Operating System Scheduling for Efficient Online Self-Test in Robust Systems

Yanjing Li
Stanford University

Onur Mutlu
Carnegie Mellon University

Subhasish Mitra
Stanford University

ABSTRACT

Very thorough online self-test is essential for overcoming major reliability challenges such as early-life failures and transistor aging in advanced technologies. This paper demonstrates the need for operating system (OS) support to efficiently orchestrate online self-test in future robust systems. Experimental data from an actual dual quad-core system demonstrate that, without software support, online self-test can significantly degrade performance of soft real-time and computation-intensive applications (by up to 190%), and can result in perceptible delays for interactive applications. To mitigate these problems, we develop OS scheduling techniques that are aware of online self-test, and schedule/migrate tasks in multi-core systems by taking into account the unavailability of one or more cores undergoing online self-test. These techniques eliminate any performance degradation and perceptible delays in soft real-time and interactive applications (otherwise introduced by online self-test), and significantly reduce the impact of online self-test on the performance of computation-intensive applications. Our techniques require minor modifications to existing OS schedulers, thereby enabling practical and efficient online self-test in real systems.

1. INTRODUCTION

Almost every system (except high-end mainframes and safety-critical systems) designed today assumes that the underlying hardware is always correct. With smaller process geometries, such an assumption is either infeasible or may result in conservative designs that are very expensive [Borkar 05, Van Horn 05]. Hence, there is a clear need for efficient robust systems with built-in self-healing. Classical duplication and Triple Modular Redundancy (TMR) are expensive and inadequate.

Concurrent autonomous self-test, also called online self-test, is essential for robust systems with built-in self-healing capability. *Online self-test* enables a system to (periodically) test itself concurrently with normal operation. Existing online self-test techniques include logic built-in self-test (Logic BIST) [Bardell 87, Parvathala 02, Shen 98], concurrent autonomous chip self-test using stored test patterns (CASP) [Li 08a], virtualization assisted concurrent autonomous self-test (VAST) [Inoue 08], and access control extension (ACE) [Constantinides 07]. Some major applications of on-line self-test for robust system design include: 1. **Circuit failure prediction** for predicting the occurrence of a circuit failure before errors actually appear in system data and states. Circuit failure prediction is ideal for early-life failures (infant mortality) and aging because of the gradual nature of degradation [Agarwal 07, 08, Agostinelli 05, Chen 08, 09, Sylvester 06]; 2. **Hard failure detection and self-repair**. Major online self-test challenges are:

ICCAD'09, November 2–5, 2009, San Jose, CA, USA.

1. Test Thoroughness: Effective online self-test requires extremely thorough tests for a rich variety of test metrics – not only stuck-at tests, but also various flavors of delay tests. There are several alternatives to achieve this objective: (a) Logic BIST assisted by techniques to enhance test coverage, e.g., test points, mapping logic, and reseeding [Al-Yamani 03, Gherman 04, Lai 05, Nakao 99, Touba 95]; (b) Off-chip non-volatile storage of highly-compressed test patterns, both structural and functional, with very high test coverage [Constantinides 07, Li 08a].

2. Possible System Downtime: Proliferation of multi-/many-core systems enables online self-test to be invoked on one or more cores in the system, concurrently with normal operation of other cores, without visible system downtime to the end user [Constantinides 07, Li 08a, Inoue 08]. Special hardware structures, such as those compliant with the IEEE 1500 [IEEE 05], are necessary to isolate signals produced in test mode (already supported by some existing SOCs [McLaurin 02]).

3. Impact on System Performance: Online self-test imposes the following constraints on system design:

(a) Most online self-test techniques use scan chains for high test coverage. Applications cannot be executed simultaneously, together with online self-test, on the same piece of hardware.

(b) Since online self-test is directly related to overall system reliability (and, hence, correct functioning of the system), it cannot be optional, and must have higher priority than all other applications.

(c) Online self-test may be executed very frequently. For example, for "predicting" early-life failure problems (e.g., gate-oxide) [Carulli 06, Chen 08, 09], it may be necessary to execute online self-test once every 10 seconds to every minute. For hard failure detection, tests may be executed even more frequently – e.g., once every second – to allow low-cost and low-latency recovery techniques such as checkpointing [Constantinides 06].

(d) The execution of online self-test may last for a long time. For example, "prediction" of aging-related problems [Agarwal 07, 08, Sylvester 06] may involve the execution of online self-test once a day. However, we need extremely thorough test patterns that specifically target aging [Baba 09], and these tests may take several seconds to run. Although it is possible to utilize idle intervals in the system for test execution, there is no guarantee that critical or interactive tasks will not be initiated during online self-test.

If tasks on a core being tested are suspended during online self-test, there can be significant impact on the overall application performance, especially for soft real-time applications (e.g., video/audio playbacks), and interactive applications (e.g., text editors). This is exactly the case with hardware-only online self-test, where self-test is implemented entirely in hardware, and none of the higher system layers, e.g., operating system (OS), virtual machine monitor, or firmware, have knowledge about online self-test running in the system. Due to this lack of coordination, the hardware-only approach stalls all tasks on the corresponding processor core for the entire duration of self-test, which may take up to a few seconds depending on the implementation.

For example, to quantify the performance impact of hardware-only online self-test, we measured the response time (i.e. the amount of time between when a user initiates a mouse/keyboard

action and when the user observes the corresponding response) of three common interactive applications: vi (a text editor), evince (a document viewer), and firefox (a web browser). The experiments were performed on a dual quad-core Xeon system, where each core runs online self-test for 1 second every 10 seconds. Previous work indicates that common user-perceptible thresholds are within 100ms-200ms, and the longest response time cannot exceed 500ms for most applications [Chen 07]. Our experiment results show that hardware-only online self-test creates unacceptable perceptible delays (i.e. response times greater than 500ms) 10%-15% of the time (Sec. 5.3). Hardware-only online self-test also creates significant performance degradation in computation-intensive and soft real-time applications (Sec. 5.2 & 5.4). Hence, hardware-only online self-test may not be suitable for future robust systems.

The premise of this paper is that hardware-only online self-test is inadequate, and system-level support is required to orchestrate efficient online self-test. In this paper, we present OS scheduling techniques for efficient online self-test, called *online self-test-aware scheduling*: Given a set of online self-test requirements (e.g., frequency of invocation and duration), the OS scheduler schedules tasks around online self-tests by task migration in a multi-core system to minimize overall application performance impact. We focus on the mainstream homogeneous multi-core systems [Borkar 07] that may or may not be equipped with spare cores. We focus on processor cores, since fault-tolerant routing and BIST techniques exist for on-chip interconnects [Azumi 07, Grecu 06], and memory BIST and error correction are routinely used for memories. Major contributions of this paper are:

- We establish system performance implications of online self-test for multi-core systems and demonstrate the inadequacy of the hardware-only approach.
- We present online self-test-aware OS scheduling techniques, and evaluate them on a real Linux-based multi-core system for a wide range of applications (computation-intensive, interactive, and soft real-time).
- We show that online self-test-aware scheduling techniques eliminate any visible performance degradation for interactive and high-priority soft real-time applications.
- We show that online self-test-ware scheduling techniques can also reduce the performance impact of online self-test on computation-intensive applications. When each core runs online self-test for 1 second every 10 seconds, the performance impact of online self-test on the PARSEC benchmark suite is reduced from 20.58% to 0.48% on average when the system is not fully utilized, and from 41.73% to 13.51% on average when the system is fully utilized.

Section 2 presents the interactions between the OS and the online self-test process. Section 3 reviews OS concepts and online self-test terminology. Section 4 explains the main ideas behind online self-test-aware OS scheduling techniques. We present experimental results in Sec. 5, followed by related work in Sec. 6, and conclusions in Sec. 7.

2. ONLINE SELF-TEST AND OS INTERACTION

In this section, we demonstrate how the OS is involved in the online self-test process. Online self-test can be supported by a simple on-chip hardware component (e.g., 0.01% of the total chip area for OpenSPARC T1 [Li 08a]), called the test controller. The OS scheduler interacts with the test controller to perform online self-test-aware scheduling, in the steps discussed below:

1. The test controller selects a core in a many-/multi-core system, in a round-robin fashion, to undergo online self-test.
2. The test controller interrupts the OS scheduler running on the core selected for test, and informs it that online self-test is scheduled.
3. The OS scheduler first performs online self-test-aware scheduling (discussed in Sec. 4), and then switches in a dummy thread on the core selected for test. Finally, it informs the test scheduler that online self-test-aware scheduling is completed using a memory-mapped register.
4. The test controller carries out necessary actions to test the core, and analyzes test results. If the test fails, the test controller initiates recovery actions (e.g., rollback to the previous checkpoint). Otherwise, the test controller interrupts the OS scheduler again, and informs it that the test completes.
5. The OS scheduler switches out the dummy thread, and schedules other useful tasks on the core that has just been tested.

3. OS CONCEPTS AND TEST TERMINOLOGY

3.1 OS Scheduling Basics

Prevalent OS scheduling techniques are priority-based [Silberschatz 05]. The scheduler ranks all tasks by their priorities, and chooses the task with the highest priority to run during context switch. For multi-core systems, each core has its own list of tasks, stored in a per-core data structure called the *run queue*. Scheduling decisions are made independently on each core. The priority of each task consists of two components: static and dynamic. The *static component* is initialized to a default value by the scheduler when a task is created. The *dynamic component* is determined by the recent CPU usage of a task, and it decreases proportionally to the time during which the task runs on a core to ensure fairness [Silberschatz 05].

For OSes that support multi-core architectures, the scheduler not only schedules tasks on each core, but also balances workload among different cores. *Load balancing* determines whether tasks should be migrated from one core to another to achieve balance, and provides a mechanism for task migration. After a task is migrated, its priority is adjusted properly with respect to the tasks on the destination core.

3.2 Runnable vs. Sleeping Tasks

A task can be in different states at different times. For scheduling purposes, we focus on two states: runnable and sleeping (blocked). If a task is runnable, it is placed in a run queue (selection of the run queue is determined by the scheduler), waiting to be selected by the scheduler for running. A task is sleeping (or blocked) if it is waiting for I/O or other hardware (e.g., timer) events. A sleeping task is dequeued from the run queue and placed in another data structure, a *wait queue*, until the corresponding event "wakes" it up.

Tasks can be classified into two categories, interactive and non-interactive, based on the amount of time they spend in the runnable and sleeping states. *Non-interactive* tasks spend most of their time utilizing processor cores (i.e., runnable for most of the time). *Interactive* tasks (e.g., text editors, web browsers) accept and respond to user inputs. It is typical for interactive tasks to spend most of their time in wait queues, waiting for user inputs (e.g., keyboard or mouse interrupts). Once woken up, they need to quickly respond to user requests.

3.3 Online Self-Test Terminology

Test period (*TP*) is the amount of time from the beginning of a self-test on a core to the beginning of the next self-test on the same core. *Test latency* (*TL*) is the duration of an online self-test. TP and TL are set based on the reliability requirements (details in Sec. 5), and are specified to the OS scheduler using global variables. A core undergoing online self-test test is a *core-under-*

test, a core not being tested and running tasks is an *available core*, and a core not being tested and not running tasks is an *idle core*.

4. ONLINE SELF-TEST-AWARE OS SCHEDULER

Our online self-test-aware OS scheduling techniques build on top of existing OS schedulers and utilize their priority-based scheduling mechanisms and task migration capabilities required by load balancing. Online self-test-aware scheduling is done separately for three cases: tasks on run queues and tasks on wait queues (discussed in Sec. 3.2), because run queues and wait queues are different data structures, and soft real-time tasks, which belong to a different scheduling class (Sec. 4.3). We have implemented our algorithms in Linux (2.6.25.9, June 2008). Implementation details can be found in [Li 08b]. These techniques are applicable to all OSes that support priority-based scheduling and task migration.

4.1 Online Self-Test-Aware Scheduling of Tasks in Run Queue(s) of Core(s)-Under-Test

We have developed three different online self-test-aware OS scheduling schemes for tasks in a run queue, discussed below.

4.1.1 Scheduling Scheme 1: migrate_all

The first scheme, *migrate_all*, migrates all runnable tasks from a core-under-test to the core that will be tested furthest in the future. Such a core can be identified if the ordering of cores running online self-test is fixed, or using a per-core variable that specifies the amount of remaining time until a core needs to run online self-test (e.g., in our implementation, it is the core with the largest *time_till_next_test*, a field stored in the run queue data structure). Task migration is performed right before the core undergoes online self-test.

4.1.2 Scheduling Scheme 2: load_balance_with_self_test

The second scheme, *load_balance_with_self_test*, models online self-test as a task with the highest priority. This special task, called the *test thread*, is switched in for the entire duration of online self-test on a core-under-test so that no other tasks can run on that core. In Linux, the highest priority task (and, thus, the test thread) contributes the most to the workload of a processor core – its workload is ~90 times the workload of tasks with the default priority. The load balancing mechanism moves tasks away from the core with the heaviest workload. Therefore, other tasks on a core-under-test are likely to be migrated to other available or idle cores.

With online self-test modeled as a task with the highest priority, the system workload can change abruptly as online self-test begins to run on one core after another. Therefore, we also modified the scheduler to bound the load-balancing interval to be the time between the start of two consecutive invocations of online self-tests.

4.1.3 Scheduling Scheme 3: migrate_smart

The third scheme, *migrate_smart*, migrates tasks based on a cost-benefit analysis carried out by the scheduler. There are two different migration opportunities: 1. If there exist one or more idle cores when a core is being tested, the idle core that will be tested furthest in the future will attempt to "pull" all tasks from the core-under-test (existence of idle cores implies that, most likely, there is only one task on the core-under-test, if the system is load-balanced), if the scheduler determines that it is beneficial to migrate based on the cost-benefit analysis (discussed below). 2. If all cores are busy running applications, our algorithm uses a distributed approach: During a context switch, the scheduler on each available core examines not only the tasks in its own run queue, but also those in the run queues of all cores-under-test (accessed through shared memory). The scheduler then compares the highest-priority tasks on these run queues, and chooses the highest-priority task. If this chosen task is from a core-under-test, it is only migrated if the cost-benefit analysis determines migration to be worthwhile. Fig. 1 depicts this process.

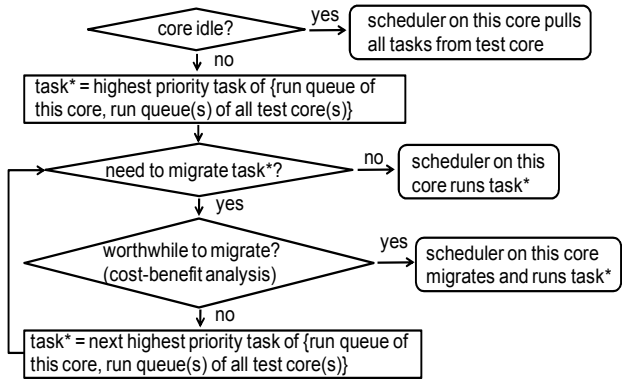

Figure 1. migrate_smart Algorithm for an Available Core

The purpose of the cost-benefit analysis is to determine whether it is worthwhile to migrate a task from a core-under-test to an available or idle core. *Cost* is defined as the amount of time the task needs to wait before it gets a chance to run (either on a core-under-test or a core that wants to pull this task). For an available/idle core attempting to migrate tasks from a core-under-test, the corresponding scheduler estimates the cost of pulling the task and the cost of keeping the task on the core-under-test, and chooses the action (migrate to the destination core or don't migrate) with the lower cost. Unnecessary migration overhead can be avoided because a task migrates only if the benefit of migration is strictly greater than that of keeping the task on the core-under-test. Note that, such a cost-benefit analysis can be used because the scheduler is aware of the latency of the test. Most general-purpose or real-time OS schedulers (e.g., in Linux or RTLinux) do not use a cost-benefit analysis because a task's runtime latency is usually unknown.

The cost for a task to wait on a core-under-test is the sum of the following two quantities:

1. Amount of time a task needs to wait until the test completes (in our implementation, this can be obtained by subtracting a per-core variable called *test_duration*, which indicates how long the test has been running, from the pre-specified test latency);

2. Sum of effective runtimes of all higher-priority tasks waiting on the run queue of the corresponding core-under-test. *Effective runtime* of a task is defined as the duration of time from when a task is switched in by the scheduler for execution till when it is switched out because it is preempted, sleeping, or yielding. In our implementation, we stored the last 5 effective runtimes of a task in a per-task data structure called *task_struct*. To estimate the total effective runtime of all tasks with higher priorities, the scheduler traverses tasks on the core-under-test and calculates the sum of the average of the 5 stored effective runtimes of tasks with higher priorities.

The cost of task migration is estimated as the sum of fhe following two quantities:

1. *Direct cost*, which includes the latency of the task migration code and the amount of time for which the task needs to wait in the run queue of the destination core after it is migrated. The latency of the task migration code is extremely small – in the order of hundreds of nanoseconds from our measurement. The time the task needs to wait in the run queue of the destination

core after it is migrated is estimated by the total effective runtime of all tasks that have higher priorities in the destination core.

2. *Indirect cost,* which is caused by cache effects. This cost is only associated with "cache hot" tasks, i.e., tasks that are already utilizing the caches. A task is identified as "cache hot" in Linux 2.6.25.9 if it has been running for more than 500,000ns without being switched out. Migrating a "cache hot" task could result in degraded performance due to compulsory misses in the destination core. This cost can be approximated by the amount of time required to fetch all cache lines utilized by the task being migrated. Cache line utilization can be estimated using the working set of an application, which can be obtained by offline runtime profiling (e.g., for benchmarking purposes). The working sets of the applications from the PARSEC benchmark suite range from 64KB to 2MB [Bienia 08]. The system we used for evaluation consists of two Xeon quad-core processors on two sockets, with cache characteristics shown in Table 1 [Intel 07]. The L2 cache is shared among the 4 cores on the same socket. Based on the PARSEC working set size, we estimate 100% utilization of the L1 data cache (i.e. 100% of L1 cache blocks are useful), and 16.7% (2MB/12MB) utilization of the L2 cache. The migration cost due to cache effects is: (total number of utilized L1 lines×L1 line refill latency) + (total number of utilized L2 lines × L2 line refill latency) = (32KB/64B)×100%×7.5ns +(12MB/64B)×16.7%×11ns (Table 1) ≈ 400,000ns, for a task to be migrated between two cores on different sockets. If a task is migrated between two cores belonging to the same socket, the L2 cache is shared, so the migration cost is the L1 cache penalty (32KB/64B)×100%×7.5ns≈4,000ns. This estimation is pessimistic. In reality, L1 cache utilization can be less than 100% [Qureshi 07]. We also assume that cache lines are fetched serially without advanced architectural data transfer techniques. Moreover, this model assumes that the indirect cost for all "cache hot" tasks is the same. The accuracy of the model can be improved by dynamically collecting cache characteristics using performance counters to estimate the indirect cost.

Table 1. Cache parameters

	Capacity	Line Size	Line Refill latency
L1 DCache	32KB	64B	7.5ns
L2 Cache	12MB	64B	11ns

4.2 Online Self-Test-Aware Scheduling of Tasks in Wait Queue(s) of Core(s)-Under-Test

As discussed in Sec. 3.2, interactive applications spend most of their time in wait queues. When a task is woken up from a wait queue, the scheduler needs to decide which run queue to place the task. The run queue selected by the unmodified Linux scheduler belongs to one of the following three cases: 1. run queue that holds the task before it sleeps; 2. run queue of the core that wakes up the task; 3. run queue of an idle core. With online self-test, however, the run queue selected by the original Linux scheduler could belong to a core-under-test. In this case, online self-test can cause significant performance impact on the task woken up, which is likely to be an interactive task. With only the algorithms discussed in Sec. 4.1, there can still be a delay (longer than the acceptable response time threshold, e.g., 500ms) before these tasks can run on cores not undergoing online self-test (or before online self-test completes), and user can experience unacceptable perceptible delays.

We overcome this problem by following the run queue selection of the original OS scheduler unless it picks one that belongs to a core-under-test. In this case, our algorithm changes the destination core to be the core that will be tested furthest in the future. Such a core can be identified as discussed in Sec. 4.1.1. After a task is enqueued to a run queue determined by this algorithm (discussed in this section), the algorithms in Sec. 4.1 are used for scheduling tasks in the run queues.

4.3 Online Self-Test Aware Scheduling of (Soft) Real-Time Tasks

In Linux, a different scheduling class is responsible for scheduling soft real-time (RT) applications. Soft RT applications perform tasks that are deadline-sensitive (i.e. tasks must be completed within a fixed amount of time), such as audio and video encoding/decoding. In Linux, soft RT tasks have higher priorities than other tasks, and cannot be preempted by lower-priority tasks. If a certain RT task is preempted (by a higher-priority RT task), the scheduler migrates the preempted task to a core with lower-priority tasks, if such a core exists. Such a policy allows the highest-priority RT tasks to meet their deadlines.

RT tasks are also interrupted and suspended by online self-test. We modify the migration policy so that the highest-priority RT tasks are most likely to meet their deadlines. When an RT task is interrupted by online self-test or preempted by a higher-priority task, it is migrated to a core that: 1. has lower priority tasks (RT and not RT) than itself, and 2. will be tested furthest in the future. The scheduler will select RT tasks for execution until there is no runnable RT task, at which point the algorithms in Sec. 4.1 & 4.2 are used to schedule non-RT tasks.

5. RESULTS AND COMPARISONS

5.1 Evaluation Environment

The system used for our evaluation consists of two 2.5GHz quad-core Intel Xeon E5420 processors [HP 08] running Linux 2.6.25.9. Since we do not have the RTL (e.g., Verilog code) for our evaluation platform, we cannot implement online self-test in hardware. Therefore, we model online self-test in the OS as a special *weightless test thread*. This thread is the same as the test thread discussed in Sec. 4.1.2, except that it does not contribute to the workload. It is a dummy thread that does no useful work. Online self-test requires test patterns to be transferred from off-chip non-volatile storage to on-chip scan flip-flops, which could result in contention in the system bus. However, bus utilization of online self-test in our evaluation system (with a 1333MHz front-side bus) is 0.014% [Li 08b], which is negligible. Thus, this model is sufficient for studying performance impact of online self-test. For a system with a slower bus of 35MHz, performance of memory-bound applications can be degraded by an additional 2% due to bus conflicts caused by online self-test [Inoue 08].

We present experiment results for three different test period and test latency combinations, summarized in Table 2. The test latency is proportional to the quality of the test: the longer the latency, the more thorough the test. Test latency of 1 sec is reasonable for today's systems [Li 08a]. Test latency of 5 sec may be necessary to allow application of more advanced tests such as path delay tests [Baba 09]. Test periods are chosen to reflect various applications of online self-test. For hard failure detection, short detection latency is important. This requires frequent self-tests, e.g., once every 10 seconds for each core in the system. However, if online self-test is mainly used for failure prediction, it may be sufficient to apply tests once every minute or even less frequently for each core in the system. Experimental results for TP=10 sec/TL=500ms, TP=10 sec/TL=5sec and TP=60 sec/TP=500ms are shown in [Li 08b], with similar conclusions as presented in this paper.

Table 2. Online self-test parameters

TP/TL	Test Quality	Test Usage
10 sec / 1 sec	High	Failure detection & prediction
60 sec / 1 sec	High	Failure prediction
60 sec / 5 sec	Extremely high	

We evaluated the three online self-test-aware scheduling algorithms (migrate_all, load_balance_with_self_test, and migrate_smart) presented in Sec. 4, and compared the results against the hardware-only approach, with the OS having no knowledge about online self-test running in the system (hardware_only). Hardware_only is implemented by switching in the weightless test thread without modifying the original Linux scheduling algorithms. However, we modified the kernel so that the task preempted by the test thread still "appears" to be running, since a hardware-only approach cannot actually change the OS data structures to indicate that the task is preempted.

The workloads we used in our experiments include: 1. Non-interactive and computation-intensive applications: PARSEC benchmark suite [Bienia 08]; 2. Interactive applications: vi, evince, and firefox (as mentioned in Sec. 1). Previous work on interactive applications used similar workloads [Flautner 00]; 3. A typical soft real-time application: H.264 encoder.

5.2 Non-Interactive Application Performance Results

We run the multi-threaded PARSEC benchmark suite with 4 threads and 8 threads to represent partial and full system utilization scenarios. Threads can run on any core in the dual quad-core machine. For the 8-thread configuration, since we disable hyper-threading, the 8 threads can fully utilize the system. We ran each application 5 times, and obtained execution latencies within 5% of the average for most applications. Results are presented as the average execution latency overhead over the baseline Linux kernel scheduler without invocations of online self-test. Figures 2-7 show the overheads of all applications, as well as the geometric mean of the overheads of all applications (average), for various TP and TL combinations.

The key observations from these results are:

1. There is significant performance degradation with hardware_only (e.g. up to 192.43% and 61.94% on average with TP=60 sec and TL=5 sec), for both the 8-thread and 4-thread configurations. Such high performance overheads are possible because, in multi-threaded applications, suspension of one thread due to online self-test can cause suspension of other threads if they are synchronized (for single-threaded applications, performance overhead of hardware_only is *TL/(TP-TL)* [Li 08b]). For example, if a thread holding a lock is suspended by online self-test, all other threads (on other cores) waiting for the same lock are delayed. The performance impact is smaller for smaller test latencies and/or larger test periods.

2. Our online self-test-aware schedulers significantly reduce the performance impact of online self-test over hardware_only. If the system is not fully utilized (4-thread), the performance impact is smaller than a few percentage points with online self-test-aware schedulers (i.e., smaller than 3% for most applications and a maximum of 7% with migrate_smart). For a fully loaded system (8-thread), e.g., for TP=10 sec/TL=1 sec, the performance impact of online self-test with our schedulers is 13.51% on average, compared to 41.73% with hardware_only. Performance degradation is inevitable in this case because hardware resources are limited. However, the performance overhead with our techniques is very close to *TL/(TP-TL)*, so there is almost no additional penalty due to synchronization. This is because our scheduling techniques ensure that all threads are making forward progress.

3. Migrate_smart outperforms load_balance_with_self_test and migrate_all for most of the cases, and it performs consistently well across different applications and test settings. In certain cases (e.g., TP=60s, TL=1s with migrate_smart), application performance with online self-test is even better than the baseline (without online self-test). This is an artifact of the different task migration policies in online self-test-aware schedulers, which result in changes in thread ordering that can reduce synchronization overhead. Consider a set of threads accessing a critical section protected by a lock. The critical section can take less time with our algorithms, because a thread with less work in the critical section can happen to acquire the lock first, which reduces the serialization of threads due to synchronization. Note that, the original goal of our algorithms is not to reduce synchronization overhead; it is a side effect.

4. On average, migrate_smart achieves only minor performance gains (≤5%) over the other two simpler online self-test-aware scheduling techniques. There are several reasons for this result. First, in a shared-memory system, the direct cost of task migration is very small – as discussed in Sec. 4.1.3, it is in the order of hundreds of nanoseconds. Second, for many applications, the indirect cost of task migration due to cache effects is significantly smaller than the test latency of 1 sec. As a result, it is almost always more beneficial for tasks to migrate than to stall on the core-under-test. Finally, one may think that migrate_all would perform poorly because it can overload the core that will be tested furthest in the future. However, we allow the Linux load-balancer to run on all available cores, and thus any imbalance among the cores will eventually be resolved. This result suggests that a simple algorithm such as migrate_all may be sufficient to sustain a certain performance level required by a system – the key is for the OS to be aware of online self-tests running in the system. Such a technique requires only minor modifications (<100 lines of code) to the scheduler, and is practical for real systems. Other OS scheduling techniques may also be useful as long as the OS is aware of online self-tests.

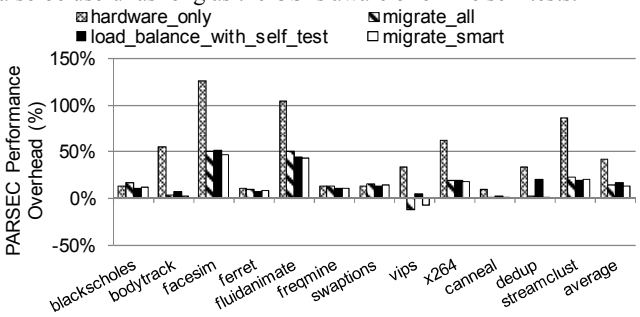

Figure 2. 8-thread overhead, TP=10sec, TL=1sec

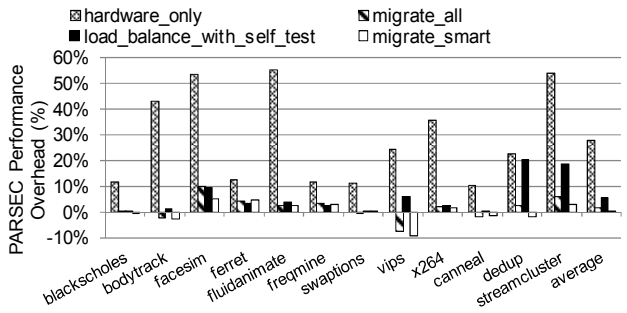

Figure 3. 4-thread overhead, TP=10sec, TL=1sec

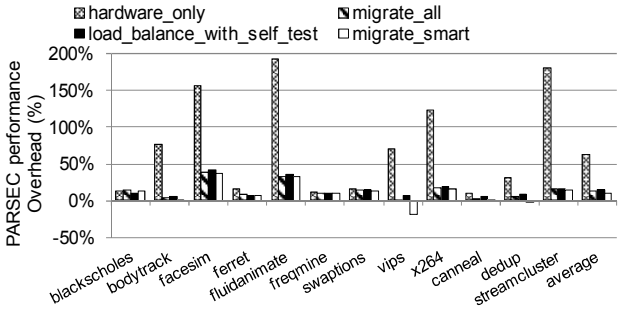

Figure 4. 8-thread overhead, TP=60sec, TL=5sec

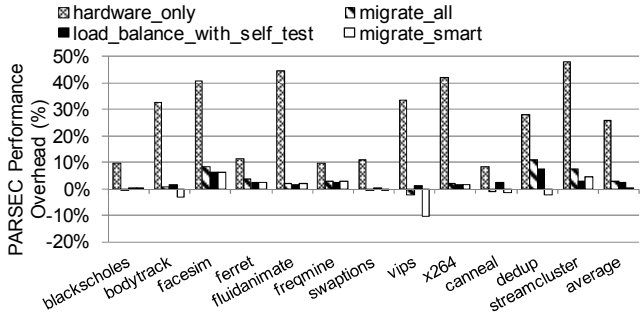

Figure 5. 4-thread overhead, TP=60sec, TL=5sec

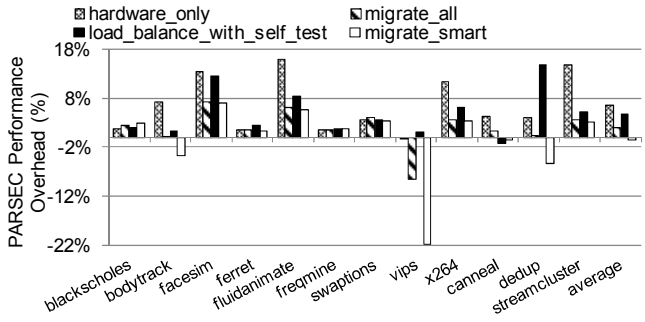

Figure 6. 8-thread overhead, TP=60sec, TL=1sec

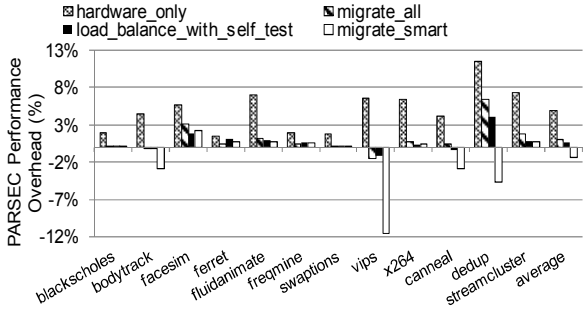

Figure 7. 4-thread overhead, TP=60sec, TL=1sec

5.3 Interactive Application Performance Results

We use response time (defined in Sec. 1) as the evaluation metric for interactive applications. We ran a total of 80 trials for each application: We typed 80 characters using vi, loaded 80 pdf pages using evince, and loaded 80 local webpages using firefox. More details of the methodology for measuring response time are described in [Li 08b]. These results were obtained with no other applications running. We also ran the same experiments with 8-thread PARSEC applications running in the background. With this more loaded system, performance degradation of interactive applications is more severe with hardware_only. However, our algorithms (load_balance_with_self_test, migrate_all, and migrate_smart) obtain the same results (i.e., no perceptible delays) as reported below [Li 08b].

For interactive applications, the criterion of vital importance is to keep all response times below human-perceptible thresholds. As discussed in Sec. 1, we classify response times below 200ms as delays that cannot be perceived by users, and above 500ms as unacceptable perceptible delays. Figures 8-10 show the response time results for TP=10 sec and TL=1 sec (similar results are obtained for the other TP and TL combinations [Li 08b]). The response times are expressed as an accumulated function. The values of the y-axis is the total percentage of experiment instances that achieved response times less than or equal to the corresponding response times in the x-axis.

The key observations from these results are:

1. For hardware_only, 10%-15% of all user interactions result in unacceptable perceptible delays (i.e., response times greater than 500ms).

2. Our online self-test-aware scheduling algorithms, migrate_all, load_balance_with_self_test, and migrate_smart, guarantee placement of interactive tasks on an available or idle core (Sec. 4.2), effectively eliminating the perceptible performance impact: None of the applications result in any visible delays (all response times are smaller than 200ms).

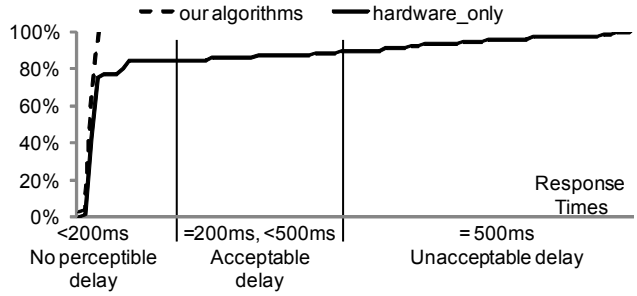

Figure 8. vi Response Times, TP=10sec, TL=1sec

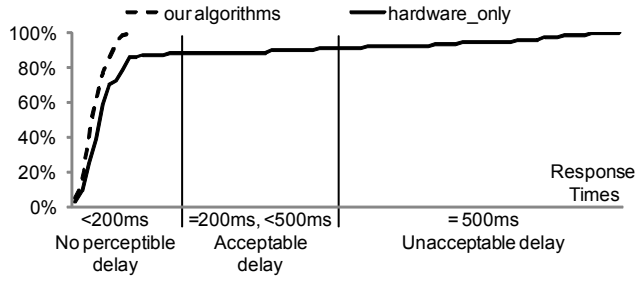

Figure 9. evince Response Times, TP=10sec, TL=1sec

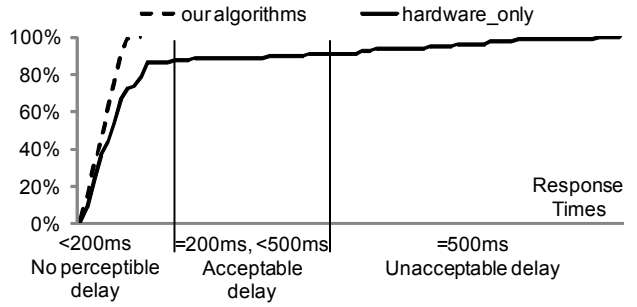

Figure 10. Firefox Response Times, TP=10sec, TL=1sec

5.4 Soft Real-Time Application Performance Results

To evaluate online self-test-aware scheduling with the real-time scheduling class (Sec. 4.3), we ran the x264 application in

the PARSEC benchmark with online self-test characterized by TP=10s and TL=1s (similar results are obtained for the other TP and TL combinations). x264 is an H.264 video encoder, which is a typical soft real-time application. We first run 7 instances of the x264 application configured with the highest real-time priority (level 99), not fully utilizing the dual quad-core system. Next, we run 7 instances of the x264 program with the highest real-time priority (level 99), and 1 instance of the same program with a lower real-time priority (level 98), overloading the system. Table 3 shows the execution latency overheads over the baseline configuration with no invocations of online self-test.

The key points from these results are:

1. Hardware_only results in approximately 11% (i.e., *TL/(TP-TL)*) performance degradation for all instances of the application, regardless of whether or not the system is fully utilized. Due to the performance degradation, the soft RT tasks can miss their deadlines.

2. With all of the online self-test-aware algorithms (migrate_all, load_balance_with_self_test, and migrate_smart), there is no performance degradation (-0.03%) when the system is not fully utilized, so all RT tasks meet their deadlines. When the system is fully utilized, there is no performance degradation of the highest-priority RT applications, but performance degrades for the lower-priority one. This is expected, because when the system is overloaded, the highest priority tasks always get all available resources for the best chance of meeting their deadlines (achieved with algorithm presented in Sec. 4.3). Performance degradation of the lower priority task is inevitable due to limited processing resources. We also ran 4 RT tasks with 4 other non-RT tasks, and these experiments show, with our algorithms, that there is negligible performance impact (~2%) on the RT tasks due to online self-test [Li 08b].

Table 3. Execution latency overheads of x264 with real-time scheduling policy

Configuration	Hardware_only	Our algorithms
System not fully loaded	11% (average) for all 7 applications	no penalty for all 7 applications
System fully loaded	11% (average) for all 8 applications	0% for the 7 higher-priority applications; 87% for the 1 lower-priority application

6. RELATED WORK

Online self-tests create temporal performance heterogeneity in a system. For example, a core being tested can be considered as a slow core, and a core to be tested soon can be considered less powerful. Previous work has investigated scheduling techniques for static heterogeneous systems [Balakrishnan 05, Kumar 03, Li 07]. These approaches are not adequate for systems with online self-test because: 1. they target a static heterogeneous system and do not take into account temporal unavailability of a core due to online self-test; and 2. they assume that cores inherently have different performance characteristics whereas our algorithms are designed for cores with homogeneous performance characteristics.

Existing scheduling techniques that assist power management can dynamically change the number of active processor cores to trade off power and performance [Li 06]. A processor core can also operate in a "low power" mode where the clock frequency is greatly reduced [Isci 06, Kadayif 04, Teodorescu 08, Zhang 02, Zhuo 05]. Task scheduling [Stavrou 06] and migration [Heo 03] techniques are proposed for power efficiency reasons. In embedded/real-time systems, power management through task scheduling is a popular research topic [Liu 01, Luo 01, Qiu 01, Quan 01, Zhang 05]. Scheduling techniques addressing both reliability requirements and energy consumption in real-time systems are also investigated [Zhang 04, Sridharan 08]. However, none of these previous techniques directly apply to scheduling with online self-test because they assume that all cores in the system are always available.

If a real-time OS (e.g., eCos [eCos 09], RTLinux [RTLinux 09]) is used, online self-test can be treated as a periodic real-time task. However, mainstream general-purpose systems do not employ real-time OSes, which are only specialized for real-time tasks.

7. CONCLUSIONS

System support is necessary for efficient orchestration of online self-test in robust systems. Otherwise, application-level performance degradation can be significant, especially for interactive and soft real-time applications. As demonstrated using extensive experiments on an actual system, our online self-test-aware OS scheduling algorithms successfully minimize such performance impact while preserving online self-test schedules, task priorities, expected quality of service for interactive and soft-real time applications, and system load balance. Moreover, our techniques eliminate any negative impact of hardware-only online self-test that can result in perceptible delays on real-time and interactive applications. For all of these algorithms, awareness of online self-test is the key – with such awareness, even a simple algorithm that migrates all tasks away from the core can provide great performance benefits. Our algorithms require only minor modifications to existing OS schedulers. Hence, online self-test-aware techniques can enable cost-effective integration of online self-test when deployed in real systems (e.g., desktops, servers, and embedded systems) where performance is a key system design target. Future directions of this research include: Extension of our techniques to heterogeneous multi-core systems, minimization of power-implications of online self-test, and online self-test of non-processor components.

REFERENCES

[Agarwal 07] Agarwal, M., *et al.*, "Circuit Failure Prediction and Its Application to Transistor Aging," *VTS*, 2007.

[Agarwal 08] Agarwal, M., *et al.*, "Optimized Circuit Failure Prediction for Aging: Practicality and Promise," *ITC*, 2008.

[Agostinelli 05] Agostinelli, M., *et al.*, "Random charge effects for PMOS NBTI in ultra-small gate area devices," *IRPS,* 2005.

[Al-Yamani 03] Al-Yamani, A.A., *et al.*, "BIST Reseeding with Very Few Seeds," *VTS*, 2003.

[Azumi 07] Azumi, M., *et al.*, "Integration Challenges and Tradeoffs for Tera-Scale Architectures," *Intel Technical Journal*, 2007.

[Baba 09] Baba, H., and S. Mitra, "Testing for Transistor Aging," *VTS*, 2009.

[Balakrishnan 05] S. Balakrishnan *et al,*, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," *ISCA*, 2005.

[Bardell 87] Bardell, P. H., *et al.*, "Built-In Test for VLSI: Pseudorandom Techniques," *Wiley*, 1987.

[Bienia 08] Bienia, C., *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *PACT*, 2008,

[Borkar 05] Borkar, S., "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *MICRO*, 2005.

[Borkar 07] Borkar, S., "Thousand Core Chips – A Technology Perspective," *DAC*, 2007.

[Carulli 06] Carulli Jr, J.M., and T.J. Anderson, "The impact of multiple failure modes on estimating product field reliability," *IEEE Design & Test*, 2006.

[Chen 07] Chen, J., and J. Thropp, "Review of Low Frame Rate Effects on Human Performance," *IEEE Trans. on Systems, Man, and Cybernetics*, 2007.

[Chen 08] Chen, T.W., *et al.*, "Gate-Oxide Early Life Failure Prediction," *VTS*, 2008.

[Chen 09] Chen, T.W., Y.M. Kim, K. Kim, Y. Kameda, M. Mizuno and S. Mitra, "Experimental Study of Gate-Oxide Early Life Failures," *IRPS*, 2009.

[Constantinides 06] Constantinides, K., *et al.,* "Ultra Low-Cost Defect Protection for Microprocessor Pipelines," *ASPLOS*, 2006.

[Constantinides 07] Constantinides, K., *et al.,* "Software-Based On-line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation," *MICRO*, 2007.

[eCos 09] http://ecos.sourceware.org/.

[Flautner 00] Flautner, K., *et al.*, "Thread-level Parallelism and Interactive Performance of Desktop Applications," *ASPLOS*, 2000.

[Gherman 04] Gherman, V., *et al.*, "Efficient Pattern Mapping for Deterministic Logic BIST," *ITC,* 2004.

[Grecu 06] Grecu, C. *et al.*, "BIST for Network-on-Chip Interconnect Infrastructures," *VTS,* 2006.

[Heo 03] Heo, S., *et al.*, "Reducing Power Density through Activity migration," *Intl. Symp. on Quality Electronic Design,* 2003.

[HP 08] "HP xw8600 Workstation Specification," http://h10010.www1.hp.com/ wwpc/us/en/sm/WF06a/12454-12454-296719-307907-296721-3432827.html.

[IEEE 05] "IEEE 1500 Standard for Embedded Core Test," http://grouper.ieee.org/groups/1500/index.html, 2005.

[Inoue 08] Inoue, H., *et al.*, "VAST: Virtualization Assisted Concurrent Autonomous Self-Test," *ITC*, 2008.

[Intel 07] "Intel 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/design/processor/ manuals/248966.pdf, 2007.

[Isci 06] Isci, C., *et al.*, "An Analysis of Efficient Multi-Core Global Power management Policies: Maximizing Performance for a Given Power Budget," *MICRO*, 2006.

[Lai 05] Lai, L., *et al.*, "Hardware Efficient LBIST with Complementary Weights," *ICCD*, 2005.

[Li 06] Li, J., and J. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors", *HPCA,* 2006.

[Li 07] Li, T., *et al.*, Efficient Operating System Scheduling for Performance-Asymmetric multi-Core Architectures," *Supercomputing*, 2007.

[Li 08a] Li, Y., *et al.*, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns," *DATE*, 2008.

[Li 08b] Li, Y., O. Mutlu, and S. Mitra, "Operating System Scheduling for Efficient Online Self-Test in Robust Systems (Technical Report)," 2008. Please contact the authors for access.

[Liu 01] Liu, J., *et al.*, "Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded Systems," *DAC*, 2001.

[Luo 01] Luo, J., and N.K., Jha, "Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems," *DAC*, 2001.

[Kadayif 04] Kadayif, I., *et al.*, "Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption in Chip Multiprocessors," *DATE,* 2004.

[Kumar 03] R. Kumar, *et. al*, "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance," *MICRO*, 2003.

[McLaurin 02] McLaurin, T., and S. Ghosh, "ETM Incorporates Hardware Segment of IEEE P1500," *IEEE Design & Test,* 2002.

[Nakao 99] Nakao, M., *et al.*, "Low Overhead Test Point Insertion for Scan-Based BIST," *ITC*, 1999.

[Parvathala 02] Parvathala, P., *et al.*, "FRITS: A Microprocessor Functional BIST Method," *ITC*, 2002.

[Qiu 01] Qiu, Q., *et al.*, "Dynamic Power Management in a Mobile Multimedia System with Guaranteed Quality-of-Service," *DAC*, 2001.

[Quan 01] Quan, G., and X. Hu, "Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors," *DAC*, 2001.

[Qureshi 07] Qureshi, M., *et al.*, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," *HPCA*, 2007.

[RTLinux 09] http://www.rtlinuxfree.com/.

[Shen 98] Shen, J., and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation," *ITC,* 1998.

[Silberschatz 05] Silberschatz, A., "Operating System Concepts (7th edition)", J*ohn Wiley & Sons Inc.*, 2005.

[Sridharan 08] Sridharan, R., *et al.*, "Feedback-Controlled Reliability-Aware Power Management for Real-Time Embedded Systems," *DAC*, 2008.

[Stavrou 06] Stavrou, K., and P. Trancoso, "Thermal-Aware Scheduling: A Solution for Future Chip Multiprocessors' Thermal Problems," *EUROMICRO*, 2006.

[Sylvester 06] Sylvester, D., *et al.*, "ElastIC: An Adaptive Self-Healing Architecture for Unpredictable Silicon," *IEEE Design & Test,* 2006.

[Teodorescu 08] Teodorescu, R., and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," *ISCA,* 2008.

[Touba 95] Touba, N.A., and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *ITC*, 1995.

[Van Horn 05] Van Horn, J., "Towards Achieving Relentless Reliability Gains in a Server Marketplace of Teraflops, Laptops, Kilowatts, & Cost, Cost, Cost," *ITC,* 2005.

[Zhang 02] Zhang, Y., *et al.*, "Task Scheduling and Voltage Selection for Energy Minimization", *DAC,* 2002.

[Zhang 04] Zhang, Y., *et al.*, "Energy-Aware Deterministic Fault Tolerance in Distributed Real-Time Embedded Systems," *DAC*, 2004.

[Zhang 05] Zhang, Y., *et al.*, "Optimal Procrastinating Voltage Scheduling for Hard Real-Time Systems," *DAC*, 2005.

[Zhuo 05] Zhuo, J., and C. Chakrabarti, "System-Level Energy-Efficient Dynamic Task Scheduling," *DAC,* 2005.