Noname manuscript No.
(will be inserted by the editor)

Exploring System Availability during Software-Based Self-Testing of Multi-core CPUs

Michael A. Skitsas · Chrysostomos A. Nicopoulos · Maria K. Michael

Received: date / Accepted: date

Abstract As technology scales, the increased vulnerability of modern systems due to unreliable components becomes a major problem in the era of multi-/many-core architectures. Recently, several on-line testing techniques have been proposed, aiming towards error detection of wear-out/aging-related defects that can appear during the lifetime of a system. In this work, firstly we investigate the relation between system test latency and test-time overhead in multi-/many-core systems with shared Last-Level Cache (LLC) for periodic Software-Based Self-Testing (SBST), under different test scheduling policies. Secondly, we propose a new methodology aiming to reduce the extra overhead related to testing that is incurred as the system scales up (i.e., the number of on-chip cores increases). The investigated scheduling policies primarily vary the number of cores concurrently under test in the overall system test session. Our extensive, workload-driven dynamic exploration reveals that there is an inverse relationship between the two test measures; as the number of cores concurrently under test increases, system test latency decreases, but at the cost of significantly increased test time, which sacrifices system availability for the actual workloads. Under given system test latency constraints, which dictate the recovery time in the event of error detection, our exploration framework identifies the scheduling policy under which the overall test-time overhead is minimized and, hence, system availability is maximized. For the evaluation of the proposed techniques, multi-/many-core systems consisting of 16 and 64 cores are explored in a full-system, execution-driven simulation framework running multi-threaded PARSEC workloads [1].

Keywords On-line Testing · Software-Based Self-Testing · System Availability

M. A. Skitsas [1,2]
E-mail: skitsas.michael@ucy.ac.cy

C. A. Nicopoulos [2]
E-mail: nicopoulos@ucy.ac.cy

M. K. Michael [1,2]
E-mail: mmichael@ucy.ac.cy

1. KIOS Research and Innovation Center of Excellence
2. Department of Electrical and Computer Engineering, University of Cyprus

1 Introduction

The era of nanoscale technology has resulted in massively complex systems consisting of billions of transistors. However, a side effect of this deep technology scaling is the exacerbation of the vulnerability of systems to unreliable components afflicted by aging and wear-out artifacts [2]. The issue of aging and gradual degradation necessitates the development of techniques that provide some form of protection against undesired system behavior. Such schemes broadly fall into two categories: (a) concurrent methods relying on fault-tolerant mechanisms (i.e., redundancy techniques), and (b) non-concurrent periodic on-line testing [3], which aims to detect errors that are – subsequently – addressed using various mechanisms.

Falling under the latter category, several on-line error detection techniques have been proposed in the literature, which enable the dynamic detection of permanent faults during the lifetime of a system [4][5][6]. Software-Based Self-Testing (SBST) is an emerging new paradigm in the testing domain, which relies on the exploitation of existing available resources resident in the

system. The SBST approach is based on software programs that are designed to test the functionality of the processor, and they target either general-purpose microprocessors [7][8][9][10][11][12][13][14], or embedded microprocessors and microcontrollers [15][16][17][18][19]. Specialized test routines (software programs) are executed just like 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.

One salient aspect of on-line testing (in general) is the *scheduling* of the testing session/process. In light of the rapid proliferation of multi-/many-core microprocessor architectures [20][21], the test scheduling issue becomes even more pertinent. One approach is to periodically initiate testing on *all* system cores simultaneously [22][23][24]. This method implies that the entire system will be offline during the duration of the test process, thereby interrupting the execution of other applications. Another approach is to initiate testing on individual cores that have been observed to be idle for some time [25][26][27]. Thus, the testing process in minimally intrusive, but the time required to complete the testing of all cores is substantially longer (since each core is individually tested at different points in time). Finally, testing may be *selective* (rather than periodic), targeting cores that have experienced prolonged stressing due to high utilization. Selective testing may be performed either at a full-core granularity, or at a sub-core granularity (testing individual intra-core components)[28].

In this work, we focus on periodic on-line SBST of the processor cores of homogeneous multi-/many-core systems with a shared and distributed Last-Level Cache (LLC). Memory testing and on-chip interconnect testing are beyond the scope of this work. A shared and distributed LLC is found in the vast majority of existing commercial Chip Multi-Processors (CMP). In such systems, each core in the CMP has a slice (bank) of the entire LLC. The work presented in this article comprises an extensive exploration of the test scheduling process in such systems. We assume that a testing session is complete when *all* cores in the microprocessor have completed their testing process. With this in mind, we perform an investigation of different test scheduling policies, based on the number of cores concurrently under test in the overall system testing session. We are motivated to study this problem, because, in shared memory systems, the time overhead of SBST for each core is affected by potential test program content – instructions and/or data – already resident in the LLC (as a result of a previous core's testing session).

The first goal of this work is to investigate the intricate relationship between the two aforementioned key metrics – the test latency and the test-time overhead – under different test scheduling policies. Typically, the system recovery mechanism imposes an upper bound on the test latency, because excessive test latency will lead to inordinate amount of wasted work (i.e., discarded work) in the event of an actual fault detection. Hence, given a specific test latency constraint, our exploration framework is able to identify the test scheduling policy that minimizes the test-time overhead and maximizes system availability. To the best of our knowledge, this is the first work in SBST for multi-/many-core systems exploring and juxtaposing these two important test metrics in a systematic way, so as to minimize the overall system availability. The second goal is related with the scalability of systems in terms of number of cores. As our target architectures are multi-/many-core systems, maintaining the performance of the proposed testing methodologies is very important and crucial for the applicability of such methods as the number of cores within a system is increased. In our experimental evaluation, we investigate the behavior of the proposed test scheduling methodologies when the number of cores in the system quadruples from 16 to 64. Results show that the test-time overhead metric and, therefore, the test latency are increased. The main reason for the additional overhead is the larger Network-on-Chip (NoC), which causes higher latencies when data is fetched from the LLC to the private cache of the core under test.

In order to mitigate the increased testing overhead as the multi-core system scales up (i.e., the number of on-chip cores increases), we introduce a *clustering* approach. The CMP is divided into a number of contiguous core clusters, i.e., each cluster comprises a number of CMP processing cores. The main idea behind this approach is to keep the test-related data resident in the LLC of each cluster as close as possible to the core under test. Indeed, using a clustering approach during the test of a core in the system, we manage to keep the LLC shared test data within the LLC banks of a number of adjacent cores in the vicinity of the core-under-test. Thus, the test program's LLC shared test data is distributed and kept across the LLC slices of each core *cluster*, instead of being uniformly distributed across the entire system's LLC. This technique can reduce the overhead to fetch the data from the LLC to the private cache of each core under test, and, consequently, assists in the reduction of the testing overhead.

The evaluation of the various test scheduling policies is performed using the previously proposed Dae-

monGuard framework [28]. DaemonGuard is a lightweight and minimally intrusive O/S-resident framework that manages the SBST procedure in multi-/many-core systems. The operating system daemons employed by DaemonGuard initiate periodic SBST testing on the CPU cores. The overall exploration is performed using an execution-driven, full-system simulation framework running a commodity operating system and executing the PARSEC benchmark suite [1] (a selection of emerging multi-threaded applications) in a multi-core setup under 16-core and 64-core CMP systems.

2 Related Work

Recently, several techniques have investigated the scheduling of test routines in multi-/many-core systems under SBST. Apostolakis et al. [22] proposed a methodology that allocates the test programs and test responses into the shared on-chip memory, and schedules the test routines among the cores. The aim of the work in [22] is to reduce the total test application time, assuming that all cores are tested simultaneously (i.e., full-system parallel testing).

Haghbayan et al. [27] proposed a power-aware non-intrusive online testing approach for many-core systems. The proposed approach schedules software-based self-test routines on the various cores during their idle periods. The scheduler selects the core(s) to be tested from a list of candidate cores. The selection is based on a criticality metric, which is calculated considering the utilization of the cores and power budget availability.

A Multi-Threaded (MT) SBST methodology was proposed in [23], in order to reduce the test execution time, based on the thread-level parallelism capabilities of the core under test. Specifically, functional-based test programs are scheduled onto individual multi-threaded cores, and the focus is on the optimization of the test time of a single core.

Yanjing Li et al. [26] developed a test-aware OS scheduling technique for robust systems. A test controller selects a core to be tested in a round-robin fashion, and – once the core is selected – the OS scheduler performs online self-test-aware scheduling, in order to schedule the test program on the selected core with minimum disruption to the normal workloads running on the system. The impact of simultaneously testing multiple cores is not considered.

In [29], the authors propose a test-program parallelization methodology for many-core architectures, in order to accelerate the online detection of permanent faults. The underlying architecture does not have a shared cache, but, instead, it relies on high-speed message passing for data sharing among the cores. Moreover, the work in [29] only examines fully parallel system testing (i.e., testing all cores simultaneously), leading to zero availability during the SBST session.

In [30][31], the authors proposed a scalable self-test mechanism for online testing of many-core processors. Software test routines are distributed among the cores of the system using hardware components that monitor the behavior of the processing cores.

3 Definitions and Framework Overview

A *testing session* is defined as the time interval required to test all cores in the system. The evaluation metrics that are used in the exploration are: (a) the *Test Latency* (TL), defined as the total time required to complete a testing session (i.e., elapsed time between initiation and completion of testing), and (b) the *Test-time Overhead* (TO), defined as the total execution time devoted to the test programs of all the cores in the system. Based on these two fundamental metrics, we derive a new metric, termed *System Availability during Test* (SAT), which is the percentage of time the system cores are available during a testing session of a given test latency. During this time, the system is able to continue execution of normal workloads, maintaining system availability.

The execution of test programs on the cores of a multi-core system employing shared memory (and shared LLC) could benefit in terms of test-time overhead and test latency. Test programs having the same text segment (test instructions) and data segment (test patterns) could share data among different cores – through the LLC – during the test execution. This sharing phenomenon could be observed in cases where: (a) test programs are executed in parallel over several cores; (b) test programs have some execution-time overlap between the various cores; (c) a test program is executed right after (or shortly after) another core's test session. In all these cases, part of the test program's content may still be resident in the cache hierarchy. On the other hand, the parallel execution of test programs on multiple cores could lead to further overhead, either by increasing the on-chip network contention (due to increased requests to the memory system for the same data), or by reducing the system throughput (since the available cores for normal operation are limited due to the testing process). This realization motivates us to investigate the parameter of the number of cores concurrently under test, and how this test attribute affects test-time overhead and the test latency.

Beyond the number of cores concurrently under test and how they affect the considered metrics, we inves-

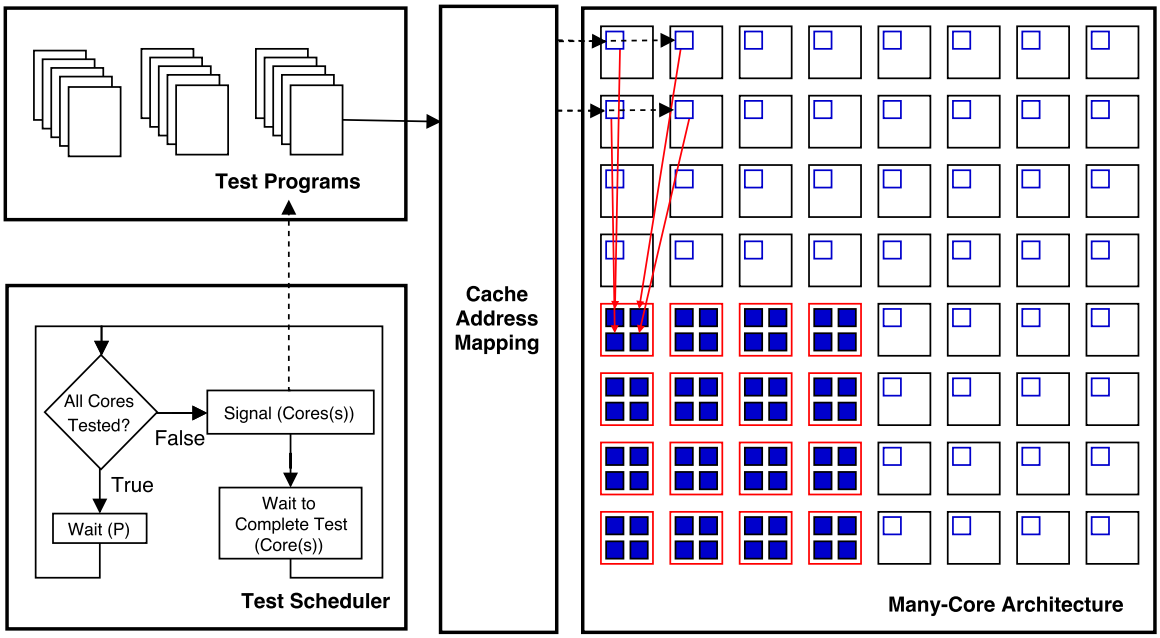

Fig. 1 Architectural overview of the employed framework. The two main components are (1) the Test Scheduler (an OS process), and (2) the actual Test Programs, which all operate at the OS level. The *Cache Address Mapping* component is responsible for the mapping of physical addresses to the cores within a cluster when employing the clustering approach. The figure illustrates an example of the clustering approach, whereby the test program data – that would otherwise be distributed across the entire CMP – is mapped (red lines) within the cluster area (solid blue squares). Without clustering, the test program data would go in the empty blue squares across the entire system.

tigate the behavior of the proposed solution when the system scales up (i.e., from 16 to 64 cores). As the system scales up, experimental results indicate that the NoC incurs a significant overhead to the testing procedure. In order to overcome this and eliminate the extra network overhead, we use a clustering approach, where the testing process is considered over a cluster instead over the entire system. Again, the testing procedure is completed when all the cores of all the clusters of the system are tested. In this work, a *cluster* is defined as a group of cores within the many-core system where the aforementioned test scheduling policies will be applied. When a test program is loaded on a cluster for execution, the data are fetched and uniformly distributed over the LLC banks of the cores under the considered cluster. With this technique, we reduce the latency of exchanging data between the LLC and the private cache of the core under test. This is achieved by limiting the distribution of data in the LLC to the region where the cores form a cluster, instead of distributing the data across the entire system's LLC.

To implement the proposed test scheduling scenarios, we use a framework operating at the Operating System (OS) level, as abstractly depicted in Fig. 1. The two main components are: (1) the test scheduler, and (2) the actual test programs. In order to perform SBST, a number of test programs are loaded onto the OS. The number of test programs depends on the number of the cores present in the system: we need one test program for each core. Test programs are regular processes loaded on the OS, so they have a portion of the main memory allocated to them. However, since the considered system is a *homogeneous* many-core system, it means that all cores are tested using the *same test program*. Thus, the memory footprint is independent of the number of cores in the system. The test programs are kept in idle mode during normal operation; they wait for the appropriate invocation signal from the test scheduler process, in order to wake up and perform their test execution on the targeted core. Note that the test scheduler is an OS process, which is loaded and executed at the OS level. The test scheduler process is responsible to orchestrate the testing process during a testing session. According to the test scheduling policy, the test scheduler sends a wake-up signal to the test program that is assigned to the selected core for testing. The execution of the test programs is managed by the OS just like the execution of any other program(s) running on the system. The OS can still perform context switches between test programs and actual applications. Hence, the OS always ensures that the core is not "blocked" by the test program for an excessively long time. This allows the OS to time-multiplex the test program with other running applications, as well as to serve high-priority interrupts

that may arrive at any given time. Upon completion of a test program's execution, the test scheduler is notified and proceeds with the test scheduling procedure. The OS-resident test scheduler and test programs described here (and shown on the left-hand side of Fig. 1) are facilitated in this work by the DaemonGuard framework [28].

The implementation of the clustering approach is achieved in our simulation framework (DaemonGuard) with the modification of the cache management units and, particularly, the module responsible for the *cache address mapping*. In real systems, the implementation of the mapping component can be done at the OS Level and/or with modifications at the micro-architectural (hardware) level. In the literature, several works have proposed techniques that allow the dynamic mapping of data to specific locations within a shared cache. Schemes to control data placement in large caches by modifying the physical addresses are studied in [32]. In [33], the authors proposed a hardware method that employs a new level of indirection for physical addresses, allowing for highly flexible data mapping. The implementation and evaluation of such techniques is orthogonal to and beyond the scope of this article. In our work, when we employ the clustering approach, we assume the presence of such a dynamic data mapping mechanism, which facilitates core clustering. Our focus here is solely on the *scheduling policies* of the test scheduler in many-core systems, with and without the clustering approach.

4 Test-Scheduling Exploration

4.1 Parameters affecting the testing process

Several parameters could affect the system behavior during the testing process. These parameters are directly related to the test-time overhead and test latency metrics. The first design parameter that could affect the testing process is the test program size. As the memory footprint of the test program increases, the test-time overhead also increases, due to memory-, processor-, and network-related latencies. Next, memory system parameters, such as the LLC size, the cache organization, and the employed cache coherence protocol could also affect the testing process. The LLC size is closely related to the test program size; a larger LLC could reduce the test-time overhead, since more data could reside in the cache during the test process. Another important parameter is the CMP size itself (in terms of number of cores). Specifically, the total number of cores in the system and, therefore, the number of cores concurrently under test, directly affect the test overhead during the testing sessions. Finally, the on-chip communication network (the NoC) is another parameter that could affect the testing process. The impact of the latter parameter becomes more important as the number of cores increases and, therefore, the size of NoC increases, too. Increased distance between the cores of the system negatively impacts the testing overhead, since extra delay is imposed for the completion of testing.

Beyond the testing procedure itself, all of the above parameters affect – to varying degree – the clustering approach as well. Basically, these parameters will help in determining the cluster size and, thus, the number of clusters in the system. More details about the clustering approach will be provided shortly, in Section 4.4.

In this work, we assume that there is a given (fixed) test program that targets the cores of a homogeneous multi-core system. Also, during the lifetime of the system, the parameters that are related to the CMP architecture and memory system remain unchanged. To perform the proposed exploration, we focus on the number of cores concurrently under test, and the clustering approach as the system scales up. These parameters could vary between different testing sessions, since it is entirely under the control of the test scheduler process.

4.2 Scheduling policies

In order to evaluate the test-scheduling process, we propose test scheduling scenarios that vary the number of cores concurrently under test. We evaluate three general scheduling scenarios. In the first, the test scheduler invokes all the test programs simultaneously, in order to test all the cores of the system at the same time. This case corresponds to *parallel* testing, whereby all the cores are under test simultaneously. During such a testing session scenario, normal workloads running on the cores of the system must be suspended. Thus, the system availability will be reduced to nearly zero, since all cores are under test (our simulations have shown that due to the shared memory, the test execution time per core is not identical, but it may vary slightly).

On the other extreme, the second scheduling scenario considers a *serial* execution of test programs during each testing session. This scenario does not exhibit any testing overlap among cores, because only one core is under test at any given time. Initially, the test scheduler sends a signal to commence testing of the first core. Then, when the end notification is received, the scheduler proceeds with the second core, and so on. The interval between two consecutive core tests must be as short as possible (ideally zero), in order to reduce the test latency and to benefit from test data already residing in the cache hierarchy.

The last scheduling scenario aims to bridge the gap between the first two. It initiates sequential testing among subsets of the cores of the system. In this scenario, the first core is tested alone at the beginning of the testing session; this core is known as the "pilot" core, i.e., the first core to bring the test instructions and data from the off-chip main memory into the on-chip cache hierarchy. Subsequently, the remaining cores are tested in groups, with each group being concurrently under test. In particular, the test scheduling policy will have the maximum number of cores k that could be concurrently tested as an input parameter. The untested cores of the system will be divided in groups of k cores. When all k cores within a group complete their tests, the test scheduler will initiate simultaneous testing on the next k cores, and so on. We assume in this work that *any* core can be selected as the pilot core, and *any* k cores can be selected for testing at any given time, i.e., the order of selecting the cores for testing is irrelevant.

The decision of using a pilot core to execute the test program alone has a two-fold advantage. The first one was briefly mentioned above: at the beginning of each testing session, the test program content (instructions and data) is not resident in the cache hierarchy. As a result of this, instructions and data will be fetched by the pilot core, since this is the first core to execute a test program in the particular test session. This could be considered as a means to pre-fetch test data for the remaining cores in the system. The second advantage of having a pilot core is related to the availability of the system. The execution of a test program by the pilot core is characterized as a time-consuming process, since all data will be fetched into the LLC. Using the pilot core, this process will be handled by one core of the system (or the cluster, when using the clustering approach), while the remaining cores are available to execute normal workloads. In other words, during the time-consuming process – due to the LLC misses – of the execution of the test program by the pilot core, we ensure the highest possible system availability (all the other cores continue to execute normal workloads).

Since this work assumes the use of full-core testing, the test program in our case is, by construction, a single program that tests the entire core. Moreover, since we assume a homogeneous multi-/many-core system, all cores are identical and, thus, they all use the same test program. Nevertheless, in a different environment, there may be a *set of test programs* that need to be executed. If (a) these programs must all be executed to complete a test session (e.g., because each test program covers a different fault type, or a different component of the CPU, etc.), and (b) we know the specific core that each program targets, then the complexity of the proposed test scheduling methodology will not be affected, since all programs (for each core) will be "grouped together" and treated as one uniform "super program" by the scheduler.

As mentioned in Section 3, the scheduling process is the responsibility of the test scheduler OS process. Algorithm 1 presents the pseudo-code implemented by the test scheduler. The test scheduler is in idle mode during normal (non-test) operation, in order to incur the minimum possible overhead to the system. The scheduler simply waits (sleeps) for a period P, before waking up to initiate and manage the testing process. In this work, we focus on the exploration of system availability during a single testing session. The period P, which determines the testing frequency, is defined by the user. In general, the frequency of testing can depend on various factors, such as usage, temperature, criticality of programs, etc. Moreover, the frequency of testing can be determined by the fault-detection *latency* that the user is willing to accept. While some sectors/domains may be willing to tolerate long fault-detection latencies, there are mission-critical applications that require much faster detection latency. Function $SendSignal(c)$ is used to initiate the test program assigned to core c by sending a wake-up signal. The $WaitCores(listC)$ function is used by the testing scheduler to wait until the completion of the test programs of the cores in list C. When the clustering approach is used, the test scheduler runs the same algorithm. The only difference is in the input, and, specifically, the List of Cores C. In the clustering case, instead of giving all the cores of the system as an input, the list of cores includes only the cores contained within the cluster under test. Algorithm 1 can be trivially modified to give priority to idle cores when selecting the pilot core, or the next k cores to test.

4.3 Optimization

Considering the two fundamental metrics of test-time overhead and test latency, we propose an optimization formula, in order to find the maximum number of cores that should be concurrently tested (i.e., tested at the same time, in parallel) at any given time, in order to maximize the system availability during test (SAT), subject to a test latency constraint. This amounts to identifying an optimal parameter k, described in the previous sub-section. As system availability is defined based on test-time overhead and test latency (see Section 3), we, in fact, optimize the ratio of these metrics. SAT_k is the system availability during the concurrent testing of k cores, and it is calculated by Equation 1.

Algorithm 1 Test Scheduler

Input: Period P
Input: List of Cores C
Input: Number of Cores Under Test k

Cores under Test List CUT

```
1: while True do
2:     CUT = []
3:     pilot ← C.dequeue()
4:     SendSignal(pilot)
5:     CUT.enqueue(pilot)
6:     if k = N then
7:         k = k − 1
8:     else
9:         WaitCores(CUT)
10:    end if
11:    while C not Empty do
12:        for i=1 to k do
13:            nc ← C.dequeue()
14:            SendSignal(nc)
15:            CUT.enqueue(pilot)
16:        end for
17:        WaitCores(CUT)
18:    end while
19:    Sleep(P)
20: end while
```

$$SAT_k = \frac{TL_k \times N - TO_k}{TL_k \times N} \quad (1)$$

The terms TO_k and TL_k are the test-time overhead and test latency, respectively, under the scheduling scenario of having k cores concurrently under test at a time.

Based on Equation 1, and given a test latency constraint, we aim to find the maximum possible SAT using the optimization formulas described in Equations 2 and 3.

$$SAT_{max} = \max_k \{SAT_k\}, k = 1..N \quad (2)$$

$$subject\ to\ TL_k < L \quad (3)$$

The term SAT_{max} is the maximum system availability, $k = 1\ to\ N$ corresponds to the number of cores that are concurrently under test, N is the total number of cores in the system, and L is the maximum test latency constraint.

Using this optimization objective, we aim to find the number of cores concurrently under test (i.e., k) that maximizes the system availability, while taking into account the test latency constraint L.

The proposed optimization formula considers metrics that are focused entirely on execution *time* (i.e., test latency and test-time overhead). Another parameter that could affect the testing overhead and, therefore, the system availability is *fault coverage*. However, in this work, we assume that the test programs are fixed to one particular fault coverage, i.e., we did not consider the presence of multiple versions of the test programs, with each version providing a progressively higher fault coverage at the cost of increased test-time overhead. In such case, the test scheduling optimization process would then include an additional parameter pertaining to the targeted fault coverage.

Beyond the consideration of only time-related metrics, the optimization process could, potentially, include other salient metrics, such as *power consumption*, which may be very critical in portable/mobile devices. When multiple programs (including the test programs) run in parallel on any CPU, the processor utilization increases, which results in higher power consumption than when the CPU is lightly utilized. However, this increase in power consumption is guaranteed not to exceed the CPU's Thermal Design Power (TDP), since the latter is the maximum that can be generated under any software workload (including functional SBST test programs). Nevertheless, using power consumption as an additional optimization parameter in our test scheduling policy is an interesting extension to the current framework.

4.4 Scaling to many-core systems: a clustering approach

The proposed test-scheduling approach works very effectively in relatively small-scale multi-core systems (e.g., with 16 on-chip CPU cores). The critical objective is to ensure scalability of the proposed framework as the system grows into the many-core realm, i.e., with tens – or even hundreds – of cores. To address this imperative goal, we introduce a clustering approach in our testing methodology, which ensures the high performance of the proposed test scheduling techniques regardless of the size of the system. With the clustering approach, the system is divided into a certain number of core clusters. The testing process is then conducted at the granularity of individual clusters; the cores of each cluster are tested following any of the scheduling policies described in the previous sub-section. A key assumption when employing the clustering approach is the presence of a mechanism that allows for *dynamic mapping of data to specific locations within a shared cache*, as described at the end of Section 3. In our case, the test-related data is mapped to the cores of each cluster, rather than being distributed across the cores of the entire CMP.

As mentioned in Section 4.1, several parameters could affect the configuration of the clusters, i.e., the cluster size (the number of cores grouped within a cluster), and, subsequently, the total number of clusters in the system. Additionally, the decision of using a clustering

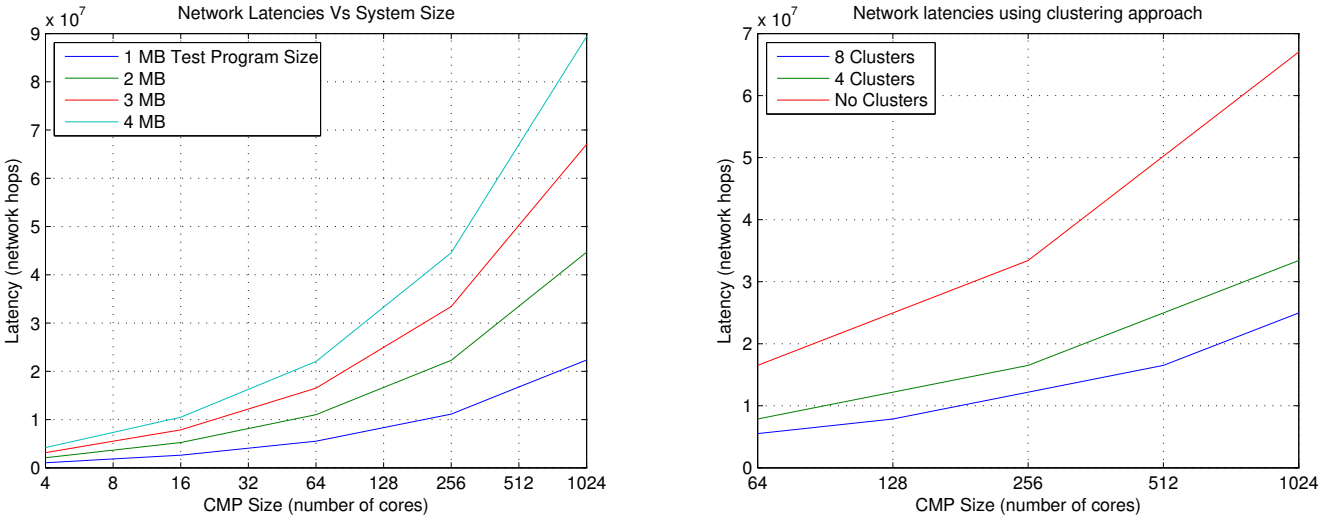

Fig. 2 A high-level statistical analysis investigating the on-chip network latency (in terms of network hops) as the number of on-chip cores in the CMP increases. Results in (a) the absence of clustering, and (b) in the presence of clustering are depicted. In the latter case, the size of the employed test program is set to 3 MB.

approach in the first place within a system is based on these parameters as well.

The clustering approach aims to reduce the imposed testing overhead as the system scales up. When the number of cores increases, the distances between the cores of the system also increase, which results in longer NoC delays. To evaluate the impact of the network and to investigate the potential of the clustering approach, we proceed with a high-level statistical analysis/exploration. In modeling the network, we assume that one network hop is required to transfer data between two adjacent cores (i.e., each CPU core is connected to its own on-chip router). The cost of transferring data between any two cores in the system is calculated based on the Manhattan distance between the two cores. This implies the use of a mesh-like NoC topology, which is most frequently encountered in the literature and even in recent commercial products. Moreover, as the goal of the analysis is to evaluate the network impact, we consider the case where all the test data is resident within the system (or within one cluster). The network latency is abstracted as the number of hops required to fetch all the required test data to the core under test. Beyond the scaling of the system itself (in terms of number of cores), we also include in our analysis cases where the test program size is also changed.

The results of our statistical analysis are depicted in Figure 2(a). Said figure shows the on-chip network latency (in terms of network hops) as the number of on-chip cores in the CMP increases all the way to 1024. The four different curves correspond to four different test program sizes. Obviously, the network latency increases exponentially as the system size increases. Furthermore, the network latency is also negatively affected by the test program size. This worrisome trend motivates the need for a different approach that would eliminate the exponential increase in network latency. Toward this end, we adopt the clustering approach, which breaks the large-scale system into a number of smaller core clusters.

The statistical analysis was repeated in the presence of the clustering approach. The results are shown in Figure 2(b). In this case, the focus is on systems ranging from 64 to 1024 cores, and three different configurations are juxtaposed: absence of clustering, dividing the CMP into 4 clusters, and dividing the CMP into 8 clusters. For example, a 512-core CMP with 8 clusters implies 64 cores per cluster, and so on. Note that the size of the employed test program is set to 3 MB in this experiment, i.e., similar to the size of the test program used in this work. The results in Figure 2(b) clearly indicate that the use of clustering almost linearizes the increase in the network latency as the system scales up. A linear (or super-linear) increase is certainly more desirable and practical than the exponential increase observed in the absence of clustering.

One key requirement – and elemental contributor to this scheme's effectiveness – is that the entire test program should fit within the LLC banks of the cores of each cluster. This is a fairly intuitive requirement, since the goal of clustering is to maintain the required test data within a cluster of cores, in order to expedite the testing process. Consequently, the minimum number of cores comprising each cluster is dictated by the size of

the test program's data and it depends on the size of each core's LLC bank (slice).

At the beginning of testing of each cluster (i.e., when the first core of each cluster will initiate testing), there is no test data available within the cluster's LLC slices. The test data required to test the cores of each cluster must somehow be brought within the cluster. To achieve this, there are two possible solutions: (1) use a pilot core (see Section 4.2) in each cluster, which will essentially pre-fetch all the test data for the remaining cores within the cluster; and (2) migrate the test-related data from a cluster that has just finished being tested to a new cluster-to-be-tested. Of course, the second solution presupposes that clusters are tested serially, one after the other. Instead, the first solution allows for the parallel testing of multiple clusters, if desired. Due to this flexibility, we employ the first solution (i.e., the use of a pilot core within each cluster) in our quantitative analysis in the next section. In any case, the test scheduling methodologies described in Section 4.2 can be applied to either of the aforementioned two approaches (for the testing of the cores within each cluster).

5 Experimental Framework and Results

5.1 Evaluation Framework

For the evaluation of the proposed test scheduling techniques, we perform full-system, execution-driven simulations using the Wind River's Simics [34] simulator extended with the Wisconsin GEMS toolset [35] and the GARNET network model [36]. We simulate two multi-core systems, as presented in Table 1, one with 16 cores in a 4×4 network topology and one with 64 cores in an 8×8 network topology. In both systems, each core is a SPARC-based in-order-execution processor, similar to the UltraSPARC III+. For the memory hierarchy, we considered a two-level cache system with two split private caches at L1 (for instructions and data), and a shared LLC (L2). The L2 banks (slices) are distributed equally (in size and configuration) to all the cores of the system. As the targeted system is homogeneous, and our goal is to test all the cores of the system, we use the full-core test program given in [28] to test each individual core according to the proposed test-scheduling techniques. Note that the fault coverage of the proposed methodology depends entirely on the employed test programs. The particular *level* of fault coverage provided by the test programs is related to the methodologies followed during the generation of the test programs themselves. The test-program generation process is beyond the scope of this work, and it can be viewed as orthogonal and complementary to the proposed framework.

System	16-Core CMP	64-Core CMP
Processors	16 UltraSparc III+	64 UltraSparc III+
Network	4×4 2D Mesh	8×8 2D Mesh
L1 Caches	32 KB I&D, 2c lat	32 KB I&D, 2c lat
L2 Caches	1 MB/core, 10c lat	0.5 MB/core, 10c lat
Main Memory	4 GB, 200-cycle latency	
OS	Solaris 10	

Table 1 Simulated system parameters.

Through our experimental exercise, we investigate the impact of test-time overhead and test latency during a testing session under all the possible scheduling scenarios discussed in the previous section. Similar experiments are applied in both systems. Hence, we consider the case of serial testing where only one core is tested at a time during the testing session ($k = 1$), the parallel case where all cores are tested in parallel ($k = N = 16$ *or* 64), and all the other cases in between where a fixed number k (power of 2) of a subset of the cores are tested in parallel at a time ($k = 2, 4, 8...$), after the test of the pilot core. We present results for both systems (16- and 64-core), and the different scheduling policies are compared for each system size. Furthermore, the scalability impact of transitioning from the 16-core CMP to the 64-core one is presented and analyzed within the context of the results of the newly proposed clustering approach.

We use workloads from the PARSEC Benchmark Suite [1] in our exploration. PARSEC is a benchmark suite of multi-threaded workloads that focus on emerging parallel workloads. For the evaluation, we use eleven of the benchmarks for both explored system sizes. The input size of the benchmarks is set to the maximum possible (large), in order to ensure that the execution of the benchmarks is not finished prior to the completion of the testing process. The scope of this work is the evaluation of the proposed test-scheduling techniques while the system is running normal workloads (i.e., PARSEC Benchmarks). The number of threads is configured to be equal to the number of cores in each system (16 or 64), in order for all cores to be fully-utilized.

5.2 Exploration Results

We simulate all the aforementioned benchmarks for each scheduling policy and system to evaluate the impact in terms of test-time overhead and test latency. To eliminate the extra impact imposed by the OS due to scheduling priorities that affect the considered metrics (TO and TL), we increased the scheduling priority

of test programs to the maximum possible for a non-privileged user (i.e., not OS admin user). This allows us to perform a fair comparison between the different scheduling policies by avoiding any overhead from the OS due to context switching between different (non-test) running processes. As a result of this, the only imposed overhead beyond the testing procedure is due to memory requirements (i.e., cache used by normal workloads). To investigate the impact of the memory system, we repeat the experiments by increasing the LLC cache associativity in both systems, without affecting the total LLC cache size. As it will be shown, the experimental evaluation indicates that the number of misses in the case of test programs is reduced.

The next two paragraphs present the simulation results for the two considered systems, i.e., the 16-core and the 64-core CMPs. The test-time overhead (TO) and test latency (TL) for each scheduling scenario, as well as the System Availability under Test (SAT) for two different cache configurations, are evaluated for both systems. Additionally, results related with the scalability of the system are presented while evaluating the 64-core CMP setup.

5.2.1 Performance of a 16-core CMP

Figure 3 presents the results of the 16-core CMP system for all examined PARSEC benchmarks (each curve corresponds to a different benchmark). The presented results pertain to the three metrics (one metric per each row of plots) for two LLC cache configurations: a setup with 4-way LLC associativity, and one with 8-way associativity (each column of plots). Recall that *the total LLC size is the same in both cases*. For all the plots of the figure, the x-axis gives the number of cores tested concurrently (in-parallel) at any given time. Hence, the case where 1 core is under test gives the results for the serial scheduling scenario, where one core at a time is tested. The right-most "All" scenario on the x-axis refers to the case where all cores are tested in parallel. More accurately, the scenario "All" assumes that the first core in an n-core system serves as the pilot core, and, subsequently, the $n-1$ remaining cores are all tested in parallel. The y-axis reports the investigated test metrics (TO, TL, SAT) in terms of overall system execution cycles.

The first – and expected – outcome of this simulation is the increase in test-time overhead as the number of cores tested concurrently increases. In particular, by doubling the number of cores concurrently under test, test-time overhead (first row of Figure 3) is increased by a factor of 9–15%, depending on the size of k. The "All" case imposed an increase of 20% over the serial case. Hence, a single parameter optimization (in this case) would suggest that the fully-serial scenario (case $k = 1$) should be selected. However, this scenario comes with a great cost in test latency, as shown in the second row of Figure 3. Actually, juxtaposing the two figures (first and second row) reveals the inverse relation between the two test metrics (which is, to some extend, expected). However, this analysis also reveals potentially good compromises for optimizing both measures. For example, increasing the number of concurrently tested cores from 1 to 2 leads to a significant test latency reduction, at the cost of a small test overhead increase. Given a realistic test latency constraint L, one can decide on the number of concurrently tested cores, in order to minimize overall system test overhead subject to the given constraint.

An interesting point arising from the experimental evaluation of the scheduling techniques is the behavior of the different workloads. As we can see in Figure 3, the majority of benchmarks have the same impact on all the consider metrics. Nevertheless, some benchmarks incur an extra test-time overhead, even though the Test Latency is not always correspondingly affected. For instance, The TO behavior under the Canneal (red line) and Fluidanimate (yellow line) benchmarks is markedly different than under other benchmarks. The reason for this peculiar behavior are LLC conflicts. Since the test programs and the benchmark applications run concurrently, there are cache conflicts which affect testing under some benchmarks more than under others.

To verify this assertion and to provide a possible solution to this problem, we increased the cache associativity. In particular, we doubled the LLC associativity from 4-way to 8-way, while the size of each cache lines was halved, in order to keep the total cache size unchanged. The results of these experiments (8-way cache system), and the impact on the two considered metrics (TO, TL) are presented in the plots of the second column of Figure 3. The test-time overhead and test latency still have the same trends, but – when considering absolute values – there is a slight reduction in the overheads. Furthermore, benchmarks that behaved erratically (outliers) in the first set of experiments (4-way) now seem to follow the same trend as the rest of the benchmarks. This shows that the increase in LLC associativity "smooths out" the issue of cache conflicts among the benchmarks and the test programs.

The last set of plots (third row of Figure 3) depicts the system availability under test (SAT) for each of the scheduling scenarios under exploration, calculated using Equation (1) (see Section 4.3) for both cache setups (4-way and 8-way). The SAT metric combines the two test metrics under consideration and reveals the best

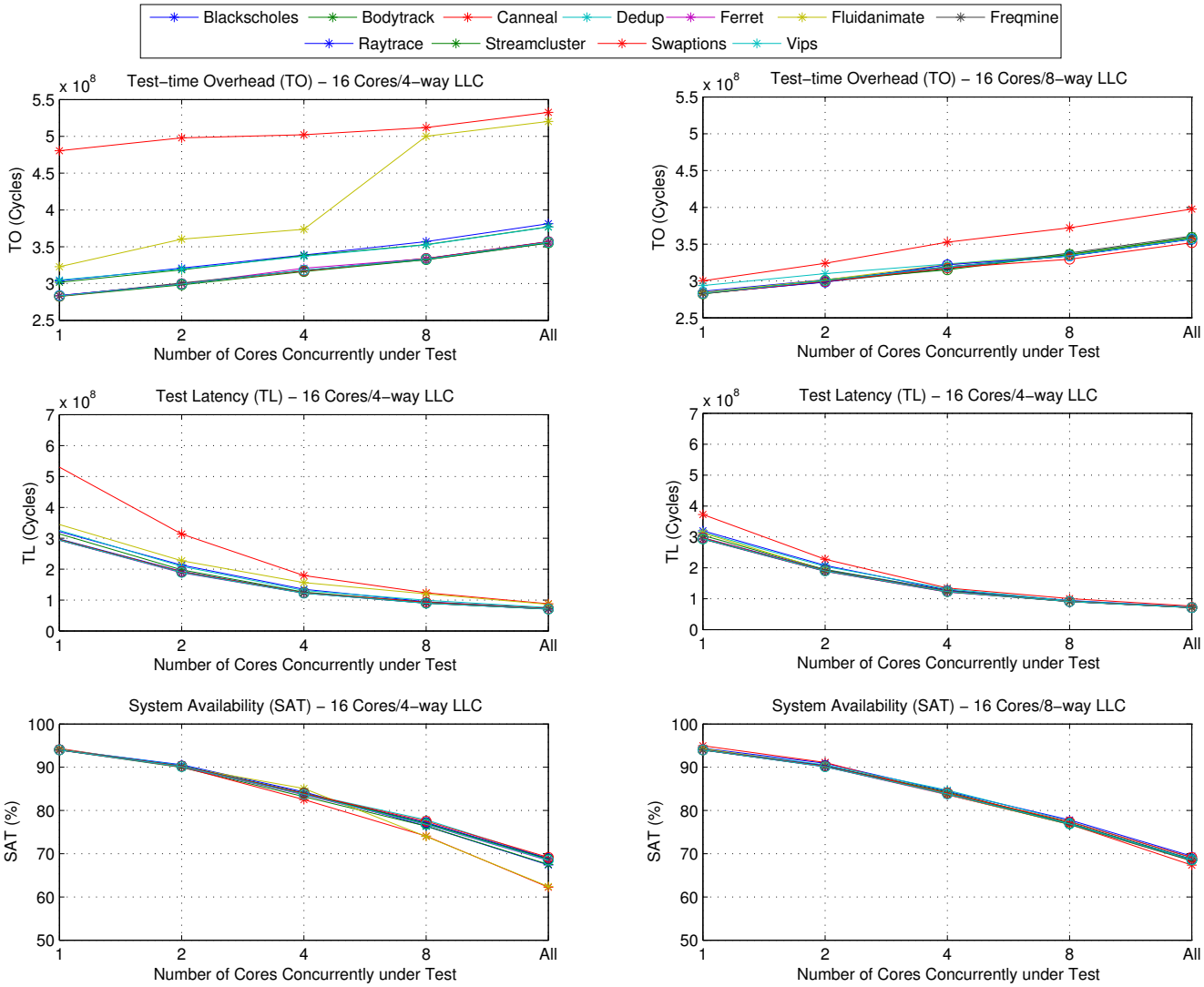

Fig. 3 The results of the *16-core CMP* system for all examined PARSEC benchmarks. Each row of plots corresponds to one of the three evaluated metrics. The left-column plots corresponds to a 4-way LLC, while the right-column plots correspond to an 8-way LLC. The total LLC size for both setups is the same.

scheduling scenario to maximize availability, which, in the case of no test latency constraints, is the same as the one minimizing the test overhead. As expected, the system availability is highly related to the number of cores under test at any given time, and the overall trend between the various benchmarks is similar.

5.2.2 Performance of a 64-core CMP

In order to evaluate the proposed test scheduling techniques in larger – in terms of core numbers – systems, where the impact of the NoC is significant, we also investigate a 64-core CMP setup. Beyond the evaluation of the proposed testing techniques in larger systems, the purpose of this experimental exercise is to also identify any scalability issues resulting from the increased system size. In fact, the results of this exercise will pave the way for the clustering approach, which will be shown to be necessary in maintaining the scalability of the system.

The exploration exercise under the 64-cores CMP system includes all the scheduling techniques (serial, parallel, and k-cores concurrently under test) investigated with the 16-core CMP system. Figure 4 presents the results of the evaluation of the test scheduling policies for the three considered metrics under a 64-core CMP system. Again, there are three rows of plots corresponding to the three metrics, and two columns for the 4-way and 8-way LLC configurations.

Evidently, the trends in the three metrics (TO, TL, SAT) are the same as under the 16-core CMP system. As the number of cores concurrently under test is increased, the TO also increases, while the TL decreases.

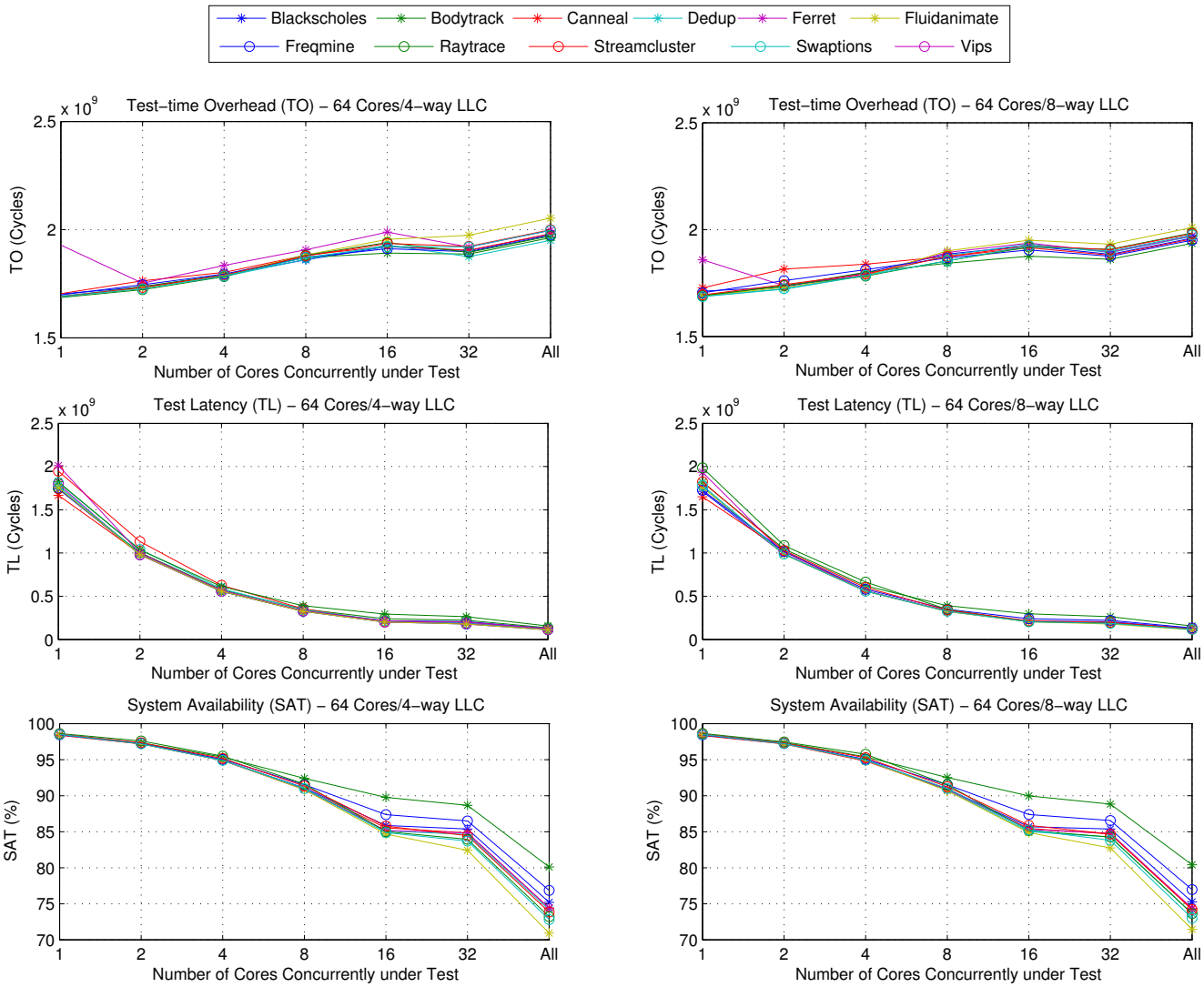

Fig. 4 The results of the *64-core CMP* system for all examined PARSEC benchmarks. Each row of plots corresponds to one of the three evaluated metrics. The left-column plots corresponds to a 4-way LLC, while the right-column plots correspond to an 8-way LLC. The total LLC size for both setups is the same.

The impact of the cache is also demonstrated by doubling the LLC associativity from 4-way to 8-way, while keeping the same total LLC size. As a result of this increase in associativity, the bahavior of all benchmarks is "smoothed out", i.e., there are no more outliers (exhibiting unusually high TO). Note that the Test-time Overhead is increased by a factor of 5–10% depending on the size of k. The "All" case incurred an increase of 15% over the serial case.

Despite the same trends in our metrics, at the core-level, the incurred overhead by the test program execution is increased. In particular, the required time to execute the test program for a core in a 64-core CMP system is higher than in a smaller system (16 cores). The average required time in terms of cycles to execute the test program in the 64-core CMP system is increased by 60%, as compared with the 16-core CMP system. Table 2 presents statistics derived from the experimental evaluation regarding the execution time of the test program on each core of the system. The table shows the number of cycles needed to run the test program on the pilot core, and the minimum, maximum, and average numbers of cycles needed to run the test program on each of the remaining cores of the system. As indicated in Table 2, when executing the test programs in larger systems, the execution time is considerably higher. The main reason for this increase is the larger distance between the cores and, therefore, the latency to fetch the data in the private caches of the core-under-test (either from main memory or the LLC) is significant. In an effort to mitigate this distance-related overhead, we propose the use of a clustering approach, as explained

System	Pilot	Min	Avg	Max
	(Millions of Cycles)			
16-Cores	50	14.5	15.5	16.5
64-Cores	58	22	25	31

Table 2 The number of cycles needed to run the test program on the pilot core, and the minimum, maximum, and average numbers of cycles needed to run the test program on each of the remaining cores of the system.

System	Min	Avg	Max
	(Millions of Cycles)		
w/o Clustering (64-Cores)	22	26	30
w/ Clustering (4 16-Core Clusters)	15.5	18	21.5

Table 3 Per-core Time-test Overhead (TO) assuming a 64-core CMP being tested with and without the clustering approach.

in Section 4.4. The main premise of the clustering approach is to maintain all the test-related data within the vicinity of the cores to be tested.

5.3 Evaluating the Clustering Approach

In this sub-section, we evaluate the effectiveness of the clustering approach of Section 4.4. We employ a 64-core CMP, which is divided into 4 symmetrical 16-core clusters; each cluster is a quadrant of the 64-core system. We assume the use of one pilot core in each of the four clusters. The test program data is fetched by the pilot core of each cluster and distributed all over the L2 banks of the cores comprising the cluster (using a dynamic data mapping mechanism, as mentioned at the end of Section 3). The testing process proceeds at the granularity of each cluster, i.e., each cluster is viewed independently, and the testing policies are applied to the cores of each cluster. Due to the symmetrical nature of the clusters, the behavior/trends observed in all clusters are identical.

Table 3 presents the *per-core* Test-time Overhead (TO) results, in terms of the number of elapsed cycles required to execute the test programs. Specifically, the table shows the minimum, maximum, and average numbers of cycles needed to run the test program on each of the cores of the system. For this experiment, we use the Fluidanimate benchmark and we consider serial execution of the test programs across all the cores of each cluster, or across all the cores of the entire CMP when clustering is not used. The first set of results is calculated over the cores of the entire system (i.e., without the use of clusters), whereby the test-program data is distributed across all the CMP cores. The second set of results is calculated using the clustering approach, i.e., when using 4 16-core clusters (the values are averaged over the four clusters). The results in Table 3 show a significant reduction in the test-time overhead when using clustering. In fact, without clustering, the execution time of the test program on a single core in a 64-core CMP incurs an extra overhead of about 73%, as compared to a 16-core system. On the contrary, when using the clustering approach, this overhead is significantly reduced to around 20%. Note that part of the incurred overhead is due to the smaller-sized L2 bank (slice) per core in the 64-core CMP (see Table 1). Hence, the clustering approach allows us to contain the TO and scale the investigated test-scheduling policies to arbitrarily large CMP systems.

Figure 5 presents an overview of the savings obtained when using the clustering approach. The y-axis shows the *percentage reduction in the test-time overhead* when using clustering, as compared to the case *without* clustering. The different bars on the x-axis correspond to the different PARSEC benchmarks that were running concurrently with the testing process. Similar to our previous experiments, we evaluate two different LLC associativity setups: 4-way and 8-way. As demonstrated by the results in Figure 5, the clustering approach yields substantial improvements in terms of test-time overhead. In particular, the majority of the benchmarks experience a significant reduction in test-time overhead in both configurations (4-way and 8-way). The only benchmark that is negatively affected by the clustering approach is Canneal. As already demonstrated in the 16-core CMP results (Figure 3), this behavior is due to the high demands of the benchmark in terms of memory usage and cache accesses.

Under the 8-way LLC setup, the average savings across all examined benchmarks are in excess of 20%. In general, the clustering approach seems to almost eliminate the NoC overhead incurred when testing larger CMPs.

6 Conclusion

This article performs an exploration of periodic, online SBST scheduling policies in homogeneous multi-core systems. The ultimate goal is to reduce the testing overhead, in terms of testing time and test latency. Toward this end, we propose and examine several test scheduling techniques, based on the number of cores concurrently under test during test sessions. Given a constraint in test latency, the proposed methodology optimizes the test scheduling process, so as to minimize the test-time overhead and maximize system availability.

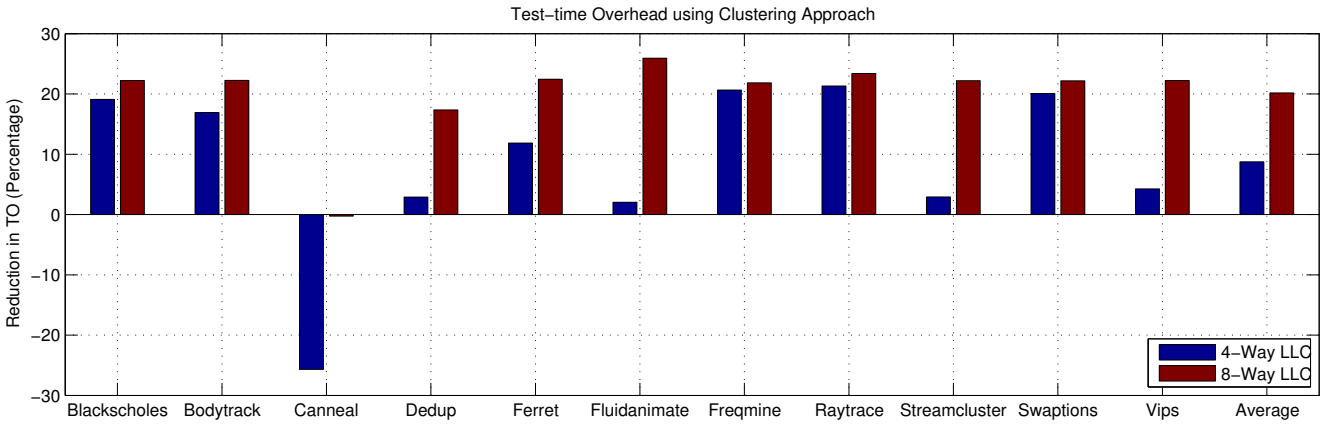

Fig. 5 An overview of the savings obtained when using the clustering approach. The graph shows the percentage reduction in the test-time overhead when using clustering, as compared to the case *without* clustering. Two different LLC associativity setups are evaluated: 4-way and 8-way.

Beyond the exploration pertaining to the number of concurrent cores under test, we also investigate the scalability of the test-scheduling policies as the CMP system grows in size, i.e., it accommodates larger numbers of on-chip cores. In order to curtail exponential increases in testing overhead in such large systems, this work proposes a clustering approach, whereby the CMP's cores are grouped into contiguous clusters. The underlying premise is to enable all test-related data to be resident in the LLC banks of the cores in the vicinity of the core-under-test, rather than being scattered throughout the CMP. The evaluation results indicate that clustering reaps substantial savings in test-time overhead, and it enables efficient scalability of the testing process to arbitrarily large systems.

References

1. Bienia C, Kumar S, Singh JP, Li K (2008) The parsec benchmark suite: Characterization and architectural implications. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT), ACM, New York, NY, USA, PACT '08, pp 72–81
2. Strong AW, Wu EY, Vollertsen RP, Sune J, Rosa GL, Sullivan TD (2006) Reliability Wearout Mechanisms in Advanced CMOS Technologies
3. Nicolaidis M, Zorian Y (1998) On-line testing for vlsi - a compendium of approaches. Journal of Electronic Testing 12(1-2):7–20
4. Constantinides K, Mutlu O, Austin T, Bertacco V (2009) A flexible software-based framework for online detection of hardware defects. IEEE Trans on Computers 58(8):1063–1079
5. Khan O, Kundu S (2011) Hardware/software codesign architecture for online testing in chip multiprocessors. IEEE Trans on Dependable and Secure Computing 8(5):714 –727
6. Rodrigues R, Kundu S (2013) A low power architecture for online detection of execution errors in smt processors. In: Proc. of the IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nano. Systems (DFTS), pp 33–38
7. Bayraktaroglu I, Hunt J, Watkins D (2006) Cache resident functional microprocessor testing: Avoiding high speed io issues. In: Proc. of the IEEE International Test Conference (ITC), pp 1–7
8. Psarakis M, Gizopoulos D, Sanchez E, Reorda M (2010) Microprocessor software-based self-testing. IEEE Design Test of Computers 27(3):4–19
9. Gizopoulos D, Psarakis M, Hatzimihail M, Maniatakos M, Paschalis A, Raghunathan A, Ravi S (2008) Systematic software-based self-test for pipelined processors. IEEE Trans on VLSI Systems 16(11):1441 –1453
10. Gurumurthy S, Vasudevan S, Abraham J (2006) Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. In: Proc. of the IEEE International Test Conference (ITC), pp 1–9
11. Parvathala P, Maneparambil K, Lindsay W (2002) Frits - a microprocessor functional bist method. In: Proc. of the IEEE International Test Conference (ITC), pp 590–598
12. Gaudesi M, Pomeranz I, Reorda MS, Squillero G (2017) New techniques to reduce the execution time of functional test programs. IEEE Trans on Computers 66(7):1268–1273
13. Lin CW, Chen CH (2017) A processor and cache online self-testing methodology for os-managed platform. IEEE Trans on VLSI Systems PP(99):1–14

14. Touati A, Bosio A, Girard P, Virazel A, Bernardi P, Sonza Reorda M (2017) Microprocessor testing: Functional meets structural test. Journal of Circuits, Systems and Computers 26(08)
15. Bernardi P, Cantoro R, Luca SD, Sanchez E, Sansonetti A (2016) Development flow for on-line core self-test of automotive microcontrollers. IEEE Trans on Computers 65(3):744–754
16. Bernardi P, Cantoro R, Ciganda L, Sanchez E, Reorda MS, Luca SD, Meregalli R, Sansonetti A (2014) On the in-field functional testing of decode units in pipelined risc processors. In: Proc. of the IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nano. Systems (DFTS), pp 299–304
17. Paschalis A, Gizopoulos D (2005) Effective software-based self-test strategies for on-line periodic testing of embedded processors. IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems 24(1):88–99
18. Cantoro R, Piumatti D, Bernardi P, Luca SD, Sansonetti A (2016) In-field functional test programs development flow for embedded fpus. In: Proc. of the IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nano. Systems (DFTS), pp 107–110
19. An G, Cantoro R, Sanchez E, Reorda MS (2017) On the detection of board delay faults through the execution of functional programs. In: Proc. of the IEEE Latin American Test Symposium (LATS), pp 1–6
20. Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6):10–16
21. Borkar S (2007) Thousand core chips: A technology perspective. In: Proc. of the 44th Annual Design Automation Conference (DAC), ACM, New York, NY, USA, DAC '07, pp 746–749
22. Apostolakis A, Gizopoulos D, Psarakis M, Paschalis A (2009) Software-based self-testing of symmetric shared-memory multiprocessors. IEEE Trans on Computers 58(12):1682–1694
23. Foutris N, Psarakis M, Gizopoulos D, Apostolakis A, Vera X, Gonzalez A (2010) Mt-sbst: Self-test optimization in multithreaded multicore architectures. In: Proc. of the IEEE International Test Conference (ITC), pp 1 –10
24. Kamran A, Navabi Z (2016) Stochastic testing of processing cores in a many-core architecture. Integration, the VLSI Journal 55:183–193
25. Li Y, Makar S, Mitra S (2008) Casp: Concurrent autonomous chip self-test using stored test patterns. In: Proc. of the Design, Automation and Test in Europe (DATE), pp 885–890
26. Li Y, Mutlu O, Mitra S (2009) Operating system scheduling for efficient online self-test in robust systems. In: Proc. of the IEEE International Conference on Computer-Aided Design (ICCAD), ACM, New York, NY, USA, ICCAD '09, pp 201–208
27. Haghbayan MH, Rahmani AM, Miele A, Fattah M, Plosila J, Liljeberg P, Tenhunen H (2016) A power-aware approach for online test scheduling in many-core architectures. IEEE Trans on Computers 65(3):730–743
28. Skitsas MA, Nicopoulos CA, Michael MK (2016) Daemonguard: Enabling o/s-orchestrated fine-grained software-based selective-testing in multi-/many-core microprocessors. IEEE Trans on Computers 65(5):1453–1466
29. Kaliorakis M, Psarakis M, Foutris N, Gizopoulos D (2014) Accelerated online error detection in many-core microprocessor architectures. In: Proc. of the IEEE VLSI Test Symposium (VTS), pp 1–6
30. Kamran A, Navabi Z (2015) Hardware acceleration of online error detection in many-core processors. Canadian Journal of Electrical and Computer Engineering 38(2):143–153
31. Kamran A, Navabi Z (2016) Self-healing many-core architecture: Analysis and evaluation. Hindawi Publishing Corporation, VLSI Design 2016
32. Awasthi M, Sudan K, Balasubramonian R, Carter J (2009) Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. Proc of the IEEE Int Symp on High Performance Computer Architecture (HPCA) 00:250–261
33. Sudan K, Chatterjee N, Nellans D, Awasthi M, Balasubramonian R, Davis A (2010) Micro-pages: Increasing dram efficiency with locality-aware data placement. In: Proc. of the ASPLOS on Architectural Support for Programming Languages and Operating Systems, pp 219–230
34. Magnusson P, Christensson M, Eskilson J, Forsgren D, Hallberg G, Hogberg J, Larsson F, Moestedt A, Werner B (2002) Simics: A full system simulation platform. IEEE Computer 35(2):50–58
35. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Computer Architure News 33(4):92–99
36. Agarwal N, Krishna T, Peh LS, Jha N (2009) Garnet: A detailed on-chip network model inside a full-system simulator. In: Proc. of the IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS), pp 33 –42