, this methodology may lead to *over-testing*, since all core modules (e.g., integer ALU, floating-point units, etc.) are tested irrespective of the actual strain they have suffered since the previous testing session. Ideally, one would want to perform selective/targeted testing to the units that have sustained the most strain, and only sporadic testing to under-utilized modules. As the number of cores in multi-core microprocessors increases, the probability that various functional units within the individual cores will be underutilized during certain periods of time increases. Hence, conducting tests at the granularity of entire cores, or CPUs, will increasingly lead to unnecessary testing and associated performance overhead.

This realization serves as the primary motivation for this work: our objective is to reduce the testing time of multi-/many-core systems by only selectively targeting the individual functional units of each CPU core that are stressed the most. By surgically testing individual modules, we can markedly reduce the overall testing time, since the testing procedure will only execute the test routines relevant to the specific unit under test. In order to enable such testing capabilities, the utilization of the various datapath components is observed at run-time and tests are initiated.

Toward this end, and in an effort to facilitate seamless and transparent selective SBST in multi-core systems, we hereby propose **DaemonGuard**, a framework that enables the real-time observation of individual sub-core modules and the initiation of on-demand selective (i.e., unit-specific) testing. By looking inside each core, DaemonGuard performs more frequent testing of over-utilized functional units and periodic, infrequent testing of under-utilized units. It should be noted that the proposed DaemonGuard framework does not impose any restrictions on the metric used to trigger the testing mechanism. *Utilization* is used here as a proof-of-concept surrogate targeting the Hot Carrier Injection (HCI) failure mechanism, which is directly related to the switching frequency and activity factor of the components [23], [24]. In reality, *any* metric may be adopted to account for any form of aging/wear-out. For example, if the underlying architecture provides temperature sensors [25], or any other type of aging sensors, DaemonGuard may utilize them for more accurate estimation of aging. Performance counters may also be used as a proxy for high temperatures, as demonstrated in [26].

The DaemonGuard framework is extremely lightweight and minimally intrusive: it is implemented predominantly in software and relies on minimal hardware support. The main component of DaemonGuard is the *Testing Manager*, which operates in the form of an Operating System (OS) process running transparently alongside other OS processes. The Testing Manager periodically checks for any pending testing requests. The latter are generated by a set of hardware counters keeping track of the utilization of individual functional units within each CPU core. Should a functional unit exceed a pre-defined threshold, the Testing Manager will invoke a *Test Daemon*, which will execute a testing routine on the specific functional unit. DaemonGuard implements the various test routines as *daemons*, i.e., software programs running as background processes without any interaction at the user level. All test routines are loaded onto the system as individual daemons in idle mode, waiting for an appropriate wake-up signal from the Testing Manager.

Furthermore, the DaemonGuard mechanism is able to exploit the memory hierarchy of the CPU to expedite the testing process. While some test programs may be small and could fit in their entirety within the L1 cache, there are also test programs that are much larger (on the order of MB) and, therefore, tend to stress the system's memory. In general, on-line self-testing may require thorough testing to account for a variety of failure modes, such as traditional stuck-at and delay faults. In such cases, it may be necessary to load a considerable amount of test patterns from off-chip memory. Given the trend toward many-core microprocessors with ever-increasing hardware complexity, the memory footprint of such SBST programs is also likely to increase. In this paper, we investigate the impact of the cache hierarchy on the testing time of memory-intensive test programs and demonstrate that the testing coordinator (the *Testing Manager* in our case) cannot be oblivious to the underlying memory sub-system. Hence, DaemonGuard is augmented with the capability to perform *cache-aware* selective testing, whereby test sessions are initiated not only based on unit utilization, but also on the recent history of test sessions by other similar units in other cores. Consequently, test programs can benefit from cache-resident blocks, thereby obviating the need for many expensive off-chip memory accesses.

Overall, the DaemonGuard framework comprises a lightweight, always-active OS process (the Testing Manager) and a number of dormant daemons (the test routines for each functional unit), which are awakened on demand. Note that DaemonGuard is simply an OS plug-in and does not require any modifications to the OS kernel. Similar to the way software *drivers* are provided by hardware vendors, microprocessor manufacturers are envisioned to release *DaemonGuard plug-ins* for the various CPU models they produce.

DaemonGuard can be efficiently deployed in large-scale many-core systems with minimal impact on performance. The growing prevalence of both large-scale datacenters (e.g., for cloud computing) and high-performance computer clusters (e.g., for exascale computing) highlights the need for effective and efficient testing methodologies that can cope with such complex and mission-critical systems. The methodology proposed in this work could potentially provide very-low-cost testing to the microprocessors of the near future.

In order to assess the advocated new testing methodology, we fully implement and evaluate DaemonGuard in a full-system simulation framework based on Simics [27]/ GEMS [28]/GARNET [29], running a commodity operating system and executing the PARSEC benchmark suite [30] (a selection of multi-threaded applications) in a multi-core setup. We juxtapose the proposed selective testing procedure to two full-core testing approaches: (a) full-core testing triggered by total core utilization, and (b) full-core testing triggered by individual sub-core unit utilizations. Instead, DaemonGuard only performs unit-specific tests to the units that exceed a certain utilization threshold. The results of our experiments indicate savings in testing time of up to 30× when testing is performed in a selective manner. Moreover, the impact on system performance of the always-on Testing Manager process is shown to be negligible, thus corroborating our assertion that OS-assisted SBST is a viable option. The latter also demostrates the potential scalability of this approach to large-scale many-core systems. Moreover, the cache-aware selective testing capabilities of DaemonGuard are shown to substantially decrease the execution time of memory-intensive test programs, thus minimizing the testing overhead and its impact on overall system performance.

The rest of the paper is organized as follows: Section 2 discusses related work in the domain of online testing. Section 3 motivates the adoption of a daemon-based approach to SBST. Section 4 presents the proposed DaemonGuard framework and its underlying architecture, while Section 5 describes the enhanced, cache-aware testing mechanism. Section 6 provides the details of the employed evaluation framework and presents the experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Several non-concurrent on-line testing techniques have been proposed in the literature [11]. SBST or functional testing is a promising solution for periodic testing of traditional microprocessor architectures [21]. Recently, the advent of the multi-/many-core era has spurred a series of techniques targeting the testing of a system at the core and system levels. When applying such methodologies in microprocessors, the main source of overhead is *testing time*. One of the main objectives of these techniques is to reduce the testing time overhead.

Constantinides et al. [31] proposed an online testing methodology using an enhanced ISA with special instructions for fault detection and isolation. Structural tests are performed by applying test patterns using software routines. The test routines are executed periodically, after a number of executed instructions have committed, and checkpoints are used for recovery. The technique of Constantinides et al. is software-*assisted*, but it requires various hardware modifications. These intrusive modifications are needed, because the goal is to enable very detailed structural testing through existing scan-chain infrastructure. On the contrary, the proposed DaemonGuard approach targets *functional* testing, which is purely software-based, and only employs regular instructions that are part of the processor's ISA.

A hardware and software co-design methodology for functional testing is proposed in [32]. The testing methodology is based on the redundancy concept, whereby two cores execute the same program and capture corresponding footprints. The results of the executions are compared for fault detection. The choice of the test program is based on profiling that can be done offline or online. In [33], the authors propose a thread relocation methodology that uses dynamic profiling based on phase tracking and prediction.

Apostolakis et al. [16] propose a scheduling methodology for the test routines, aiming to reduce the test execution time by exploiting core-level parallelism. A Multi-Threaded (MT) SBST methodology is proposed in [18]. The authors propose an efficient MT version of functional unit test programs, in order to reduce the execution time of testing. All test routines are run simultaneously, based on the thread-level parallelism capabilities of the core under test. In all cases, testing is assumed to be initiated periodically (at regular time intervals) and is performed for the entire core. Note that the proposed DaemonGuard scheme is orthogonal to MT SBST, and, thus, it may be used to complement such methods.

Yanjing Li et al. [34] demonstrate the need for OS support in efficiently orchestrating online self-testing in future robust systems. Their contribution is the development of test-aware OS scheduling techniques for multicore systems. These techniques take into account the availability of each core when deciding the initiation of full-core testing, with the ultimate goal being to reduce the performance impact of the testing procedure. Test-aware OS scheduling is orthogonal to the work presented in this article and can, in fact, complement the DaemonGuard framework. In other words, while the current incarnation of DaemonGuard does not modify the OS scheduler, it is conceivable that a more holistic approach could allow the DaemonGuard's Testing Manager to coordinate with the OS scheduler.

A recent test-scheduling study for online error detection in multicore systems is discussed in [35]. The authors evaluate the performance of test programs applied on Intel's 48-core Single-chip Cloud Computer (SCC) architecture. Due to possible congestion within common hardware resources used by the various cores, the test time can be quite large with a significant impact on performance. As a result, the authors of [35] develop effective test scheduling algorithms to expedite the test process in such systems.

Table 1 presents a high-level comparison of the proposed selective SBST technique to other relevant online testing techniques (including full-core functional test-

TABLE 1: High-level comparison of relevant online testing techniques.

	Selective Funct. Testing	Full-Core Functional Testing				Struct. Testing
	(Proposed)	[16]	[18]	[32]	[34]	[31]
Test-Time Overhead	Low	High				Low
Overall Fault Coverage	Moderate to High					Very High
Fault Coverage per Testing Session	Unit-based	Core/System-based				
Detection Latency	Low to High (based on stress)	Depends on testing period				
System Avail. During Testing	Very High	High				None
Targeted Module	Functional unit	Core/System				
Test Triggering Method	Stress-based	Periodic				
Frequency of Test Triggering	High	Low (Depends on testing period)				
HW Support	Minimal	No	No	Yes	No	Yes
OS Modif.	No	No	No	Yes	Yes	No

ing). The main contribution of our work is the reduction of the test-time overhead, by avoiding unnecessary testing. As shown in Table 1, the "Overall Fault Coverage" of the selective and full-core testing approaches is the same. However, the limitation of selective testing is in terms of "Fault Coverage per Testing Session." The provided fault coverage of each testing session refers only to the particular functional unit under test (since only one functional unit is tested during each test session). Of course, this limitation only applies to each individual test session; once all units are tested over time, the overall fault coverage is identical to the one provided by full-core testing. Note that the under-utilized units are tested periodically. Another importand parameter affected by the proposed selective testing technique is the detection latency. Since test triggering is based on utilization (or, more generally, stress), the detection latency could vary from low to high: fault detection in highly-utilized units is much faster than in under-utilized units, due to the more frequent testing sessions. Most importantly, the impact of the proposed testing approach on overall system performance is minimized by utilizing a software-based framework (with minimal hardware support), which runs seamlessly and transparently within the OS of the multi-core system.

3 Daemon-Based Selective SBST

In order to enable *selective* testing – whereby the execution of unit-specific test routines is initiated on demand at run-time – it is essential to be able to dynamically track the utilization of the functional units within each CPU core. This motivates us to implement DaemonGuard, a light-weight, minimally-intrusive framework, which transparently orchestrates the procedure of selective testing. DaemonGuard is responsible to (a) monitor the system activity during normal operation, (b) initiate the execution of the appropriate test routines on the appropriate cores, and (c) collect the test results. Test routines that target individual units within a CPU core are loaded onto the operating system and executed on demand. In order to facilitate this process at minimum performance cost, we implement DaemonGuard as an always-on active OS process awakening a number of idle daemons, which execute the unit-specific test routines.

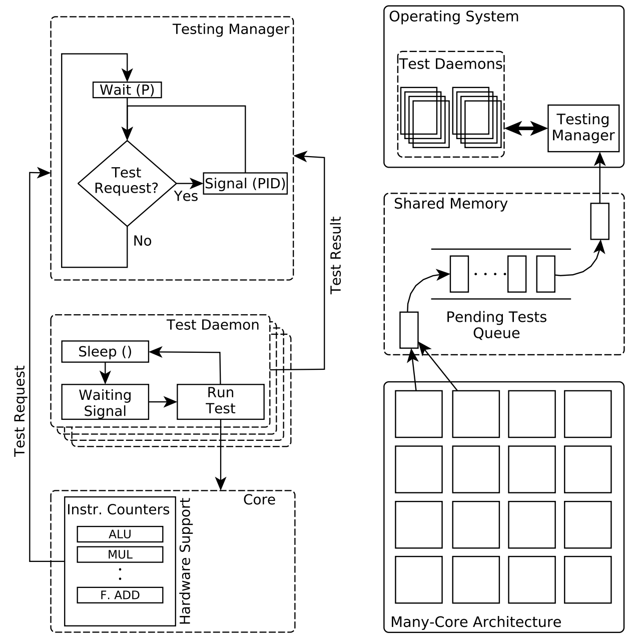
In multitasking computer operating systems, daemons are programs that run as background processes without any interaction at the user level. Daemons can be characterized as common processes; i.e., they have a Process ID (PID) and all operations pertaining to processes – such as the sleep function and various signals – can be applied. Daemon programs are loaded onto the system once, when the operating system starts up, and they run continuously during the normal operation of the system. In the most cases, daemons are idle processes waiting for an appropriate signal, or interrupt, in order to become active and perform their task(s). An example of such OS daemon is the *printer server* in unix-based operating systems. The daemon is loaded and executed during the startup phase of the operating system and runs continuously in the background (in idle mode) while waiting for a job to print.

In our work, the test programs are normal OS processes that run in the same way as normal applications. As a result, the isolation of test programs is facilitated by the operating system through context switching, whereby the state of a process is stored and restored on-demand, and the execution of both test programs and normal workloads is seamlessly time-multiplexed on the system. Hence, it is the OS that handles the scheduling and isolation of test programs, since the latter are normal OS processes.

A series of experiments are performed in Section 6 to investigate the impact of DaemonGuard on overall system performance. It will be demonstrated that the cost of employing an always-active Testing Manager process is very low. In fact, the results in Section 6 indicate that the overhead imposed by DaemonGuard is near-negligible, at around 0.23% (average over all benchmarks).

The incurred overhead is attributed to the always-on Testing Manager process, which periodically checks for pending test requests. A significant parameter within the Testing Manager is the period P over which pending test requests are monitored; i.e., period P is the time (in cycles) between each check of the Testing Manager for pending test requests. Period P is determined by the threshold T of committed instructions that a functional unit must exceed in order to initiate testing. This threshold is typically determined by the underlying technology, the criticality of the system, and the application of on-line self-testing. For example, for circuit failure prediction [13], [14], [36], periodic on-line self-testing can be performed less frequently – say once every 10 secs – than in the case of hard failure detection, which can occur as often as once per second, in order to allow for low-cost and low-latency recovery [34]. In our framework, a testing request is initiated by the core, based on workload utilization, when the threshold T is reached. Hence, the monitoring of pending test requests must occur frequently, so as to enable timely detection of test requests. The relationship between P and T is expressed as $P << \tilde{T}$, where $\tilde{T}$ is the estimated time in cycles required to commit T instructions.

The number of loaded test *daemons* does not impact the performance overhead, since the daemons are always *idle* during normal operation. The cost incurred by the test daemons is only in terms of memory overhead. Since daemons are common loaded processes of the OS, they have a portion of the main memory allocated to them. However, the memory footprint is determined by the *number of unique test routines* to be executed, not the number of daemons running. Since we consider 7 functional units, we need 7 test routines, which will determine the total memory footprint. If multiple daemons execute the same test routine, then all these daemons will execute the same routine instructions from the same physical address space. For example, in a 16-core system, we would need one ALU test daemon per core, i.e., a total of 16 test daemons executing the same ALU test routine. Thus, the memory footprint for the ALU test routine would only be incurred once, *not* 16 times. This is precisely the reason why this daemon-based approach is scalable with the number of CPU cores.



(a) The three main components of the DaemonGuard framework. (b) The flow diagram of DaemonGuard's operation.

Fig. 1: Architectural Overview of the DaemonGuard Framework. The three main components of DaemonGuard work in unison in order to facilitate real-time monitoring of the utilization of the various functional units within each core of the multi-core system. Once a unit exceeds a certain number of executed instructions, a test request is generated. Accordingly, the Testing Manager OS process invokes the appropriate OS-resident test daemon to initiate the execution of the unit-specific test routine.

4 The Basic DaemonGuard Framework

The proposed DaemonGuard framework considers utilization-based (hence on-demand/selective) testing and comprises three main components, as illustrated in Figure 1(a). The top two components (the Testing Manager OS Process and the Test Daemons) are implemented purely in software and reside within the OS. The bottom component in Figure 1(a) shows the minimal hardware support (the Instruction Counters for the various functional units within each CPU core) required to provide utilization information to the Testing Manager.

4.1 The *Testing Manager* OS process

The main part of the DaemonGuard framework is the Testing Manager process that runs at the OS-level and is responsible for the invokation of the various test daemons, based on the utilization information provided by the hardware instruction counters residing alongside each functional unit within the CPU cores. The main function of the Testing Manager is the periodic checking of pending test requests by any functional unit of any core within the system. Since our aim was to present a proof-of-concept implementation for the DaemonGuard framework, a polling-based approach was employed for simplicity. It should be noted that an interrupt-based implementation would be more efficient. Regardless, the results show that even the polling-based approach incurs near-zero overhead. Irrespective of the implementation details of the Testing Manager, the main goal of this work is to demonstrate the potential benefits of selective testing. The test requests are stored into a Pending Tests Queue structure, as depicted in Figure 1(b). This memory-mapped software structure is located in shared memory, where every core of the system has direct access. This means that the Testing Manager can run on any available core of the system and it can still have access to the queue.

The OS is responsible to schedule the Testing Manager process, just like all other processes running during normal system operation. During each checking period (the period P described in Section 3), the Testing Manager checks the queue, it dequeues all pending test requests, and it sends a wake-up signal to the corresponding test daemons. The signal, in this case, is an inter-process communication message and it is facilitated by the OS. Due to this OS-assisted functionality, the proposed testing methodology is minimally intrusive at the system level, and the test programs can run "simultaneously" with the normal applications, using OS context switching. In fact, depending on the running application, the OS may sometimes be able to completely "hide" the execution of the test daemons.

Upon completion of a test session, the corresponding test daemon sends the test result back to the Testing Manager. We note here that in our current implementation of DaemonGuard, the analysis of the test results (i.e., the comparison of the test results with the golden/expected results) is carried out by each test daemon individually, followed by a pass/fail signal from each test daemon to the Testing Manager. An alternative implementation could delegate the analysis of every test result to the Testing Manager. Recovery is beyond the scope of this work; however, it can be supported in the proposed framework by means of checkpointing and roll-back [31]. In general, recovery/reconfiguration/disabling actions could be initiated by the Testing Manager when a test failure is detected. The complete flow diagram of the DaemonGuard's operation is illustrated in Figure 1(b). The high-level pseudo-code of the Testing Manager is given in Algorithm 1. The checking period of the Testing Manager is decided a priori and remains constant during the normal operation of the system.

Algorithm 1 Testing Manager

Input: Period P, Pending Tests Queue (PTQ)

```
1: while $True$ do
2:     while PTQ $not$ Empty do
3:         SendSignal(PTQ.dequeue()) /* trigger test */
4:     end while
5:     while $Available\ Results$ do
6:         CollectResult($daemon$)
7:     end while
8:     Sleep($P$)
9: end while
```

4.2 Hardware support

The proposed selective testing methodology relies on the run-time gathering of information regarding the utilization of the various functional units within each core of the system. The term "utilization" refers to the number of instructions that have made use of the specific unit during normal operation. In order to collect this data, we assume the presence of a set of hardware instruction counters. There is one such counter for each functional unit within each core. For example, if an executed instruction uses the ALU, the corresponding counter will increase by one. The value of each counter is then used to determine when a functional unit must request a test. When the predefined threshold of T instructions is met, the affected unit places a test request in the Pending Tests Queue of Figure 1(b). The Testing Manager process will then invoke the appropriate daemon (e.g., the ALU test daemon) to execute the unit-specific test routine on the corresponding core. After the execution of the test routine finishes, the specific counter will receive a reset signal from the Testing Manager process to restart counting from zero.

4.3 The test daemons

In order to perform selective SBST, a number of test daemons (as described in Section 3) are loaded onto the OS. The number of loaded test daemons depends on both the number of CPU cores present in the system, and the number of functional units within each core. DaemonGuard requires one test daemon per functional unit per core. As previously mentioned at the end of Section 3, the memory footprint is *not* affected by the number of daemons, but by the number of *unique* unit-specific test routines. Daemons are kept in idle mode during normal operation; they wait for the appropriate invokation signal from the Testing Manager OS process, in order to wake up and execute their specified unit-specific test routine on the targeted functional unit of a specific core. The flow diagram of the operation of a test daemon is shown in the middle box of Figure 1(a). Upon completion of the testing process, the outcome (i.e., if a fault has been detected or not) is reported to the Testing Manager process.

4.4 Proposed selective testing based on functional-unit utilization

The current trend in SBST schemes is to periodically test either the entire core, or the entire system. This may lead to considerable over-testing, especially in the case of multi-/many-core systems where certain parts of the system may be under-utilized at certain points in time. Hence, *utilization-based testing* may be more appropriate and can lead to a considerable reduction in the overall testing time, since unnecessary testing can be avoided.

We have derived sub-core unit utilization statistics through a series of full-system simulations over the PARSEC benchmark applications [30]. For each benchmark, we periodically collect information pertaining to the number of executed instructions. We classify the instructions based on their type and the corresponding functional unit that is responsible for their execution. The detailed results of the profiling exploration are provided as supplemental material in Appendix A, Section A.1. The profiling results show that the distribution of the instructions over the various units *within each core* is non-uniform, mainly due to the nature of the running application. This motivates us to explore the potential of *selective testing* at the sub-core granularity by applying appropriate test routines to only the strained units and, thus, reducing the overall testing time overhead.

Hence, we propose a selective testing methodology based on sub-core level utilization, i.e., on the utilization of individual functional units within a CPU core. We refer to this testing policy as *Selective Core-Unit Testing based on Core-Unit Utilization (ST-UU)*. The system takes into consideration all of the core's functional units and performs individual unit tests. The execution of a particular test routine targeting a specific functional unit within a core is triggered after a pre-defined number of T instructions (the threshold) have used the specific unit. Each time the number of executed instructions on a unit meets the threshold T, a test request is placed into the Pending Tests Queue. The Testing Manager process

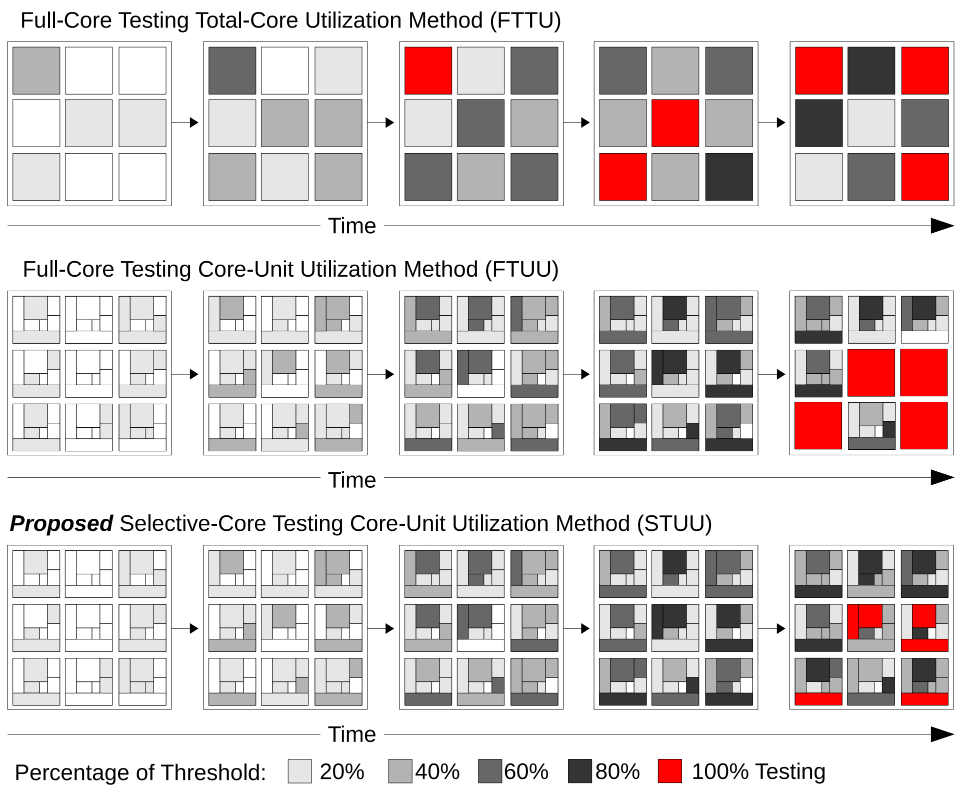

Full-Core Testing Total-Core Utilization Method (FTTU)
Time
Full-Core Testing Core-Unit Utilization Method (FTUU)
Time
Proposed Selective-Core Testing Core-Unit Utilization Method (STUU)
Time
Percentage of Threshold: 20% 40% 60% 80% 100% Testing

Fig. 2: Abstract illustration of the differences between the three examined testing methodologies, FT-TU, FT-UU, and ST-UU. The stress on each core or unit (i.e., number of instructions executed) is represented by the color intensity, as a percentage of the threshold required to trigger testing. Red squares/rectangles are cores/units under test.

then invokes the corresponding test daemon to initiate execution of the unit-specific test routine. Using this approach, unnecessary testing of the remaining (under-utilized) functional units is not performed.

We explore two more testing methodologies that perform *full-core* testing, based on utilization at the core- and sub-core levels. These two scenarios will be used to demonstrate the impact of selective testing on over-testing savings, as compared to non-selective, full-core testing schemes. The first method – *Full-Core Testing based on Total-Core Utilization (FT-TU)* – tracks the total number of executed instructions within each core, without considering the functional units used by each instruction. When the number of executed instructions meets the testing threshold, the OS invokes a test daemon that tests the entire core. Note that this scenario is not directly comparable to the philosophy of unit-based utilization. It is included here to serve as an indicative measure of the potential savings that may be reaped for cases where utilization statistics can only be obtained at the core level.

A directly comparable testing methodology is the second full-core testing methodology used in our exploration, the *Full-Core Testing based on Core-Unit Utilization (FT-UU)*. Here, the system tracks the number of executed instructions on each functional unit within each core, similar to the proposed DaemonGuard methodology. When the functional unit with the highest utilization within a core exceeds the testing threshold (i.e., the unit executes more instructions than the threshold T), the OS invokes a test daemon that tests the *entire* core. Note that after the execution of the full-core testing process, all unit counters will restart counting from zero.

Figure 2 illustrates abstractly the differences between the three examined testing methodologies. Let us assume that we have a 9-core CMP. In the case of the FT-TU testing scheme (top part of Figure 2), the CMP only observes the number of instructions executed in each core, so the minimum granularity of observation is a single core (represented by each small square in the 9-core CMP). The stress on each core (i.e., number of instructions executed) is represented by the square's color intensity, as a percentage of the threshold required to trigger testing. For example, a light grey core indicates low stress, and as the threshold utilization target is approached (i.e., closer to 100%), the color becomes darker. Once the threshold target is hit, the core transitions to testing mode (indicated by the red color). As time progresses, more squares (cores) become darker in color (i.e., total-core stress increases), and various cores transition into testing mode. The second testing methodology, FT-UU (middle part of Figure 2), observes core stress at a sub-core granularity, as indicated by the smaller rectangles, which represent individual functional units within each core. However, testing is still performed on the entire core once the threshold of a functional unit is exceeded. Finally, the proposed ST-UU methodology (bottom part of Figure 2) conducts selective testing only on the functional units that exceed their stress threshold. As evident by the smaller red rectangles in the bottom row of Figure 2, the proposed methodology provides fine-grained testing and, thus, it avoids overtesting. Even though the same workload is executed in all three cases, the *test-trigger time* and the *test target* are different, based on the methodology employed. The proposed ST-UU approach provides finer-granularity observations and finer-granularity testing sessions.

5 Cache-Aware Selective Testing

The DaemonGuard framework is designed to initiate testing based on the individual unit utilizations. While this philosophy is extremely beneficial in terms of providing fine-grained testing (which is faster and avoids overtesting), the framework seems to ignore a fundamental aspect of the system: the memory hierarchy. The memory hierarchy comes into play when the test programs become large enough that they start to stress the memory sub-system. Some of the test programs used in this work are small enough to fit entirely within the L1 cache of any individual core in the CMP. Hence, other than the unavoidable compulsory misses, those test programs will not experience any other cache misses throughout their execution. However, there are other test programs with larger working sets, which cannot fit within the cache hierarchy. These test programs are *memory-intensive*; they may experience large numbers of cache misses, and, consequently, suffer from very expensive (in terms of wasted stall cycles) off-chip memory accesses.

Moreover, the trend towards larger-scale and more complex (e.g., heterogeneous) CMPs implies an accompanying increase in the size of some of the test programs used in SBST schemes. Some researchers have already reported fairly large (in terms of memory footprint) test programs [9], [35], and it is not unreasonable to expect further increases in the memory footprints of various test programs in the near and distant future. It is, therefore, imperative to investigate the impact of the cache hierarchy on the testing time of memory-intensive test programs, in an effort to devise ways to decrease the execution time of such large test programs.

Motivated precisely by this need to further contain the testing time, we augment the DaemonGuard framework with the capability to be *cache-aware*. Selective testing can now be initiated not only based on unit utilization, but also on the recent history of test sessions by other similar units in other cores. Our simulation results indicate that test-related data (both test patterns in the data segment and instructions in the text segment) remains cached in the hierarchy for a substantial period of time after the test session concludes its execution. This period can be viewed as a *window of opportunity* for other cores to exploit. Remember, that the data required by a test daemon is shared between all the loaded daemons targeting the same functional unit in the various cores. Hence,

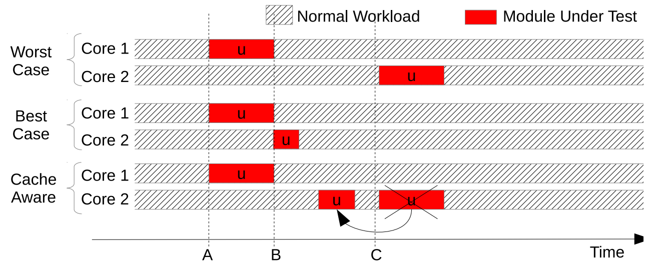


Fig. 3: An abstract example of how cache awareness can benefit the testing process. The goal is to take advantage of the so called *window of opportunity*, which is a limited period of time after the testing session of a particular type of functional unit in one core in the CMP. During this time window (designated by the time between points B and C on the timeline in this figure), other cores wishing to test the same type of functional unit could take advantage of cache-resident blocks from the other core's testing session.

if the *Testing Manager* could observe the recent testing history on all cores, the DaemonGuard mechanism would be able to exploit the memory hierarchy of the CPU to re-use cache-resident blocks, thereby limiting the need for time-consuming off-chip memory accesses. As a result, the overall execution time of memory-intensive test programs would be significantly reduced. Figure 3 presents an abstract example of how cache awareness can benefit the testing process. The top portion of Figure 3 illustrates the *worst-case* (in terms of testing overhead) scenario: Core 1 conducts a test session for a particular functional unit; Core 2 conducts the same test for the same type of functional unit (situated in Core 2), but Core 2's test commences at a time beyond the window of opportunity. In this figure, the window of opportunity is denoted as the time between points B and C on the timeline. This worst-case scenario occurs when all the test instructions and data have been evicted from the cache, i.e., it corresponds to a cold run yielding the longest possible test execution time. The middle portion of Figure 3 illustrates the *best-case* scenario: Core 2's test commences immediately after the end of Core 1's test (or, it could also overlap with Core 1's test); thus, Core 2's test benefits from cache-resident blocks and the total time needed to complete the test session is minimized. The worst- and best-case scenarios constitute the performance *bounds* pertaining to the impact of the cache on the testing time. Finally, the bottom part of Figure 3 depicts the solution proposed in this work, which triggers earlier testing in Core 2 (i.e., the test is moved within the window of opportunity), in order to take advantage of cache-resident blocks from Core 1's testing session. Obviously, the proposed methodology would fall somewhere between the worst- and best-case performance bounds, as only those tests falling outside but close to the window of opportunity will be triggered earlier, in order to benefit from cache-resident blocks.

Based on this premise, we propose a new cache-aware selective methodology, where we consider *both* the stress on each individual unit *and* the recent history of test sessions targeting the same unit in other cores.

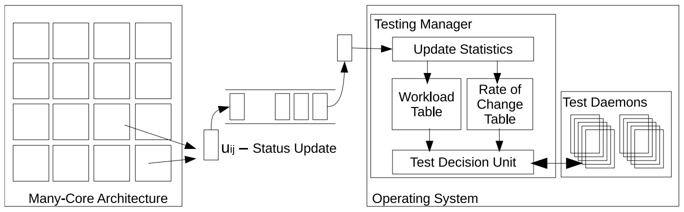


Fig. 4: A high-level overview of the enhanced – *cache-aware* – DaemonGuard framework. The new structures employed are the *Status Update Queue* (situated in shared main memory) and two statistics tables, the *Workload Table* and the *Rate-of-Change Table* (both residing in main memory, within the address space of the Testing Manager itself).

In the baseline selective method (i.e., the non-cache-aware one), each core is responsible to request a test when the number of executed instructions on a specific functional unit exceed a threshold T. In the cache-aware version, the cores are no longer responsible to request a test. Instead, every T/d executed instructions on a particular unit of a particular core – where d is a constant integer and determines the granularity of threshold subdivisions – the core updates the corresponding unit's status by sending a signal to the Testing Manager. This is achieved using a Status Update Queue (SUQ) situated in main memory and shared by all cores. In other words, the signal sent by each core for each of its functional units simply indicates that T/d instructions have been executed on the particular unit since the previous status update. The signal also includes a timestamp, the purpose of which will be explained shortly. Figure 4 presents a high-level overview of the enhanced – cache-aware – DaemonGuard framework. Thus, each core c_i in the CMP inserts individual status updates for each of its functional units u_j in the Status Update Queue. These update signals are indicated by u_{ij} in Figure 4. The coordination of the testing procedures on the entire CMP is now performed by the Testing Manager module, which is modified to support and orchestrate the cache-aware selective testing scheme. Algorithm 2 presents the high-level pseudo-code of the cache-aware Testing Manager OS process.

The Testing Manager reads the SUQ and updates two statistics tables: (1) the *Workload Table*, which holds the percentage of executed instructions with respect to the threshold T for each unit within the system, and (2) the *Rate-of-Change Table*, which holds statistical information indicating the rate of executed instructions per unit for a particular time period. Note that both tables reside in main memory, within the address space of the Testing Manager itself. The *Workload Table* is updated by the Testing Manager each time a core c_i submits an updated status for unit u_j, using the following equation:

$$K_{ij} = K_{ij} + 1/d \tag{1}$$

The term K_{ij} indicates the stress level of each unit u_{ij}, and $1/d$ is the percentage of executed instructions by functional unit u_{ij} – with respect to the threshold T – between two consecutive status updates. Recall that d is an integer constant. Assuming K_{ij} is initialized to 0, then its value ranges from 0 to 1, with 1 indicating that the threshold T has been reached and the unit u_{ij} must be tested.

Additionally, the rate-of-change R_{ij} of the stress on a functional unit (i.e., a measure of how frequently the unit is currently being used) is calculated using the following equation:

$$R^t_{ij} = \frac{T/d}{C^t_{ij} - C^{t-1}_{ij}} \tag{2}$$

The parameter t in the above equation is a discrete counter indicating the t^{th} submitted status update in the SUQ, T/d is the number of executed instructions between two consecutive status updates for functional unit u_{ij}, and C^t_{ij} is the timestamp at time instant t (as indicated by the previously explained status update signals sent by each core). The rate-of-change R_{ij} is calculated as the ratio of the number of executed instructions T/d over the number of elapsed cycles within the interval $t-1$ to t.

Algorithm 2 Testing Manager (Cache-Aware)

```
Input: Period $P$, Status Update Queue ($SUQ$)
 1: while $True$ do
 2:     while SUQ $not$ Empty do
 3:         $u_{ij}$ ←SUQ.dequeue()
 4:         Update Workload Table($u_{ij}$) /* $K_{ij}$ (Eq. 1) */
 5:         Update Rate of Change Table($u_{ij}$) /* $R_{ij}$ (Eq. 2) */
 6:         $EC_{ij} = (1 - K_{ij}) \times T/R_{ij}$ /* (Eq. 3) */
 7:         if ($u_{ij}$ $has\ exceeded$ $T$) or ($EC_{ij} < L$) then
 8:             SendSignal(unit($u_{ij}$)) /* trigger test */
 9:         end if
10:     end while
11:     while $Available\ Results$ do
12:         CollectResult($daemon$)
13:     end while
14:     Sleep($P$)
15: end while
```

The Testing Manager (running as a process within the OS of the multi-core system) uses three pieces of information: (a) data from the two above-mentioned statistics tables, (b) the type(s) of units that have just exceeded their threshold T, and (c) all units of the same type(s) that have recently completed their testing session(s). By combining this valuable history information, DaemonGuard becomes cache-aware and triggers the appropriate testing daemon(s). A test daemon for some unit u_{ij} is triggered under any of the following two conditions: (i) threshold T has been reached at unit u_{ij} (this information is contained in the Workload Table), or (ii) the estimated expected number of clock cycles remaining until the next testing session of unit u_{ij} – denoted by EC_{ij} – is less than L. The term L is the number of cycles required by a functional unit in the system to execute 10% of the instructions (a value chosen empirically here, without loss of generality) of its testing threshold T. Recall that the testing threshold

TABLE 2: Simulated system parameters.

Processors	16 UltraSparc III+ cores
L1 Caches	32 KB I& 32 KB D, 3-cycle latency
L2 Caches	16 MB, 10-cycle latency
Main Memory	4 GB, 200-cycle latency
Network	4x4 2D Mesh
OS	Solaris 10

TABLE 3: The execution times and memory footprints of the employed unit-specific test programs [18].

Functional Unit	Perfect Memory (K Cycles)	w/Memory Hierarchy (K Cycles)	Code Size (KB)	# of Instr.	Pattern Size (KB)
Int ALU	32.8	63.6	1.5	280	1
Int Mult	8.8	23.2	0.6	110	1
Int Div	51	62.8	0.5	800	2
Branch	32.8	43.6	1.45	270	1
Fload Add	1,302	20,000	3.18	30	1696
Fload Mult	534	8,600	2.96	30	464
Fload Div	774	12,569	2.7	30	672
Full-Core	2,737	41,000	12.89	1550	2837

T is a particular number of executed instructions. EC_{ij} is an estimate of the expected remaining time before the threshold of unit u_{ij} is reached, and it is calculated by:

$$EC_{ij} = (1 - K_{ij}) \times T / R_{ij} \tag{3}$$

Hence, upon completion of a test program on a particular functional unit on a particular core, the Testing Manager examines the status of the same functional units in the other cores to see if the estimated number of cycles to reach the threshold is less than L. Those units that satisfy this condition are immediately tested (i.e., early testing is triggered for those units), so as to take advantage of the window of opportunity in the cache. Note that since the rate-of-change R_{ij} of the stress is also considered in the calculation of the estimate EC_{ij}, units with a low utilization rate are left un-tested until they are much closer to their threshold limit (thereby avoiding unnecessarily early testing for units that are not utilized often).

In the cases where no similar functional units in other cores are close to their testing threshold – i.e., the aforementioned inequality is not satisfied – the Testing Manager simply resorts to the baseline scenario, whereby units are tested based only on whether threshold T has been exceeded.

6 Experimental Framework and Results

6.1 Evaluation framework

For the evaluation of the proposed testing methodologies, we use a full-system, execution-driven simulation framework based on the Wisconsin GEMS toolset [28], in conjunction with Wind River's Simics [27]. We simulate a 16-core CMP, as presented in Table 2. Each core in the system is an in-order-execution UltraSPARC III+ core.

Our approach of selective testing requires one test daemon for each functional unit of each CPU core. We first use the test routines from [18], which is the most recent work that developed testing routines at the functional-unit level. In order to use these routines in our framework, we implemented them as daemons and integrated them within the OS, as descriped in previews sections. These test routines target the OpenSPARC microprocessor [37], which uses the SPARC v9 ISA. The UltraSPARC III+ core in our evaluation framework also uses the SPARC v9 ISA, which means that the test routines of [18] are applicable to our system. The execution times, number of instructions, and memory footprints of these routines are given in Table 3. In terms of execution time, two sets of times are reported: one for a microprocessor with a perfect memory system (i.e., without considering any memory latencies), and one for a realistic system employing the memory hierarchy described in Table 2. The two sets of execution times are shown to identify the test routines that experience elevated cache misses in the realistic system (i.e., the system used in our simulations). In terms of the memory footprint, Table 3 shows the memory capacity consumed by each test program, which includes both the source code of the test program (i.e., instructions in text segment) and the size of the test patterns loaded from memory (i.e., the data segment). Obviously, the test programs targeting the integer units (the first four functional units in Table 3) are extremely small and can easily fit within the L1 cache of a core. Thus, these test programs are not memory-intensive.

As previously mentioned in Section 5, there are larger test programs already employed in multi-core SBST schemes [9], [35]. These programs are considered larger, in the sense that they rely on large numbers of test patterns loaded from off-chip memory. Furthermore, given the continuous increase in both the size and complexity of multi-/many-core microprocessors, it is reasonable to anticipate an equivalent increase in the amount of required test patterns. To account for these trends, we develop a second set of test programs, as shown in Table 4. These test programs have the same structure as those in Table 3, with the only difference being in the increased amount of loaded test patterns. We primarily change the size of the data segment of the first four test routines of Table 3, as these are the ones invoked more frequently under the selective testing scenario proposed in this work. The last three test routines (the ones for the floating-point units) already have a large data segment (see Column 6 of Table 3), so we do not modify these. The second set of these test programs, shown in Table 4, will be used to assess the efficacy and efficiency of the cache-aware DaemonGuard framework.

Full-system, execution-driven simulations are performed using the PARSEC benchmark suite [30]. PARSEC is a benchmark suite of multi-threaded workloads that focus on emerging parallel workloads. We use PARSEC benchmarks for our extensive profiling of the distribution of instructions over the units and the cores of the system. To evaluate the proposed testing methodologies, we use eleven of the benchmarks. Details about

TABLE 4: The execution times and memory footprints of the, *memory-intensive* test programs used to assess the enhanced, *cache-aware* DaemonGuard framework.

Functional Unit	Perfect Memory System (K Cycles)	w/Memory Hierarchy (K Cycles)	Pattern Size (KB)
Int ALU	32.8	921	50.5
Int Mult	8.8	233	11.0
Int Div	33.9	801	44.2
Branch	32.8	909	50.5
Fload Add	1,302	20,000	1696
Fload Mult	534	8,600	464
Fload Div	774	12,569	672
Full-Core	2,718	44,000	2990

TABLE 5: Details of the PARSEC benchmark applications [30] used in our evaluation framework.

	Benchmark	Input Set	Executed Instr. (Billions)	Time M Cycles
1	Blackscholes	medium	0.76	552
2	Bodytrack	small	1.08	776
3	Canneal	medium	1.15	1,400
4	Dedup	small	2.84	1,700
5	Ferret	small	2.03	958
6	Fluidanimate	small	1.36	553
7	Freqmine	small	2.24	2,529
8	Raytrace	medium	1.36	193
9	Swaptions	small	2.57	890
10	Vips	small	4.04	2,180
11	x264	small	0.74	660

the input sizes, number of executed instructions, and execution times are given in Table 5. All the benchmarks are configured with 16 threads, since the multi-core setup in our evaluation framework consists of 16 cores.

6.2 Impact of the DaemonGuard Framework

To investigate the impact of DaemonGuard on overall system performance, we perform a series of experiments explicitly showing the performance cost of this OS-resident testing framework. Through this investigation, we aim to confirm the low cost of leveraging an always-active Testing Manager process invoking idle daemons on demand. Table 6 presents the results of this exploration considering the PARSEC benchmark applications. In order to speed up the simulation time, the memory system is not considered here, because no test routine is actually executed (which would require a memory access), i.e., the memory system does not affect the experiment under consideration. For the purposes of our investigation here, we run two simulations per benchmark: one of a "clean" run of the benchmark without the proposed framework, and one with the full-fledged DaemonGuard framework running on the system. A test daemon is loaded for the testing routine of each functional unit within each CPU core. Since we consider 7 functional units per core of the 16-core system, 112 test daemons are loaded in total. The testing routines are initiated, but not actually "executed," since we are only interested (at this point) in the performance overhead imposed by the proposed monitoring and test initiation framework, and not the actual testing time overhead. As can been seen in Table 6, the overhead imposed by DaemonGuard over all benchmarks is near-negligible, at around 0.23%.

TABLE 6: Impact of the DaemonGuard Framework on System Performance, $P = 750K$ cycles.

	Benchmark	"Clean" Run K Cycles	with DaemonGuard K Cycles	with DaemonGuard Overhead
1	Blackscholes	46,187	46,210	0.050%
2	Bodytrack	296,423	296,440	0.006%
3	Canneal	64,921	64,962	0.063%
4	Dedup	201,362	202,488	0.559%
5	Ferret	435,702	436,130	0.10%
6	Fluidanimate	165,291	165,483	0.116%
7	Freqmine	2,399,959	2,400,465	0.021%
8	Raytrace	85,453	85,733	0.328%
9	Swaptions	57,403	58,094	0.695%
10	Vips	357,257	359,263	0.561%
11	x264	240,723	240,879	0.064%
			Average	**0.232%**

For the experiments of Table 6, the period P is set to 750K, so the Testing Manager checks for pending test requests every 10 ms. The overhead, in terms of instructions, each time the Testing Manager checks for test requests is only 83 instructions. As the overall DaemonGuard performance overhead is inversely proportional to P, larger values of P will reduce the overhead even further. A smaller than 10 ms P is not considered viable, as this would imply an even smaller T, which, in turn, would activate SBST more frequently than needed. It is worth mentioning that the low overhead of the Testing Manager process is partially due to the optimal scheduling that the operating system performs; the Testing Manager process is not bound to any specific core within the system – it can run on any available core.

6.3 Evaluation results of the baseline (non cache-aware) DaemonGuard mechanism

We simulate all aforementioned benchmarks according to the testing methodologies described in Section 4.4, using the proposed DaemonGuard framework and the test routines of Table 3. Simulations are performed in order to evaluate the benefit of the proposed selective testing policy, ST-UU, as compared to the two full-core testing policies, FT-TU and FT-UU.

One key issue is the selection of an appropriate *testing threshold* T, i.e., the number of executed instructions that trigger the initiation of a test session. As already mentioned, T will be defined based on technology parameters related to aging and wear-out effects, the criticality level of the system, as well as the considered application of on-line self-testing (e.g., failure prediction vs. failure detection). First, we study how different values of T impact the proposed methodology. Specifically, we evaluate the testing-time overhead when modifying the testing threshold. Figure 5 shows the results when applying different threshold values. In particular, we performed simulations using threshold values T of 5M and 10M instructions for all the considered benchmarks. The three

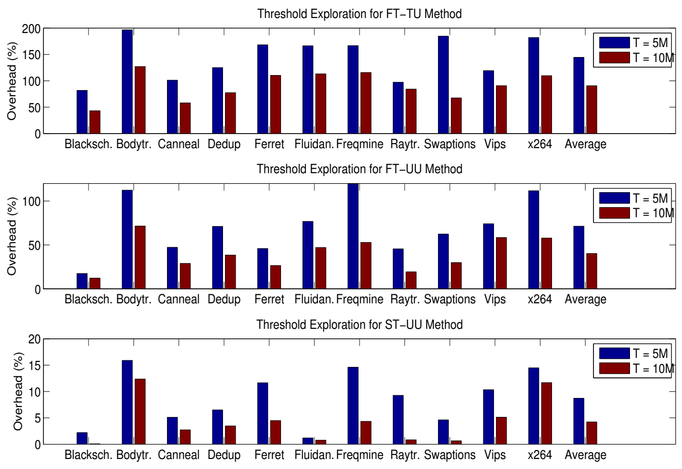


Fig. 5: Exploring the impact of the *testing threshold* T (the number of executed instructions required to trigger a testing session) on the execution time of the various benchmark applications. The three graphs correspond to the three testing methodologies under evaluation: FT-TU, FT-UU, and the proposed ST-UU (bottom graph).

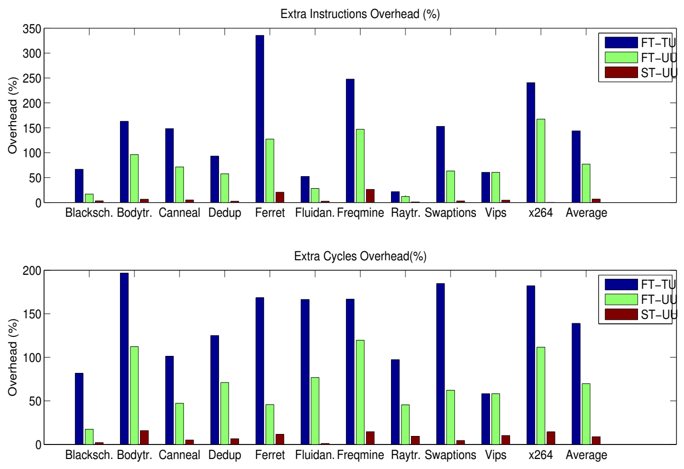


Fig. 6: The testing overhead imposed by the three testing methodologies across all benchmarks. The top graph shows the testing overhead in terms of extra executed instructions, while the bottom graph shows the overhead in terms of extra cycles needed. The testing routines are executed together with the running applications, whenever a testing session is triggered.

figures correspond to the three testing methodologies under evaluation. The x-axis of each graph shows the 11 benchmarks and each pair of bars corresponds to the two investigated threshold values. The last group of bars on the x-axis refers to the average results for all the benchmarks. The y-axis shows the extra overhead – in terms of cycles – incurred by each method due to testing. As it can be seen, the overhead is drastically reduced as the threshold increases, since the number of testing sessions also decreases. On average, doubling the threshold yields a reduction in testing overhead of about 40% in all three testing methodologies. This result highlights the significance of the testing threshold: a very low threshold will result in excessive testing and unnecessary stressing of the functional units by the test routines themselves. On the other hand, setting the testing threshold too high may result in late detection of faults.

For the remainder of our experiments, we use $T = 5M$ instructions. In practice, T may be larger; however, we limit its value here, in order to be able to invoke the various testing policies an adequate number of times for comparison purposes, while maintaining reasonable simulation times in our full-system, cycle-accurate framework.

Since the main objective of this work is to reduce the testing time, we investigate the total execution time (application time plus testing time), in terms of cycles, and the total number of executed instructions in the presence of each testing methodology. Figure 6 presents the results over all the considered benchmarks. The top graph of Figure 6 presents the testing overhead in terms of total committed instructions under each testing policy, while the bottom graph shows the testing overhead in terms of extra clock cycles. Each triplet of bars represents the three testing methodologies and shows the testing overhead normalized to a system with DaemonGuard

TABLE 7: Testing overhead results. The testing overhead is defined as the ratio of the test execution time (i.e., the time expended on testing sessions) over the total execution time of the system (i.e., until the benchmark application completes its execution). The results are averaged over the 16 cores of the system.

	Benchmark	Testing Overhead (%)		
		Best-Case (Ideal)	Non-Cache Aware	Cache-Aware
1	Blackscholes	10%	21%	15%
2	Bodytrack	8%	23%	12%
3	Canneal	7%	22%	9%
4	Dedup	3%	19%	3%
5	Ferret	6%	16%	9%
6	Fluidanimate	15%	33%	20%
7	Freqmine	3%	15%	3%
8	Raytrace	11%	36%	18%
9	Swaptions	8%	32%	12%
10	Vips	11%	22%	13%
11	x264	2%	9%	3%
	Average	8%	24%	12%

loaded but with no tests performed. The last triplet in both graphs shows the average results across all benchmarks. Clearly, the proposed selective testing approach (the right-most bar – ST-UU – in each triplet) yields a significant reduction in the testing cost, both in terms of extra executed instructions and extra cycles needed. On average, the testing overhead is less than 8% when applying the proposed selective testing approach (ST-UU), while full-core testing based on unit-utilization (FT-UU) imposes a 70% overhead.

We also compare the obtained testing overhead incurred – in terms of extra cycles needed – with the *expected* overhead. The results can be found in the supplemental material (Appendix A), in Section A.2.1.

6.4 Evaluation results of the *cache-aware* DaemonGuard mechanism

For the evaluation of the cache-aware DaemonGuard framework, we employ the memory-intensive set of test programs, as shown in Table 4 and described in Section 6.1. The main goal of cache-aware selective testing is to further decrease the testing time by exploiting the cache hierarchy to minimize expensive off-chip memory accesses. In order to assess the efficacy of the proposed cache-aware DaemonGuard mechanism described in Section 5, we investigate three different testing mechanisms: (1) the *best-case* scenario depicted in Figure 3 (ideal execution), (2) the non-cache-aware DaemonGuard framework of Section 6.3, and (3) the proposed enhanced cache-aware DaemonGuard. The first evaluated mechanism (best-case test scenario) corresponds to the situation where most test instructions and data are cache-resident, i.e., the previous testing session has just finished and its working set is still in the cache. This scenario is the optimal in terms of test execution time – since it minimizes the number of off-chip memory accesses – and it can be viewed as the ideal reference point for the evaluation of cache-aware selective testing. The second evaluated test mechanism (non-cache-aware DaemonGuard) initiates testing based solely on the thresholds of the various functional units, i.e., it is oblivious to the recent testing history of other similar functional units in other cores. Finally, the third mechanism (the proposed cache-aware technique) corresponds to the methodology described in Section 5, whereby test sessions can be initiated earlier than usual to ensure that they are executed within the window of opportunity provided by the cache hierarchy.

The evaluation results of cache-aware selective testing are analyzed using two important measures of merit: the total execution time of the test programs, and the number of Last-Level Cache (LLC) misses imposed by each of the three investigated simulation scenarios.

Table 7 presents the testing overhead results. The testing overhead is defined as the ratio of the test execution time (i.e., the time expended on testing sessions) over the total execution time of the system (i.e., until the benchmark application completes its execution). The results are averaged over the 16 cores of the system. As can be seen from the results, the proposed cache-aware selective testing methodology reduces the testing overhead incurred by memory-intensive test programs, as compared to the non-cache-aware DaemonGuard of Section 6.3. On average, the cache-aware DaemonGuard reduces the testing overhead from 24% to 12%, as compared to the non-cache-aware technique. This decrease corresponds to a quite significant reduction of 50% in testing overhead. The ideal (best-case) results are also shown as an indication of the theoretical maximum achievable improvement.

Details pertaining to the number of test-program-related LLC misses incurred by the three evaluated testing mechanisms can be found in the supplemental material (Appendix A), in Section A.2.2.

7 Conclusion

This paper introduces the notion of *selective* SBST as a means to drastically reduce the testing time overhead in multi-/many-core microprocessors. The proposed testing methodology views system activity at the sub-core granularity and initiates targeted testing of only the over-utilized (and, thus, strained) functional units. Under-utilized units are only sporadically tested.

To facilitate this testing regime, we introduce a framework called DaemonGuard, which enables real-time observation of individual sub-core modules and performs on-demand *selective testing* of individual functional units. This discriminatory testing approach offers drastic savings in testing time, since the testing phase only executes test routines relevant to the specific functional unit under test. DaemonGuard employs a transparent, minimally-intrusive, and lightweight operating system process that monitors the utilization of individual data-path components at run-time. Whenever a unit requires testing, DaemonGuard invokes OS-residing unit-specific test daemons to execute appropriate test routines.

Additionally, the DaemonGuard mechanism is augmented with the capability to exploit the memory hierarchy of the system in order to expedite the testing process. Large, memory-intensive test programs tend to stress the memory sub-system of the CMP. We demonstrate that cache-aware selective testing can significantly reduce the execution of memory-intensive test programs, by exploiting cache-resident blocks and minimizing the number of expensive off-chip memory accesses. Thus, the cache-aware DaemonGuard scheme initiates test sessions based not only on unit utilization, but also on the recent history of test sessions by other similar units in other cores.

Selective testing is compared against two full-core SBST approaches to evaluate the testing time overhead incurred on the system. Our results indicate substantial reductions in testing overhead of up to 30×. Moreover, the cache-aware testing scheme is shown to be very effective in exploiting the memory hierarchy to minimize the testing time of memory-intensive test programs. Most importantly, the impact of the DaemonGuard framework on system performance is shown to be negligible, thus corroborating our claim that OS-assisted selective SBST is a feasible option in modern microprocessors.

References

[1] S. Borkar, "Thousand core chips: A technology perspective," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 746–749.

[2] ——, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[3] A. W. Strong, E. Y. Wu, R.-P. Vollertsen, J. Sune, G. L. Rosa, and T. D. Sullivan, *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. John Wiley & Sons, 2006.

[4] J. Van Horn, "Towards achieving relentless reliability gains in a server marketplace of teraflops, laptops, kilowatts, and "cost, cost, cost"...: making peace between a black art and the bottom line," in *Test Conf.,ITC 2005. IEEE International*, Nov 2005, pp. 8–678.

[5] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya, "Integration challenges and tradeoffs for tera-scale architectures." *Intel Technology Journal*, vol. 11, no. 3, 2007.

[6] A. Baba and S. Mitra, "Testing for transistor aging," in *VLSI Test Symposium, 2009. VTS '09. 27th IEEE*, May 2009, pp. 215–220.

[7] D. Gizopoulos, M. Psarakis, S. Adve, P. Ramachandran, S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[8] S. Mitra and E. McCluskey, "Which concurrent error detection scheme to choose ?" in *Test Conference (ITC), 2010 IEEE International*, 2000, pp. 985–994.

[9] Y. Li, S. Makar, and S. Mitra, "Casp: Concurrent autonomous chip self-test using stored test patterns," in *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 885–890.

[10] H. Inoue, Y. Li, and S. Mitra, "Vast: Virtualization-assisted concurrent autonomous self-test," in *Test Conference (ITC), 2008 IEEE International*, Oct 2008, pp. 1–10.

[11] M. Nicolaidis and Y. Zorian, "On-line testing for vlsi - a compendium of approaches," *J. Electron. Test.*, vol. 12, no. 1-2, pp. 7–20, feb 1998.

[12] M. Agarwal, V. Balakrishnan, A. Bhuyan, K. Kim, B. Paul, W. Wang, B. Yang, Y. Cao, and S. Mitra, "Optimized circuit failure prediction for aging: Practicality and promise," in *Test Conference, 2008. ITC 2008. IEEE International*, Oct 2008, pp. 1–10.

[13] T. W. Chen, K. Kim, Y. M. Kim, and S. Mitra, "Gate-oxide early life failure prediction," in *VLSI Test Symposium, 2008. VTS 2008. 26th IEEE*, April 2008, pp. 111–118.

[14] T. W. Chen, Y. M. Kim, K. Kim, Y. Kameda, M. Mizuno, and S. Mitra, "Experimental study of gate oxide early-life failures," in *Reliability Physics Symposium, 2009 IEEE International*, April 2009, pp. 650–658.

[15] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-in Test for VLSI: Pseudorandom Techniques*. New York, NY, USA: Wiley-Interscience, 1987.

[16] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *Computers, IEEE Trans. on*, vol. 58, no. 12, pp. 1682 –1694, dec. 2009.

[17] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache resident functional microprocessor testing: Avoiding high speed io issues," in *Test Conference, 2006. ITC. IEEE International*, oct. 2006, pp. 1 –7.

[18] N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, and A. Gonzalez, "Mt-sbst: Self-test optimization in multithreaded multicore architectures," in *ITC'10*, nov. 2010, pp. 1 –10.

[19] S. Gurumurthy, S. Vasudevan, and J. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *Test Conference, 2006. ITC '06. IEEE International*, Oct 2006, pp. 1–9.

[20] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Test Conference, 2002. Proceedings. International*, 2002, pp. 590–598.

[21] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Reorda, "Microprocessor software-based self-testing," *Design Test of Computers, IEEE*, vol. 27, no. 3, pp. 4–19, May 2010.

[22] C. Grecu, P. Pande, A. Ivanov, and R. Saleh, "Bist for network-on-chip interconnect infrastructures," in *VLSI Test Symposium, 2006. Proceedings. 24th IEEE*, April 2006, pp. 6 pp.–35.

[23] F. Oboril and M. Tahoori, "Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, June 2012, pp. 1–12.

[24] A. Tiwari and J. Torrellas, "Facelift: Hiding and slowing down aging in multicores," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, Nov 2008, pp. 129–140.

[25] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Maestro: Orchestrating lifetime reliability in chip multiprocessors," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10, 2010.

[26] K.-J. Lee and K. Skadron, "Using performance counters for runtime temperature sensing in high-performance processors," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 8 pp.–.

[27] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[28] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[29] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *ISPASS 2009*, april 2009, pp. 33 –42.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, New York, NY, USA, 2008, pp. 72–81.

[31] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *Computers, IEEE Trans. on*, vol. 58, no. 8, pp. 1063 –1079, aug. 2009.

[32] O. Khan and S. Kundu, "Hardware/software codesign architecture for online testing in chip multiprocessors," *Depen. and Sec. Comp., IEEE Trans on*, vol. 8, no. 5, pp. 714 –727, sept.-oct. 2011.

[33] ——, "Thread relocation: A runtime architecture for tolerating hard errors in chip multiprocessors," *Computers, IEEE Trans on*, vol. 59, no. 5, pp. 651 –665, may 2010.

[34] Y. Li, O. Mutlu, and S. Mitra, "Operating system scheduling for efficient online self-test in robust systems," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 201–208.

[35] M. Kaliorakis, N. Foutris, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Online error detection in multiprocessor chips: A test scheduling study," in *IOLTS, 2013*, July 2013, pp. 169–172.

[36] J. Carulli and T. Anderson, "The impact of multiple failure modes on estimating product field reliability," *Design Test of Computers, IEEE*, vol. 23, no. 2, pp. 118–126, March 2006.

[37] "Opensparc t1 microarchitecture specification," Aug. 2006.

Michael A. Skitsas received his 5-year Engineer Diploma (*Dipl.-Ing.*) in Electronic and Computer Engineering from Technical University of Crete in 2009. Currently, he is a PhD candidate in ECE department at the University of Cyprus. His main research interest areas are Reliability and Testing of Digital Systems, Embedded Systems and Computer Architecture.

Chrysostomos A. Nicopoulos received the B.S. and Ph.D. degrees in electrical engineering with a specialization in computer engineering from Pennsylvania State University, State College, PA, USA, in 2003 and 2007, respectively. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus. His current research interests include Networks-on-Chip (NoC), computer architecture, and multi/many-core microprocessor design.

Maria K. Michael aria K. Michael holds B.Sc. and M.Sc. degrees in Computer Science and a Ph.D. degree in Electrical and Computer Engineering from Southern Illinois University, Carbondale (SIU-C). She is currently an Assistant Professor with the ECE Dept., University of Cyprus, Cyprus. Her current research interests include topics in EDA, test, reliability and fault tolerant design of digital systems and architectures. She serves on steering, organizing and program committees of several IEEE and ACM conferences in the areas of test and reliability. She is a Member of the IEEE and the ACM.