Department of Electrical and Computer Engineering

O/S-enabled on-line Software-Based Self-Test and Recovery for Resilient Shared-Memory Multicore Systems

Michael A. Skitsas

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the University of Cyprus

December, 2017

VALIDATION PAGE

Michael A. Skitsas

O/S-enabled on-line Software-Based Self-Test and Recovery for Resilient Shared-Memory Multicore Systems

The present Doctorate Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering, and was approved on December 5, 2017 by the members of the Examination Committee.

Research Supervisor _____
Dr. Maria K. Michael

Research Supervisor _____
Dr. Chrysostomos Nicopoulos

Committee Chair _____
Dr. Theocharis Theocharides

Committee Member _____
Dr. George Ellinas

Committee Member _____
Dr. Stelios Neophytou

Committee Member _____
Dr. Michalis Psarakis

DECLARATION OF DOCTORAL CANDIDATE

The present doctoral dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

_____

_____

Περίληψη

Καθώς η τεχνολογία εξελίσσεται, τα μεγέθη αποτύπωσης ολοκληρωμένων κυκλωμάτων συρρικνώνονται, τα τρανζίστορ γίνονται λιγότερο αξιόπιστα. Ως αποτέλεσμα αυτής της εξέλιξης, τα μελλοντικά συστήματα αναμένεται να είναι πιο ευάλωτα σε φαινόμενα φθοράς (με την πάροδο του χρόνου και την χρήση). Το ζήτημα της φθοράς με την πάροδο του χρόνου και της βαθμιαίας υποβάθμισης καθιστά αναγκαία την χρήση μηχανισμών που επιτρέπουν την προστασία του συστήματος από ανεπιθύμητες συμπεριφορές διευκολύνοντας έτσι στην ανίχνευση, τον μετριασμό ή / και την αποκατάσταση ορθής λειτουργίας από σφάλματα καθ 'όλη τη διάρκεια ζωής του συστήματος. Πρόσφατα, στην βιβλιογραφία έχουν προταθεί αρκετές τεχνικές για έλεγχο των συστημάτων που να επιτρέπουν τη δυναμική ανίχνευση μόνιμων σφαλμάτων. Η ανίχνευση σφαλμάτων από τα ίδια τα συστήματα με την χρήση λογισμικού είναι μια διαδεδομένη τεχνική στον τομέα ελέγχου ψηφιακών κυκλωμάτων και μικροεπεξεργαστών. Η λειτουργία αυτή βασίζεται στην εκμετάλλευση των υφιστάμενων διαθέσιμων πόρων που υπάρχουν στο σύστημα. Πέρα από την ανίχνευση σφαλμάτων, τα σύγχρονα συστήματα πρέπει να ενισχυθούν με μηχανισμούς που είναι σε θέση να επιδιορθώσουν και να ανακτήσουν την ορθή λειτουργία του συστήματος στην παρουσία σφάλματος, προκειμένου να παραμείνει λειτουργικό παρά την ύπαρξη μόνιμων βλαβών.

Σκοπός αυτής της διατριβής είναι η ανάπτυξη τεχνικών για: (i) ανίχνευση σφαλμάτων, (ii) μεθοδολογίες προγραμματισμού για την αύξηση της διαθεσιμότητας του συστήματος κατά τη διάρκεια των ελέγχου του συστήματος και (iii) ενίσχυση του συστήματος με δυνατότητες αποκατάστασης. Το πρώτο μέρος αυτής της εργασίας εισάγει ένα νέο παράδειγμα ανίχνευσης σφαλμάτων που ελέγχει το σύστημα για σφάλματα στη διακριτότητα των επί μέρους συστημάτων (υπολογιστικών μονάδων) ενός επεξεργαστή σε συστήματα πολλαπλών πυρήνων λαμβάνοντας υπόψιν το ιστορικό της λειτουργίας τους. Συγκεκριμένα, αναπτύχθηκε το πλαίσιο DaemonGuard που επιτρέπει την παρατήρηση σε πραγματικό χρόνο των επί μέρους υπολογιστικών συστημάτων ενός επεξεργαστή εκτελώντας

μια διαδικασία για έλεγχο (σε τοπικό επίπεδο) χωρίς να γίνεται ολικός έλεγχος του επεξεργαστή για σφάλματα σε επίπεδο υλικού. Αυτή η τεχνική στοχεύει στη μείωση του χρόνου εκτέλεσης ελέγχου αποφεύγοντας τον συχνό έλεγχο των μονάδων των οποίων η χρήση ήταν σε χαμηλό επίπεδο. Το δεύτερο μέρος διερευνά τη σχέση μεταξύ του χρόνου κατά τον οποίο το σύστημα βρίσκεται υπό έλεγχο και του συνολικού χρόνου που χρειάζεται να ελεγχθούν όλοι οι πυρήνες του συστήματος. Για αυτό το σημείο στην έρευνα μας, αναπτύσσουμε ένα πλαίσιο εξερεύνησης ικανό να προσδιορίσει την καλύτερη πολιτική προγραμματισμού για να αυξήσει τη διαθεσιμότητα του συστήματος. Επιπλέον, προτείνουμε, αξιολογούμε και ενσωματώνουμε μια νέα μεθοδολογία που στοχεύει στην περαιτέρω βελτίωση των τεχνικών καθώς το σύστημα μεγαλώνει. Για το τελευταίο μέρος της παρούσας διατριβής, προτείνουμε τεχνικές που βελτιώνουν το προτεινόμενο πλαίσιο και μπορούν να υποστηρίξουν δυνατότητες αποκατάστασης της σωστής λειτουργίας παρά την εμφάνιση σφαλμάτων. Συγκεκριμένα, προτείνουμε έναν αποδοτικό μηχανισμό ανάκτησης και επαναφοράς, ο οποίος, μετά την ανίχνευση σφαλμάτων, μπορεί να επαναφέρει το σύστημα στην πιο πρόσφατη έγκυρη κατάσταση ορθής λειτουργίας και να επαναλά-βει την εκτέλεση, υποθέτοντας την απενεργοποίηση του ελαττωματικού πυρήνα, οδηγώ-ντας έτσι σε ένα υποβαθμισμένο μεν, αλλά λειτουργικό σύστημα. Όλες οι προτεινόμενες τεχνικές αξιολογούνται μέσω μιας σειράς πειραμάτων με τη χρήση προσομοίωσης.

Abstract

As technology scales deep into the sub-micron regime, transistors become less reliable. Future systems are widely predicted to suffer from considerable aging and wear-out effects. The issue of aging and gradual degradation necessitates the use of mechanisms that can enable protection against undesired system behavior by facilitating detection, mitigation, and/or recovery from faults throughout the lifetime of the system. Recently, several on-line testing techniques have been proposed in literature enabling dynamic detection of permanent faults. 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. Beyond the detection of faults, modern systems must be enhanced with mechanisms able to self-repair and recover the system to a fault-free state, in order to remain functional despite the presence of permanent faults.

The objectives of this work are to develop techniques for: (i) on-line fault detection, (ii) scheduling methodologies to increase the system availability during testing and (iii) enhance the system with recovery capabilities. The first part of this thesis introduces a new paradigm of SBST that performs testing at the granularity of individual microprocessor core components in multi-/many-core systems based on the utilization. In particular, we develop the DaemonGuard, a framework that enables the real-time observation of individual sub-core modules and performs on-demand selective testing of modules that have been stressed. This technique aims to reduce the testing time by avoiding the over-testing of under-utilized units. The second part investigates the relation between system test latency and test time overhead under several scheduling policies. For this part we develop an exploration framework able to identify the best scheduling policy in order to increase system availability under a given test latency constraint. Additionally, 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) is integrated and evaluated under the developed exploration framework. For the last part of this thesis, we propose to enhance our framework to support fault recovery capabilities. In particular, we pro-

pose an efficient check pointing and rollback recovery mechanism which, upon fault detection, can restore the system to the most recently valid correct state and resume the normal operation assuming disabling of the faulty core, thereby leading to a healthy (but degraded) system. All the proposed techniques are evaluated through a series of experiments using a full-system, execution-driven simulation framework running a commodity operating system and real multi-threaded workloads.

Acknowledgments

First, I would like to express my deepest gratitude to my research co-advisors, Dr. Maria Michael and Dr. Chrysostomos Nicopoulos for their help, guidance and support during my research studies. I am also indebted to the members of my examination committee Dr. Theocharis Theocharides, Dr. George Ellinas, Dr. Michalis Psarakis and Dr. Stelios Neophytou, who dedicated time to review this dissertation, and provided feedback and suggestions.

I would like to thank the faculty of Electrical and Computer Engineering department,as well as the KIOS administrative assistants. I would also like to acknowledge the University of Cyprus for supporting my research.

Furthermore, I would like to express my gratitude to my colleagues at KIOS Research Center for contributing in the creation of a friendly work environment, their friendship and the constructive discussions. A special thanks to my best man Demetris Stavrou and my friends Demetris Eliades, George Milis, Vasso Reppa, Alexandros Kyriakides, Stavros Hadjitheophanous and Nikodimos Georgiades for their ongoing encouragement towards the completion of my goals.

From these acknowledgments, I could not have left behind my beloved parents, my brother Marios and my sister Polyxeni for their endless support and understanding.

This thesis is dedicated to my wife Ismini for her love and patience and to my three children Andreas, Irene and Argyro for giving me extra motivation and filling my life with wonderful moments!

Publications

Journal Publications

1. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”Exploring System Availability during Software-Based Self-Testing of Multi-core CPUs,”, Under Review, Journal of Electronic Testing: Theory and Applications.

2. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors,” in IEEE Transactions on Computers, vol. 65, no. 5, pp. 1453-1466, May 1 2016.

Book Chapters

1. **M. A. Skitsas**, C. Nicopoulos, P. Bernadi, E. Sanchez, M. K. Michael, ”Chapter 15: Self-testing of multi-core processors”, The Institution of Engineering and Technology (IET), IET Digital Library, to appear.

Conference Proceedings

1. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”Toward efficient check-pointing and rollback under on-demand SBST in chip multi-processors,” 2015 IEEE 21st International On-Line Testing Symposium (IOLTS), Halkidiki, GR, 2015, pp. 110-115.

2. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”Exploring Check-Pointing and Rollback Recovery Under Selective SBST in Chip Multi-Processors,” in Proc. 4th MEDIAN Workshop, pp. 74-77, Grenoble, FR, 2015.

3. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”Exploration of system availability during software-based self-testing in many-core systems under test latency constraints,” 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam, 2014, pp. 33-39.

4. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”DaemonGuard: O/S-assisted selective software-based Self-Testing for multi-core systems,” 2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), New York City, NY, 2013, pp. 45-51.

5. **M. A. Skitsas**, C. A. Nicopoulos and M. K. Michael, ”Toward Selective Software-Based Self-Testing in Multi-Core Microprocessors,” in Proc. 1st MEDIAN Workshop, pp. 71-76, Annecy, FR, 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The era of nanoscale technology has ushered designs of unprecedented complexity and immense integration densities. Billions of transistors now populate modern multi-core microprocessor chips and the trend is only expected to grow, leading to single-chip many-core systems [1]. However, a side effect of this deep technology scaling is the exacerbation of the vulnerability of systems to unreliable components [2]. Beyond the static variation of transistors that can occur during the fabrication, which is expected to get worse, current and future technology also suffers from dynamic variations. Single-event upsets (soft errors) are another source of concern with a direct impact on the system's reliability. Finally, a third source of unreliable hardware operation that can lead to permanent system failures is the increased sensitivity to aging (time-dependent device degradation) and wear-out artifacts, due to the extreme operating conditions.

The issue of increased vulnerability and the expected increase in the occurrence of transient and permanent failures – as a result of future technologies – render the one-time factory testing of the system inadequate. The new state of affairs necessitates the use of mechanisms that can enable protection against undesired system behavior by facilitating detection, mitigation, and/or recovery from faults throughout the lifetime of the system [3]. Several fault detection techniques have been proposed in order to detect faults during the normal lifetime of the chip. Such schemes broadly fall into two categories: (a) Concurrent methods relying on fault-tolerant mechanisms (i.e., redundancy techniques) [4], and (b) Non-concurrent periodic on-line testing [5], which aims to detect errors that are, subsequently, addressed using various techniques.

Multi-/many-core microprocessor chips with an abundance of identical computational resources would appear to be ideal for implementing high availability solutions on-chip, due to the inherent replication of resources (i.e., the processing cores). Multi-core systems should

remain operational despite the occurrence of permanent and/or transient faults. Detection and diagnosis of such fautls constitute the first and perhaps the most important step towards the implementation of self-healing multi-core systems. The already proposed self-testing techniques for simple and even more complex microprocessors have matured enough, while current and future trends in the self-testing research area are adapting these techniques to multi-/many-core processors. Considering the huge range of today's applications that require many and different types of computational systems, researchers aim to develop self-testing techniques targeting either general-purpose multi-core microprocessors, or embedded microprocessors and microcontrollers that constitute application-specific Systems-on-Chip (Soc).

Non-concurrent periodic online testing is one methodology used traditionally for the detection of permanent faults (hard failures). Moreover, it can be used for circuit failure prediction within the cores of modern microprocessors, due to either infant mortality reasons (early-life failures), or aging-related factors [6–10]. Hardware-based schemes, typically using Built-In Self-Testing (BIST) [11], as well as software-based schemes, known as Software-Based Self-Testing (SBST), can be employed for this problem. The SBST technique [12–20] is an emerging new paradigm in testing that avoids the use of complicated dedicated hardware for testing purposes. Instead, SBST employs the existing hardware resources of a chip to execute specific (software) programs that are designed to test the functionality of the processor. The test routines used in this technique are executed as normal programs by the CPU cores under test. As a result, the major cost of SBST is the *time* overhead incurred by the execution of the appropriate test routines on the CPU. The hardware overhead is either non-existent, or negligible, and no Instruction Set Architecture (ISA) extensions are required.

Beyond the detection of faults, modern systems must be enhanced with mechanisms able to self-repair and recover the system to a fault-free state, in order to remain functional despite the presence of permanent faults. To maintain proper operation of the system, several error recovery techniques have been proposed. These techniques are classified mainly in two categories: (i) Forward Error Recovery (FER), and (ii) Backward Error Recovery (BER). In the first (FER), the usage of redundant hardware is necessary for error detection and recovery. On the other hand, BER requires to store a fault-free state of the system using checkpoints for error recovering. Once an error is detected, the system is able to rollback to the fault-free state and re-execute the affected workload, assuming it supports reconfiguration/fault-containment capabilities to rule out the malfunctioning component. In most cases, BER does not require extra hardware to support the recovery procedure, and the imposed overhead is in execution time, since a portion of the executed workload needs to be re-executed. There is also some

storage overhead to store the checkpoints.

1.1 Thesis Objectives

The objectives of this thesis are to develop techniques for: (a) on-line fault detection, specifically SBST methodologies for multi-core systems in order to reduce the imposed testing overhead during testing and to increase the system availability by proposing test scheduling policies, and (b) efficient check-pointing and roll-back mechanisms to assist recovery in the presence of permanent faults. Combining the proposed techniques we aim to develop a dependable system able to tolerate permanent hardware failures encountered during the normal operation of many-core architectures. The first part of this thesis introduces a new paradigm of SBST that performs testing at the granularity of individual microprocessor core components in multi-/many-core systems based on the utilization. This thesis, is the first work that proposed and implemented selective testing. The second part investigates the relation between system test latency and test time overhead under several scheduling policies targeting large systems in terms of number of cores. As a result, we aim to identify the best scheduling policy that increases system availability under a given test latency constraint. Both the on-line fault detection approaches are deployed and evaluated under a real environment using a unix based OS (Solaris) targeting a SparcV9 multi-core architecture. Finally, in the third and last part of this thesis, we enhanced our framework to support fault recovery in the presence of permanent faults.

For the development of the above techniques, we proposed the DaemonGuard Framework, a light-weight, minimally-intrusive framework, which transparently orchestrates the procedure of SBST and accommodates algorithms for the creation of checkpoints and to support the recovery to a fault-free state. The basic DaemonGuard Framework is generic enough and can support different techniques for SBST as well as test programs that target hardware components of a multi-core system at different granularities (i.e. core level, functional unit). The DaemonGuard Framework comprises an always-active OS process (the Testing Manager) and a number of dormant daemons (the test routines), which are awakened according to the implemented technique.

The developed framework in this thesis can be considered as an flexible tool for the scientific community with research interests in the area of self-testing and specifically the SBST. The ability to monitor the system's activity and provide a real-time feedback of the "health" of the system's module combined with the O/S resident processes for testing can help research groups to easily implement, deploy and evaluate new SBST techniques. Besides the scientific

community, the concept of DaemonGuard Framework can be adopted by different industry sectors (i.e automotive industry) towards a low cost reliable solution.

1.2 Self-Testing of Multi-Core Microprocessors

The main purpose of this thesis is to provide new approaches for self-testing of multi-core microprocessors. The existing SBST approaches are mainly focused on the development of test programs that are periodically applied at microprocessors. The first part of our research activity, focuses to the reduction of imposed testing overheads from the execution of test programs during the normal operation of systems. The motivation of this approach is to perform test based on the utilization by avoiding testing under-utilized components. To achieve the utilization-based SBST, the individual cores of a multi-core system are divided in several functional units that can be tested independently. This allows us to perform selective testing on the high-utilized functional units.

To achieve this, we utilize the DaemonGuard Framework to enable the real-time observation of individual sub-core modules and the initiation of on-demand selective testing. For this approach, the main component of the DaemonGuard Framework is the Testing Manager software process, which is responsible for the invocation of the various Test Daemons (test programs targeting individual functional units of each core of the system), based on the utilization information provided by hardware instruction counters residing alongside each functional unit within the CPU cores. The main function of the Testing Manager is the checking for pending test requests by any functional unit of any core within the system. Furthermore, the DaemonGuard mechanism is able to exploit the memory hierarchy of the CPU to expedite the testing process. DaemonGuard is augmented with the capability to perform *cache-aware* selective testing, whereby test sessions are initiated not only based on unit utilization, but also on the recent history of test sessions by other similar units in other cores. Consequently, test programs can benefit from cache-resident blocks, thereby avoiding the need for many expensive off-chip memory accesses.

In order to provide a comprehensive framework for the orchestration of testing activity under multi-core systems, a different approach of SBST is also investigated in this thesis. In particular, we investigate the impact of SBST for shared-memory multi-core systems in cases where the need for testing the entire system is necessary. One salient aspect of on-line self-testing is the scheduling of the test programs during a testing session. For the second part of this thesis, we proposed to investigate scheduling methodologies aiming to increase system

availability during testing. The main focus is to investigate the intricate relationship between the test latency and the test-time overhead under different test scheduling policies. 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 already resident in the Last Level Cache as a result of a previous core's testing session.

1.3 System Recovery in the Presence of Permanent Faults

For the last part of this thesis, an enhancement of DaemonGuard Framework to support fault recovery capabilities is proposed. When a testing session is completed, during selective testing, a checkpoint that holds the fault-free state of the system must be captured. Since the considered detection mechanism performs on-demand testing at sub-core granularity, checkpoints are captured and stored in the system at irregular time intervals. As a result of this, we are motivated to investigate techniques that aim to solve the problems arising by the application of recovery policy over selective testing.

The main focus of this thesis is the system recovery process, once a permanent fault has been detected. Hence, a recovery mechanism is introduced, which is able to keep the system operational despite the occurrence of permanent faults. The main components of the proposed recovery mechanism are (a) the Checkpoint Manager, and (b) the Recovery Manager, which are responsible, respectively, for the creation of system checkpoints and determining a valid checkpoint (among the multiple stored ones) to roll back.

After the detection of a permanent fault and the subsequent recovery to a fault-free state, the system should be able to isolate the faulty module and reconfigure itself to a fault-free (albeit degraded) operational mode. We assume that the operating system is aware of the reconfiguration policy and, upon rollback, it is able to re-distribute the workload to the fault-free (and still active) cores, and resume execution from the selected checkpoint. Note that the reconfiguration mechanism is beyond the scope of this thesis.

1.4 Contributions

The contributions of this thesis are:

- The Development of DaemonGuard Framework, a light-weight framework that can enable SBST from the O/S level. The modular and scalable design of this framework allows the accommodation and evaluation of different SBST approaches.

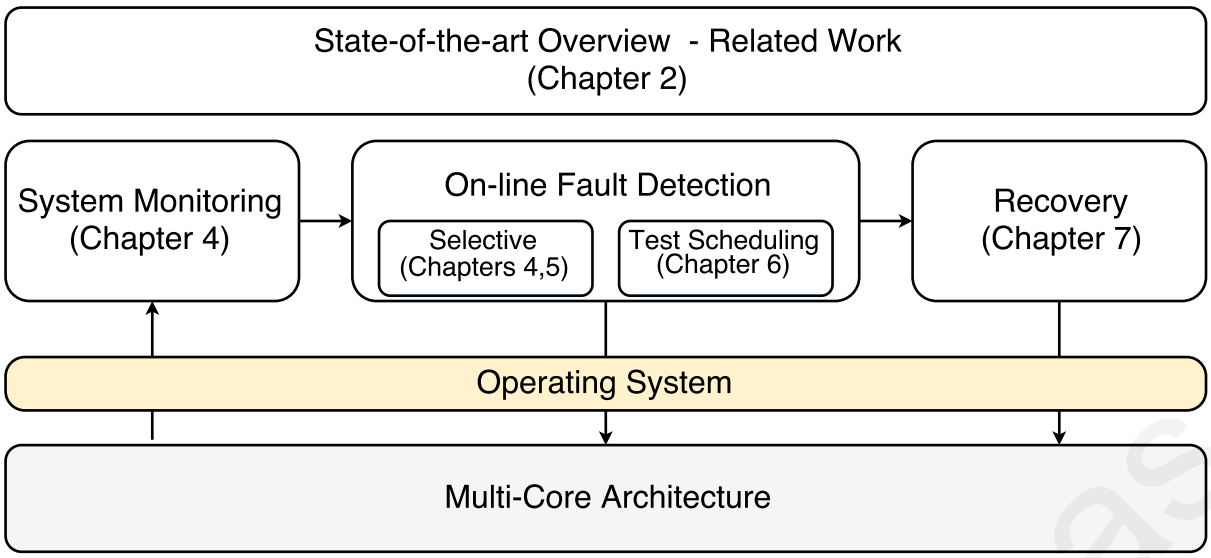


Figure 1.1: Thesis Overview

- The Utilization-Based Selective SBST contributes to the substantial reduction in testing overhead by avoiding the testing of non-utilized functional units.
- A further improvement of Selective SBST, by considering the Cache-Aware Selective Testing. Testing sessions are initiated not only on utilization statistics per unit, but considering the recent activity of testing in order to exploit the cache-resident blocks of test data.
- Optimization of test scheduling process in order to minimize the test-time overhead and maximize the system availability.
- A clustering approach of selecting the cores under test is proposed in order ensure the reduction in test-time overheads while the system scales up.
- An efficient checkpointing and roll-back recovery mechanisms for systems that perform on-demand testing on a specific part of the system (i.e. Selective SBST).

1.5 Thesis Outline

The outline of this thesis is illustrated in Figure 1.1. Chapter 2 presents a review of state-of-the-art techniques and methodologies for the self testing of multi-core processors. In Chapter 3 we present the DaemonGuard Framework by describing the most important components. Details about a series of simulation experiments in order to acquire statistics about the executed instructions per functional unit within a core by the application of several benchmarks is also

presented in Chapter 3. Chapter 4 introduces the Selective SBST while Chapter 5 presents the cache-aware selective SBST approach. Test scheduling techniques and optimization of system availability is presented in Chapter 6. In Chapter 7, we propose mechanisms to support recovery in the presence of permanent faults. Finally, in Chapter 8 we present the concluding remarks and future work.

Chapter 2

State-of-the-Art Overview

2.1 Taxonomy of On-line Fault Detection Methods

Over the last several years, several self-testing approaches have been proposed towards reliable and dependable multi-core microprocessor systems. Considering the implementation details as well as the architectural level of application, on-line fault detection techniques can be classified in four main categories [3]. Figure 2.1 presents a tree diagram of the taxonomy of on-line fault detection methods. The four categories that on-line fault detection methodologies can be classified are: (a) *redundant execution* where the exploitation of "spare" processing elements (i.e. cores) for the replication of normal workload can lead to the detection of failures, (b) *dynamic verification* where the fault detection is based on the validation of program invariants during runtime, (c) *anomaly detection* where the system is monitored for the detection of symptoms of faults, and (d) the *Self-Test-based* techniques where the on-line fault detection is done by the application of test patterns. Based on the level of implementation and the way that faults are detected in each category, the first three categories can be considered as a Non-Self-Test-based approaches as well.

The main characteristic of Non-Self-Test-based methods is that the detection of faults is achieved by exploiting the normal workload that is applied in the system. As a result of these approaches, in fault-free executions the imposed performance overheads in normal workloads is almost zero while hardware overhead exists either using additional hardware components (i.e. using checkers) or by increasing (i.e. doubling) the resources for the execution of the workload (i.e redundant execution). In Self-Test-based techniques, the detection of faults is achieved by the application of test patterns either by using hardware support (i.e. scan chains), in hardware-based techniques, or by exploiting the available resources and ISA, in software-

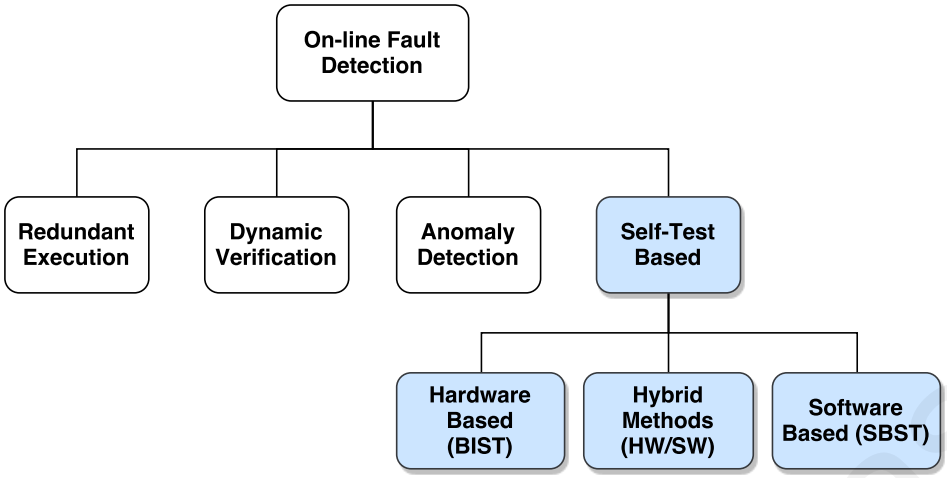


Figure 2.1: Taxonomy of on-line fault detection methods

based techniques.

During self-testing, test patterns are applied in a periodic manner where the normal operation of module under test (i.e. core) is suspended and turned into testing mode. As a result of this, Self-Test-based approaches imposed performance overheads in the system. In the era of multi-/many-core architectures with multiple homogeneous cores appearing in the same chip, despite the suspension of the normal operation of the core under test, the entire system remain operational as normal workload can be scheduled in one of the remain available resources.

2.2 Non-Self-Test-based Methods

A fault detection approach targeting microprocessor cores is to run two identical copies of the same program (different executions either at the thread or process levels) and compare their outputs. Redundant execution is feasible both at the hardware level and the software level. In the era of multi-/many-core architectures with multiple homogeneous cores integrated on the same chip, and the capability to execute multiple threads (or processes) simultaneously, the hardware-based redundant techniques (such as Dual/Triple Modular Redundancy, DMR, TMR) can be applied with significantly reduced performance overheads, targeting both transient and permanent faults. The application of these hardware-based redundant techniques is feasible because of the rather improbable simultaneous utilization of all the processing resources at any given time. In literature, two forms of redundant techniques at hardware level can be found the structural redundancy (lockstep configuration) and temporal redundancy (redundant multi-threading).

In the former, identical cores are working in close synchronization either at instruction level or even more at cycle level. Aggarwal et al. [21] propose DMR and TMR configurations for CMPs which provide error detection and error recovery through fault containment and component retirement. LaFrieda et al. [22] presents a dynamic core coupling (DCC) technique that allows arbitrary CMP cores to verify each other's execution. Unlike existing DMR techniques that require a static binding of adjacent cores via dedicated communication channels and buffers, the proposed technique avoids the static binding of cores. Li et al. [23] proposed a variation-aware core-level redundancy scheme in order to achieve robust computation in many-core systems with inter-core variations and mixed workloads.

Several Redundancy Multi-Threading (RMT) techniques have been proposed targeted single-core chips that support Simultaneously Multi-Threading (SMT) [24,25]. The evolution of technology and the era of multi-core systems forced researches to develop techniques exploiting the nature of Chip Multi-Processors. Mukherjee et al. [26] studied RMT techniques in the context of both single- and dual-processor simultaneous multi-threaded (SMT) propose a Chip-level Redundant Threading (CRT) for CMP architectures. [27] present a Software-based Redundant Multi-Threading (SRMT) approach for transient fault detection targeting general-purpose chip multi-processors (CMPs). Furthermore, Chen et al. [28] explores how to efficiently assign the tasks onto different cores with heterogeneous performance properties in order to achieve high reliability and satisfy the tolerance of timeliness. Mitropoulou et al. [29] proposes a compiler-based technique that makes use of redundant cores within a multi-core system to perform error checking.

As we mentioned, redundant execution techniques can be found at software level as well. In this case, the redundant execution of workload at different architectural levels (i.e. instruction, thread, process) is based on the re-execution at the same resources. As a result, redundant techniques at this level can only detect transient faults. Oh et al. [30] propose a pure software technique Error Detection by Duplicated Instructions(EDDI) that duplicates instructions during compilation and uses different registers and variables for the new instructions. Reis et al. [31] present SWIFT, a novel, software-only, transient fault detection technique. Recently, Mushtaq et al. [32] propose an error detection mechanism that is optimized to perform memory comparisons of the replicas efficiently in user-space. Kuvaiskii et al. [33] present HAFT, a fault tolerance technique using hardware extensions of commodity CPUs to protect unmodified multi-threaded applications against such corruptions. HAFT utilizes instruction-level redundancy for fault detection and hardware transactional memory for fault recovery.

The second category of on-line fault detection in the considered taxonomy is the **Dynamic**

Verification. The operation of this approach is based on the run-time verification of specific invariants that in a fault free execution are true. The verification is based on dedicated hardware checkers. The research challenge within these approaches beyond to maintain a low-cost in terms of hardware implementations, is to provide a comprehensive set of invariants aiming to increase the detected faults. Dynamic Verification was first introduced by Todd Austin in [34], where a novel micro-architecture based technique that permits detection and recovery of all transient and permanent faults in the processor core is proposed. DIVA, a Dynamic Implementation Verification Architecture,) uses a simple checker core to detect errors in a speculative, super-scalar core. Despite, the low-cost implementation requirements compared with a complex super-scalar core, in multi-core systems with simpler microprocessors the overhead becomes significant. Meixner et al. [35] proposes Argus, a low-cost, comprehensive fault detection targeting simple cores. Based on dynamic verification, Argus uses four invariants that guarantee the correct operation of a core, control flow, computation, dataflow and memory. Additionally, Meixner and Sorin [36] proposes the Dynamic Dataflow Verification (DDFV) another approach of dynamic verification using a high-level invariant. Fault are detected by verifying at runtime the dataflow graph.

The last category of Non-Self-Test-based approaches is the **Anomaly Detection** where that faults are detected by monitoring the software for anomalous behavior or symptoms of faults. According the level of symptoms that can detect, anomaly detection approaches can be further classified in three categories [3], (a) those that detect data value anomalies, (b) those that detect micro-architectural behavior anomalies, and (c) those that detect software behavior anomalies. Racunas et al. [37] dynamically predict the valid set of values that an instruction will produce, and consider a departure from this prediction as a symptom of a (transient) fault. Wang and Pater [38] detect transient faults without significatn overhead by utilizing symptom. Feng et al. [39] in Shoestring, enhance ReStore by selectively duplicating some vulnerable instructions with simple heuristics. Li et al. [40] propose the detection of faults by deploying low overhead monitors for simple software symptoms at the operating system level. Their approach is rely on the premise that micro-architectural strcuctures eventually propagate symptoms to the operating system.

2.3 Self-Test-based Methods

The last category of the considered taxonomy is the **Self-Test-based Methods** where the detection of faults is based on the application of test patterns. Based on the architectural level

of implementation and application of test patterns, Self-Test-based methods are further classified in three categories: (i) the Hardware-based methods, known as Built-In Self-Test (BIST), where self-testing is maintained by hardware components, (ii) Software based methods, known as Software-Based Self-Testing (SBST) where the application of test patterns in done using software programs, and (iii) the hybrid approaches where self-testing is supported both by hardware level and software level.

2.3.1 Hardware-based Methods (BIST)

Built-In Self-Test (BIST) approaches can, theoretically, perform non-concurrent error detection of microprocessors during their entire lifetime. Hardware-based BIST exploits special circuits located on the chip that produce, monitor, and evaluate the tests needed by the cores. Traditionally, BIST techniques are used for manufacturing testing, but the advances in technology necessitates the application of such techniques during the lifetime and normal operation of the system targeting wear-out and aging-related faults.

Shyam et al. [41] utilize existing distributed hardware BIST mechanisms to validate the integrity of the processor components in an on-line detection strategy. For each of the pipeline components, a high quality input vector set is stored in an on-chip ROM, which is fed into the modules during idle cycles. A checker is also associated with each component to detect any defect in the system.

Li et al. [42] present CASP, Concurrent Autonomous chip self-test using Stored test Patterns, is a special kind of self-test where a the system tests itself during normal operation without any downtime visible to the end-user. The operation of CASP is based on two main functions: (i) the storage of very thorough test patterns in non-volatile memory, and (ii) the architectural and system-level support for autonomous testing of one or more cores in a multi-core system without suspend the normal system operation. The testing procedure under CASP solution is composed by four phases: (1) the test scheduling where one or more cores may be selected for testing, (2) the pre-processing phase where the core under test is temporarily isolated saving the current state, (3) the testing where test patterns are loaded and applied to the core under test and finally, (4) restore the state and resume the operation of tested core. The evaluation of the proposed technique is done using the OpenSPARC T1 multi-core processors where a fault coverage more than 99% using 5MB of stored patterns is achieved.

Lee et al. [43] propose a novel self-test architecture which achieves high fault coverage by using deterministic scan-based test patterns. The main idea of this work is the compression and

storage on the chip while the decompression and application to the circuits under test will take place during the testing. As the testing is performed based on deterministic patterns, pseudo-random patterns are not required and this results in the reduction of testing time. Experimental results on OpenSPARC T2, a publicly accessible 8-core processor containing 5.7M gates, show that all required test data for 100% testable stuck-at fault coverage can be stored in the scan chains of the processor with less than 3% total area overhead for the whole test architecture.

2.3.2 Software-based Methods (SBST)

The SBST technique is an emerging new paradigm in testing that avoids the use of complicated dedicated hardware for testing purposes. Instead, SBST employs the existing hardware resources of a chip to execute normal (software) programs that are designed to test the functionality of the processor itself. The test routines used in this technique are executed as normal programs by the CPU cores under test. The processor generates and applies functional-test programs using its native instruction set. In recent years, several active research teams have been working in the area of SBST focusing on different approaches. The two main phases of SBST towards the detection of a fault are (a) the test-program development, and (b) the execution of the test program on the system.

Test program development for multi-core architectures Several research teams are working on the development of test programs with multi-dimensional scope, such as to increase the fault coverage, extend the considered fault models (i.e., to also include delay faults), reduce the test-program size, achieve savings in testing time overhead, etc. More recently, in the age of chip multiprocessors and multi-threading, the development of test programs also focuses on the effective adoption of the underlying hardware to yield self-test optimization strategies that benefit from the targeted architectures.

Foutris et al. [14] propose a Multi-Threaded Software-Based Self-Test (MT-SBST) SBST methodology targeting multi-threaded multi-core architectures. The proposed MT-SBST methodology generates an efficient multi-threaded version of the test program and schedules the resulting test threads into the hardware threads of the processor to reduce the overall test execution time and on the same time to increase the overall fault coverage. MT-SBST approach significantly speeds up testing time at both the core level (3.6 times) and the processor level (6.0 times) against single-threaded execution, while at the same time it improves the overall fault coverage

Kaliorakis et al. [44] propose a test-program parallelization methodology for many-core architectures, in order to accelerate the on-line detection of permanent faults. The proposed methodology is based on the identification of the memory hierarchy parameters of many-core architectures that slow down the execution of parallel test programs in order to identify the parts that can be parallelized and therefor to improve the performance. The evaluation of the methodology in [44] is done using the Intel's Single Chip Cloud Computer showing an up to 47.6X speedup compared to a serial test program execution approach.

Test program scheduling One salient aspect of on-line testing and specifically of SBST is the scheduling of the test program(s). In light of the rapid proliferation of multi-/many-core microprocessor architectures, the test scheduling issue becomes even more pertinent. One approach is to periodically initiate testing on the system targeting individual cores, or all the cores simultaneously. In any case, the interruption of the current execution of normal workload is unavoidable. Another approach is the execution of test programs on cores that have been observed to be idle for some time. Recent techniques have proposed the monitoring of the utilization of the system and subsequent selection of specific cores to be tested.

Apostolakis et al. [12] 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 aiming at the reduction of the total test application time, and thus, test cost, for the SMP, by increasing the execution parallelism and reducing both bus contentions and data cache invalidations The proposed solution is demostrated with detailed experiments on several multi-core systems based on OpenRISC 1200 processor.

A recent test-scheduling study for online error detection in multicore systems is discussed in [45]. The authors evaluate the performance of test programs applied on Intel's 48-core Single-chip Cloud Computer (SCC) architecture. Due to possible congestion within common hardware resources used by the various cores, the test time can be quite large with a significant impact on performance. As a result, the authors of [45] develop effective test scheduling algorithms to expedite the test process in such systems.

Skitsas et al. [46] investigate the relation between system test latency and test-time overhead in multi-/many-core systems with shared Last-Level Cache (LLC) for periodic SBST, under different test scheduling policies. The investigated scheduling policies primarily vary the number of cores in the overall system testing session. 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.

Monitoring system activity Recently, several techniques proposed the monitoring of system status over the time as an indicator for initiation testing procedure. Power, utilization, performance are among the metrics that are considered in the type of testing activity.

Gupta et al. [47] propose an adaptive online testing framework to significantly aiming to reduce the testing overhead. The proposed approach is based on the ability to assess the hardware health and apply detailed tests. Hardware health assessment is done using in-situ sensors that detect the progress of various wearout mechanisms. Results show a reduction in software test instructions about 80% while the sensor area overhead for a 16-core CMP system is 2.6%.

Haghbayan et al. [48] 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.

Skitsas et al. [49] investigate the potential of SBST at the granularity of individual microprocessor core components in multi-/many-core systems. While existing techniques monolithically test the entire core, the proposed approach aims to reduce testing time by avoiding the over-testing of under-utilized units. The methodology is based on a real-time observation of individual sub-core modules and performs on-demand selective testing of only the modules that have recently been stressed. Results indicate substantial reductions in testing overhead of up to 30×.

2.3.3 Hybrid Methods (HW/SW)

Beyond Self-Test-based methods that are purely implemented at one architectural level either on hardware or software, there is a different approach that spans in both architectural levels. The purpose of these hybrid approaches is to further improve the performance of self-test based methods by reducing the testing time, increasing the fault coverage, etc. The hardware architectural support provides the necessary substrate to facilitate testing, while the software makes use of this substrate to perform the testing. For the implementation of such approaches, modifications of the ISA and/or the extension of hardware components may be required.

Inoue et al [50] propose VAST, a Virtualization-Assisted concurrent, autonomous Self-Test that enables a multi-/many-core system to test itself, concurrently during normal operation, without any user-visible downtime. VAST is hardware and software co-design of on-line self-test features in a multi-/many-core system through integration of BIST (i.e. CASP) methods

with a virtualization software. Testing can be done in two ways, stop-and-test and migrate-and-test. Experimental results from an actual multi-core system demonstrate that VAST-supported self-test policies enable extremely thorough on-line self-test with very small performance impact.

Constantinides et al. [51] proposed an online testing methodology using an enhanced ISA with special instructions for fault detection and isolation. Structural tests are performed by applying test patterns using software routines. The test routines are executed periodically, after a number of executed instructions have committed, and checkpoints are used for recovery. The technique of Constantinides et al. is software-assisted, but it requires various hardware modifications. These intrusive modifications are needed, because the goal is to enable very detailed structural testing through existing scan-chain infrastructure.

A hardware and software co-design methodology for functional testing is proposed by Khan et al. [52]. The testing methodology is based on the redundancy concept, whereby two cores execute the same program and capture corresponding footprints. The results of the executions are compared for fault detection. The choice of the test program is based on profiling that can be done offline or online. In [53], the authors propose a thread relocation methodology that uses dynamic profiling based on phase tracking and prediction.

In [54], a scalable self-test mechanism for online testing of many-core processors has been proposed. Several hardware components are incorporated in the many-core architecture that distribute software test routines among the processing cores, monitor behavior of the processing cores during test routine execution, and detect faulty cores. Results indicate a good fault coverage in a limited number of test cycles while the scalability in terms of hardware and timing overhead is maintained making the application to many-core systems feasible.

2.4 Work Related to this Thesis

The developed fault detection techniques under this thesis are classified in the Self-Test-based methods and particular under the Software-based methods. As the scope of this thesis is to reduce the imposed testing overheads and increase the system availability by applying the proposed test scheduling policies, the test program development is beyond the scope of this thesis. Test programs that are used in our framework are adopted from the literature and specifically for the Foutris et al [14].

As we mentioned earlier in this Chapter, several research groups are working on the development of test programs targeting processing elements at different granularities (i.e. functional

Table 2.1: High-level comparison of relevant online testing techniques.

	Selective SBST *(Proposed)*	Full-Core Functional Testing				Struct. Testing [51]
		[12]	[14]	[52]	[59]	
Test-Time Overhead	Low	High				Low
Overall Fault Coverage	Moderate to High					Very High
Fault Coverage per Testing Session	Unit-based	Core/System-based				
Detection Latency	Low to High (based on stress)	Depends on testing period				
System Avail. During Testing	Very High	High				None
Targeted Module	Functional unit	Core/System				
Test Triggering Method	Stress-based	Periodic				
Frequency of Test Triggering	High	Low (Depends on testing period)				
HW Support	Minimal	No	No	Yes	No	Yes
OS Modif.	No	No	No	Yes	Yes	No

units, entire core). Psarakis et al. [18] discussed the role of SBST in the microprocessor test and they proposed a taxonomy for different SBST methodologies according their test program development philosophy. The efficiency of an SBST approach depends on its test program development methodology and the effectiveness of the generated test routines. Another important parameter towards the efficient development of test programs is the automation of the self-test program generation process. The field of automatic test program generation includes generic approaches targeting processor cores [55,56] or approaches targeting specific high-performance processor architectures [57, 58].

The main contribution of this thesis the selective SBST where the test programs are applied at sub-core granularity (functional units) based on the utilization. Thus, an important element in this thesis is the usage of test programs that target the individual functional units of the considered architecture. Based on the literature, the most applicable test programs for our framework are those that are developed by Foutris et al. [14] targeting the functional units of a Chip Multi-Threaded (CMT) multiprocessor architecture.

Table 2.1 presents a high-level comparison of the proposed selective SBST technique to other relevant on-line testing techniques (including full-core functional testing). The main contribution of the work in this thesis is the reduction of the test-time overhead, by avoiding un-

necessary testing. As shown in Table 2.1, the "Overall Fault Coverage" of the selective and full-core testing approaches is the same. However, the limitation of selective testing is in terms of "Fault Coverage per Testing Session." The provided fault coverage of each testing session refers only to the particular functional unit under test (since only one functional unit is tested during each test session). Of course, this limitation only applies to each individual test session; once all units are tested over time, the overall fault coverage is identical to the one provided by full-core testing. Note that the under-utilized units are tested periodically. Another important parameter affected by the proposed selective testing technique is the detection latency. Since test triggering is based on utilization (or, more generally, stress), the detection latency could vary from low to high: fault detection in highly-utilized units is much faster than in under-utilized units, due to the more frequent testing sessions. Most importantly, the impact of the proposed testing approach on overall system performance is minimized by utilizing a software-based framework (with minimal hardware support), which runs seamlessly and transparently within the OS of the multi-core system.

Chapter 3

DaemonGuard Framework

In order to enable on-line software-based self-testing, targeting either sub-core functional units or the entire core and additionally to be able to support recovery in the presence of permanent faults, we proceeded to the development and implementation of the DaemonGuard Framework. This framework is mainly constituted from test programs residing at the operating system level and during their execution are targeting on the detection of permanent fautls. On top of these test programs, another software process, that is running at O/S level as well, is responsible for the orchestration of the testing activity. In particular, The Testing Manager process initiates the execution of the test programs based on a triggering mechanism either by monitoring the activity of the system or based on a predefined time interval (periodically). For each proposed methodology under this work, the Testing Manager process is suitably adapted, i.e in selective testing the execution of test programs is driven by the utilization metrics of the functional units within the cores of the system.

Figure 3.1 depicts a high-level overview of the DaemonGuard framework used to facilitate fault detection and recovery in a multi-/many-core setup. According to the architectural level of implementation, DaemonGuard is implemented in two levels, the Software Level residing within the O/S and the Hardware Level including the memory elements as well. The right side of the figure depicts the many-core architecture where each core of the system has its own private cache memory and is interconnected to the NoC. Additionally, with orange color are the enhancements related to the monitoring as well as the communication with message queues. Arrows with red color indicates the communication between the several components of the DaemonGuard Framework.

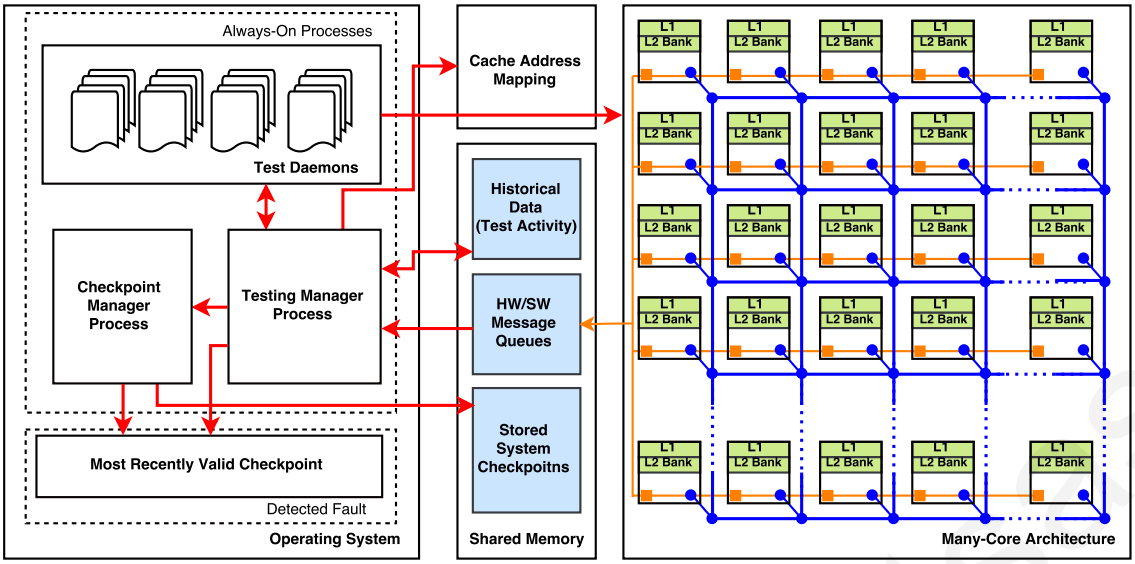


Figure 3.1: A high-level overview of the general framework employed in this thesis to facilitate the detection of faults (the focus is on *permanent* faults) and the recovery of the multi-/many-core system to a correct state. Correct system states are maintained in the form of checkpoints.

3.1 Software Level Implementation

In DaemonGuard Framework, fault detection and recovery mechanisms are maintained by software processes that resides in the O/S with the minimum support of hardware components (i.e. for the monitoring of the system). In this section, we are presenting an overview of those software processes that form the fault detection mechanisms, the Test Daemons that are responsible to apply test patterns over the functional components of a core and the Testing Manager process mainly responsible for the invocation of Test Daemons. Additionally, in order to assist the recovery procedure upon the detection of a permanent fault, two software processes are running at the O/S level as well. The first process is responsible for the creation of checkpoints (a fault free state of the system) and the second, implements an algorithm for the selection of a valid checkpoint for recovery.

3.1.1 Deamon-Based Test Programs (Test Daemons)

In multitasking computer operating systems, daemons are programs that run as background processes without any interaction at the user level. Daemons can be characterized as common processes; i.e., they have a Process ID (PID) and all operations pertaining to processes – such as the sleep function and various signals – can be applied. Daemon programs are loaded onto the system once, when the operating system starts up, and they run continuously during the

normal operation of the system. In the most cases, daemons are idle processes waiting for an appropriate signal, or interrupt, in order to become active and perform their task(s). An example of such OS daemon is the *printer server* in unix-based operating systems. The daemon is loaded and executed during the startup phase of the operating system and runs continuously in the background (in idle mode) while waiting for a job to print.

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

3.1.2 Testing Manager Process

The basic component of the DaemonGuard framework is the Testing Manager process which is responsible for the invocation of the various test daemons based on each consider self-testing methodology. Just like all the other other processes running during normal system operation, the O/S is responsible to schedule the Testing Manager process as well. The ability of the implementation of different algorithms (i.e. scheduling policies) for the orchestration of self-testing in multi-core systems is one of the characteristics of Testing Manager. This and the possibility to exploit information by monitoring the system allow us to implement and evaluate several efficient techniques and different scheduling schemas related with SBST.

Beyond the invocation of test daemons, the Testing Manager process is responsible for the collection of test results in order to initiate the recovery mechanism in the case that a permanent fautl is detected. In particular, upon completion of a test session, the corresponding test daemon sends the test result back to the Testing Manager. We note here that in our work, the analysis of the test results (i.e., the comparison of the test results with the golden/expected results) is carried out by each test daemon individually, followed by a pass/fail signal from each test daemon to the Testing Manager. An alternative implementation could delegate the analysis of every test result to the Testing Manager.

For the scheduling approaches of this work, two different self-testing scheduling policies are considered, (a) the utilization-based approach where a feedback about the executed instructions determines the initiation of a testing session at the granularity of sub-core unit, and (b) the periodic testing of entire system where the testing procedure is applied at core level. More

details and the benefits of each approach are given in the Chapters 4, 5 and 6.

3.1.3 Recovery Management and Support

Beyond the detection of faults, modern systems must be enhanced with mechanisms able to self-repair and recover the system to a fault-free state, in order to remain functional despite the presence of permanent faults. Once an error is detected by on-demand SBST, the system must be able to recover via rollback to a fault-free state. Hence, a recovery mechanism is introduced in DaemonGuard Framework, which is able to keep the system operational despite the occurrence of permanent faults. The main components of the proposed recovery mechanism are (a) the Checkpoint Manager, and (b) the Recovery Manager, which are responsible, respectively, for the creation of system checkpoints and determining a valid checkpoint (among the multiple stored ones) to roll back. Both mechanisms are resides in the O/S and are triggered by the Testing Manager process.

In the very simple form, the Checkpoint Manager is triggered upon the completion of each testing session (either for a sub-core functional unit or the entire core) and initiates the process for the creation of a checkpoint with the fault-free state of the system. As the main part of this work is based on on-demand testing (Selective SBST), an efficient Checkpoint Manager is proposed in order to reduce the number of checkpoints within the system. Details about the algorithm and its implementation are given in Chapter 7.

The second process related with the recovery mechanism is responsible to determine a valid checkpoint for rollback. Considering the utilization-based testing, the most recent checkpoint cannot ensure the fault-free state as the checkpoint is created upon the completion of a testing session targeting in many cases part of a core and not the entire system. As a result, an intelligent mechanism for to determine a valid checkpoint for recovery is introduced. Chapter 7 provides details about the Most Recent Valid Checkpoint algorithm.

3.2 Hardware Level

The DaemonGuard Framework is mainly developed to support Software-Based Self-Testing orchestrated by the O/S. However, to support the on-line SBST, a minimum hardware enhancements are required. The cores of the system are enhanced with monitoring components per functional units (counting the executed instructions) in order to support the utilization-based Selective SBST (see Chapters 4 and 5) while some modifications on the Cache Address

Mapping component are required in order to support the clustering approach of test scheduling techniques in large systems (see Chapter 6).

3.2.1 Hardware Support

The proposed selective testing methodology relies on the run-time gathering of information regarding the utilization of the various functional units within each core of the system. The term "utilization" refers to the number of instructions that have made use of the specific unit during normal operation. In order to collect this data, we assume the presence of a set of hardware instruction counters. There is one such counter for each functional unit within each core. instruction counters are developed to support the needs of this work and specifically the Selective SBST. Since, our scope is to develop the DaemonGuard as a generic Framework that can support different forms of SBST that may required feedback from the system, the instruction counters can be replaced by any other components or can be enhanced to include other type of information (i.e. historical data).

Furthermore, as one of the contributions of this work is to maintain the scalability of multi-core systems in terms of the number of cores, low-cost enhancements of hierarchical memory system are considered as well. In particular, in Chapter 6 we proposed the clustering approach where the testing data are distributed over the Last Level Cache Bank (LLC-Bank) of a set of cores in the system that forms the cluster. For the implementation of the clustering approach, the Cache Address Mapping component is modified in order to force the placement to the desired LLC-Bank.

3.2.2 Shared Memory

Another component of the system that is exploited for the implementation of DaemonGuard Framework is the shared memory. Beyond the usage during the execution of test daemons, shared memory is used as the main storage component of the created checkpoints as well as data structures (i.e queues) for the fault detection mechanisms.

In the utilization-based self-testing, where the feedback from the activity of functional units is necessary, a message queue located in main memory is developed in order to assist the communication between the Hardware and Software Levels. This queue is accessible both by the hardware monitoring units and Testing Manager process. Additionally, memory structures (i.e. queues, arrays) are developed as well in order to support the cache-aware testing, an improvement of selective testing where the history of testing activity is required for the prediction of

future testing sessions.

3.3 Impact of DaemonGuard Framework

As DaemonGuard Framework is mainly based on the execution of processes by the O/S concurrent with the normal workload, a series of experiments are performed in order to investigate the impact of DaemonGuard on overall system performance. It will be demonstrated that the cost of employing an always-active Testing Manager process is very low.

For the purposes of our investigation here, we run two simulations per benchmark: one of a "clean" run of the benchmark without the implemented framework, and one with the full-fledged DaemonGuard framework running on the system. A test daemon is loaded for the testing routine of each functional unit within each CPU core. Since we consider seven functional units per core of the 16-core system, 128 test daemons (including the 16 Test Daemons for full-core testing) are loaded in total. The testing routines are loaded, but not actually "executed," since we are only interested (at this point) in the performance overhead imposed by the proposed monitoring and test initiation framework, and not the actual testing time overhead. This simulation exercise shows that the overhead imposed by DaemonGuard over all benchmarks is near-negligible, at around 0.23%. Detailed analysis of the impact of DeamonGuard Framework considering the Selective SBST is presented in Section 4.5.2.

Regarding the impact of loaded Test Daemons, a Test Daemon is required for each level and each functional unit as DaemonGuard Framework supports testing at core level as well as at sub-core granularity (i.e. functional units). The number of loaded test *daemons* does not impact the performance overhead, since the daemons are always *idle* during normal operation. The cost incurred by the test daemons is only in terms of memory overhead. Since daemons are common loaded processes of the OS, they have a portion of the main memory allocated to them. However, the memory footprint is determined by the *number of unique test routines* to be executed, not the number of daemons running.

Considering the microprocessor architecture that we are using in this thesis, the core is divided in seven functional units that can be tested. Thus, the total number of test daemons required to be hosted in DaemonGuard Framework is eight, one for the full core testing and one for the seven test routines. The total memory footprint required for the accommodation of all the test daemons within the system is determined by these eight daemons. The simulation exercise that is done for the performance evaluation of DaemonGuard Framework indicates that the amount of memory required is 712 KB, on average, per unit-specific test routine and about

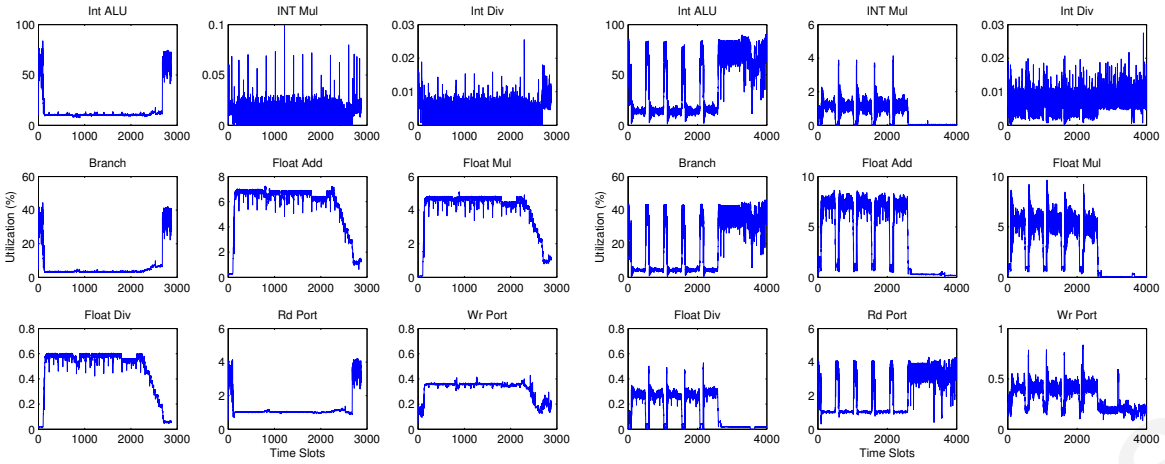
3 MB for the full-core Test Daemon. If multiple daemons execute the same test routine, then all these daemons will execute the same routine instructions from the same physical address space. For example, in a 16-core system, we would need one ALU test daemon per core, i.e., a total of 16 test daemons executing the same ALU test routine. Thus, the memory footprint for the ALU test routine would only be incurred once, *not* 16 times. This is precisely the reason why this daemon-based approach is scalable with the number of CPU cores.

3.4 DaemonGuard Framework: a Profiling Exercise

As we already mentioned, DaemonGuard Framework is an O/S based system with hardware support that mainly is used for the orchestration of SBST techniques for multi-core systems. The first utilization of DaemonGuard Framework in our work, is to implement the Selective SBST. Initially, we deployed part of the DaemonGuard Framework and specifically the instruction counters for each core of a multi-core system. Using this, we proceeded to a profiling exercise of workloads applied to the system in order to classify the executed instructions in classes based on their functional characteristics/functional units. Results of this profiling exercise motivates us to further investigate the utilization-based SBST and proposed the Selective SBST. In this section, the profiling exercise is presented.

Sub-core functional unit utilization statistics are derived through a series of full-system simulations over the PARSEC benchmark applications. For each benchmark, we periodically collect information pertaining to the number of executed instructions. We classify the instructions based on their type and the corresponding functional unit that is responsible for their execution. Figure 3.2 presents some of the results of our profiling experiments pertaining to CPU utilization during the execution of various multi-threaded benchmark applications. Note that the CPU utilization is observed at the granularity of individual functional units (i.e., at the sub-core level). In particular, for each of four PARSEC benchmarks, we present the percentage utilizations of each of the nine functional units (integer ALU, integer multiplier, integer divider, branch, floating-point adder, floating-point multiplier, floating-point divider, read-ports and write-ports) of a specific core over a period of 50 K cycles during the benchmark's execution (similar trends have been observed in all cores of the system).

This profiling exercise assumes a 16-core Chip Multi-Processor (CMP); the details and parameters of the evaluation framework are given in Section 4.5. The x-axis of each sub-figure in Figure 3.2 shows the benchmark's execution timeline, while the y-axis shows the percentage utilization of each intra-core functional unit. Figure 3.3 illustrates the utilization of each func-



(a) Blackscholes

(b) Bodytrack

(c) Fluidanimate

(d) Swaptions

Figure 3.2: Profiling experiments of CPU utilization during the execution of four PARSEC benchmark applications. The CPU utilization is measured at the granularity of individual functional units within each processing core. In particular, the percentage utilizations of each of the nine functional units (integer ALU, integer multiplier, integer divider, branch, floating-point adder, floating-point multiplier, floating-point divider, read-ports and write-ports) of a specific core in a 16-core CMP are presented over a period of 50 K cycles during the benchmark's execution.

tional unit when all PARSEC benchmarks are executed sequentially, one after the other. This experiment aims to show how unit utilization varies as the application type changes over time (since different applications are executed sequentially). The averaging period in the results of Figure 3.3 is 1 minute, in order to observe functional unit utilization over a more practical interval (i.e., substantially longer than the 50 K cycles of Figure 3.2). The entire duration of the sequential execution of all PARSEC benchmarks is more than 2 minutes, so Figure 3.3 presents results for two averaging periods: one for the first minute of execution (T=0–1 mins), and one for the second minute of execution (T=1–2 mins).

The overall outcome of this profiling exercise is that despite the well balanced distribution

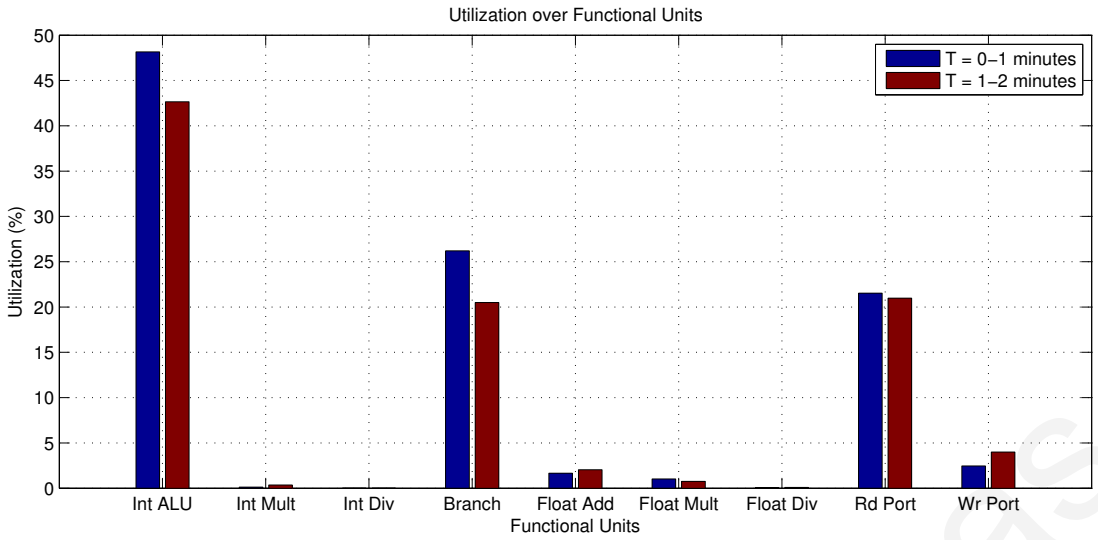


Figure 3.3: Utilization of each functional unit when all PARSEC benchmarks are executed sequentially, one after the other. The averaging period is 1 minute. Two averaging samples are shown: one for the first minute of execution (T=0–1 mins), and one for the second minute of execution (T=1–2 mins).

of workload over the cores of the system the distribution at sub-core granularity, over the functional units, is non-uniform. This motivates us to investigate the utilization-based self-testing at the granularity of functional unit, Selective SBST.

Chapter 4

Selective SBST for Shared-Memory Multicore Systems

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

4.1 Introduction

Existing SBST mechanisms for multi-core systems [12–14] periodically test either an entire CPU *core*, or – in more aggressive scenarios – the entire CPU (i.e., all CPU cores). Naturally, this methodology may lead to *over-testing*, since all core modules (e.g., integer ALU, floating-point units, etc.) are tested irrespective of the actual strain they have suffered since the previous testing session. Ideally, one would want to perform selective/targeted testing to the units that have sustained the most strain, and only sporadic testing to under-utilized modules. As the number of cores in multi-core microprocessors increases, the probability that various functional units within the individual cores will be underutilized during certain periods of time increases. Hence, conducting tests at the granularity of entire cores, or CPUs, will increasingly lead to unnecessary testing and associated performance overhead.

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

Toward this end, and in an effort to facilitate seamless and transparent selective SBST in multi-core systems, we hereby propose an adjustment of the DaemonGuard Framework to support Selective SBST. In particular, DaemonGuard Framework enables the real-time observation of individual sub-core modules and initiates on-demand selective (i.e., unit-specific) testing. By looking inside each core, DaemonGuard performs more frequent testing of over-utilized functional units and periodic, infrequent testing of under-utilized units. It should be noted that the DaemonGuard framework does not impose any restrictions on the metric used to trigger the testing mechanism. *Utilization* is used here as a proof-of-concept surrogate targeting the Hot Carrier Injection (HCI) failure mechanism, which is directly related to the switching frequency and activity factor of the components [60,61]. In reality, *any* metric may be adopted to account for any form of aging/wear-out. For example, if the underlying architecture provides temperature sensors [62], or any other type of aging sensors, DaemonGuard may utilize them for more accurate estimation of aging. Performance counters may also be used as a proxy for high temperatures, as demonstrated in [63].

In order to assess the advocated new testing methodology, we fully implement and evalu-

ate DaemonGuard in a full-system simulation framework based on Simics [64]/ GEMS [65], running a commodity operating system and executing the PARSEC benchmark suite [66] (a selection of multi-threaded applications) in a multi-core setup. We juxtapose the proposed selective testing procedure to two full-core testing approaches: (a) full-core testing triggered by total core utilization, and (b) full-core testing triggered by individual sub-core unit utilizations. Instead, DaemonGuard only performs unit-specific tests to the units that exceed a certain utilization threshold. The results of our experiments indicate savings in testing time of up to 30× when testing is performed in a selective manner. Moreover, the impact on system performance of the always-on Testing Manager process is shown to be negligible, thus corroborating our assertion that OS-assisted SBST is a viable option. The latter also demonstrates the potential scalability of this approach to large-scale many-core systems.

This work was published and presented in a peer-reviewed conference [**?**] and published as a journal paper [49].

4.2 Related Work

Several non-concurrent on-line testing techniques have been proposed in the literature [5]. SBST or functional testing is a promising solution for periodic testing of traditional microprocessor architectures [18]. Recently, the advent of the multi-/many-core era has spurred a series of techniques targeting the testing of a system at the core and system levels. When applying such methodologies in microprocessors, the main source of overhead is *testing time*. One of the main objectives of these techniques is to reduce the testing time overhead.

Constantinides et al. [51] proposed an online testing methodology using an enhanced ISA with special instructions for fault detection and isolation. Structural tests are performed by applying test patterns using software routines. The test routines are executed periodically, after a number of executed instructions have committed, and checkpoints are used for recovery. The technique of Constantinides et al. is software-*assisted*, but it requires various hardware modifications. These intrusive modifications are needed, because the goal is to enable very detailed structural testing through existing scan-chain infrastructure. On the contrary, the proposed DaemonGuard approach targets *functional* testing, which is purely software-based, and only employs regular instructions that are part of the processor's ISA.

A hardware and software co-design methodology for functional testing is proposed in [52]. The testing methodology is based on the redundancy concept, whereby two cores execute the same program and capture corresponding footprints. The results of the executions are com-

pared for fault detection. The choice of the test program is based on profiling that can be done offline or online. In [53], the authors propose a thread relocation methodology that uses dynamic profiling based on phase tracking and prediction.

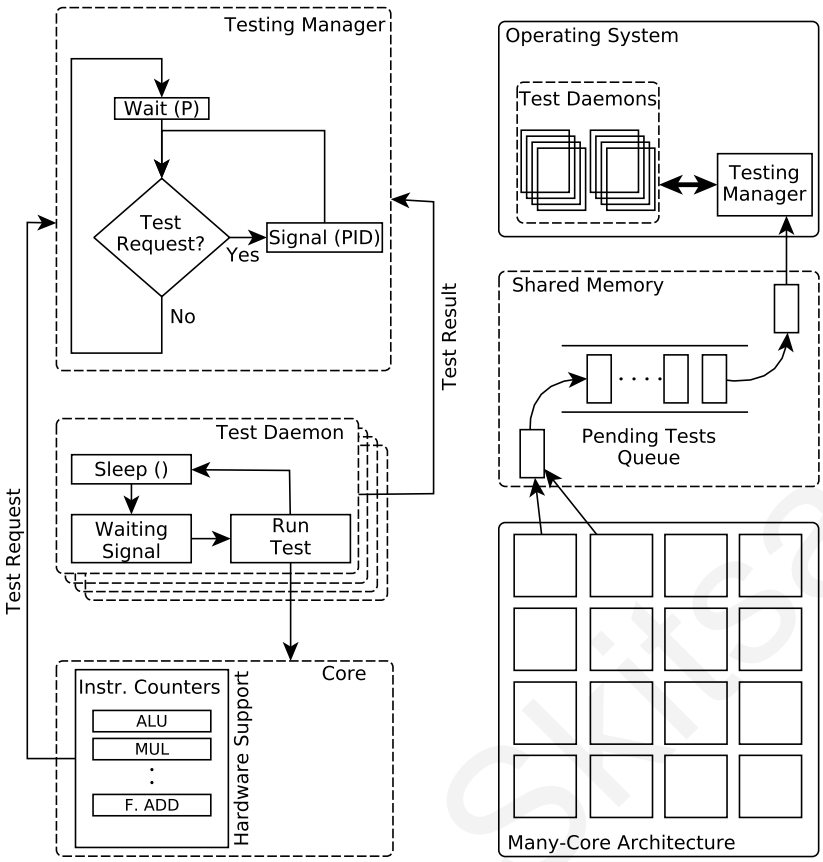
Apostolakis et al. [12] propose a scheduling methodology for the test routines, aiming to reduce the test execution time by exploiting core-level parallelism. A Multi-Threaded (MT) SBST methodology is proposed in [14]. The authors propose an efficient MT version of functional unit test programs, in order to reduce the execution time of testing. All test routines are run simultaneously, based on the thread-level parallelism capabilities of the core under test. In all cases, testing is assumed to be initiated periodically (at regular time intervals) and is performed for the entire core. Note that the proposed DaemonGuard scheme is orthogonal to MT SBST, and, thus, it may be used to complement such methods.

Yanjing Li et al. [59] demonstrate the need for OS support in efficiently orchestrating online self-testing in future robust systems. Their contribution is the development of test-aware OS scheduling techniques for multicore systems. These techniques take into account the availability of each core when deciding the initiation of full-core testing, with the ultimate goal being to reduce the performance impact of the testing procedure. Test-aware OS scheduling is orthogonal to the work presented in this article and can, in fact, complement the DaemonGuard framework. In other words, while the current incarnation of DaemonGuard does not modify the OS scheduler, it is conceivable that a more holistic approach could allow the DaemonGuard's Testing Manager to coordinate with the OS scheduler.

A recent test-scheduling study for online error detection in multicore systems is discussed in [45]. The authors evaluate the performance of test programs applied on Intel's 48-core Single-chip Cloud Computer (SCC) architecture. Due to possible congestion within common hardware resources used by the various cores, the test time can be quite large with a significant impact on performance. As a result, the authors of [45] develop effective test scheduling algorithms to expedite the test process in such systems.

4.3 DaemonGuard Framework for Selective SBST

In order to enable *selective* testing – whereby the execution of unit-specific test routines is initiated on demand at run-time – it is essential to be able to dynamically track the utilization of the functional units within each CPU core. This was the initial motivation for the implementation of DaemonGuard, a light-weight, minimally-intrusive framework, which transparently orchestrates the procedure of selective testing. In Selective SBST the DaemonGuard Framework is



(a) The three main components of the DaemonGuard framework.

(b) The flow diagram of DaemonGuard's operation.

Figure 4.1: Architectural Overview of the DaemonGuard Framework for selective SBST. The three main components of DaemonGuard work in unison in order to facilitate real-time monitoring of the utilization of the various functional units within each core of the multi-core system. Once a unit exceeds a certain number of executed instructions, a test request is generated. Accordingly, the Testing Manager OS process invokes the appropriate OS-resident test daemon to initiate the execution of the unit-specific test routine.

responsible to (a) monitor the system activity during normal operation, (b) initiate the execution of the appropriate test routines on the appropriate cores, and (c) collect the test results. Details about the implementation of DaemonGuard Framework have been presented in Chapter 3. In this chapter we provide technical details of the components of DaemonGuard Framework that are adapted and/or extended in order to support the Selective SBST.

The three main components of DaemonGuard Framework for the Selective SBST are illustrated in Figure 4.1(a). The top two components (the Testing Manager OS Process and the Test Daemons) are implemented purely in software and reside within the OS. The bottom component in Figure 4.1(a) shows the minimal hardware support (the Instruction Counters for the various functional units within each CPU core) required to provide utilization information to the Testing Manager.

4.3.1 Daemon-Based Selective SBST - Test Daemons

In order to perform selective SBST, a number of test daemons (as described in Section 3.1.1) are loaded onto the OS. The number of loaded test daemons depends on both the number of CPU cores present in the system, and the number of functional units within each core. DaemonGuard requires one test daemon per functional unit per core. As previously mentioned the memory footprint is *not* affected by the number of daemons, but by the number of *unique* unit-specific test routines. Daemons are kept in idle mode during normal operation; they wait for the appropriate invocation signal from the Testing Manager OS process, in order to wake up and execute their specified unit-specific test routine on the targeted functional unit of a specific core. The flow diagram of the operation of a test daemon is shown in the middle box of Figure 4.1(a). Upon completion of the testing process, the outcome (i.e., if a fault has been detected or not) is reported to the Testing Manager process.

4.3.2 The *Testing Manager* O/S Process

The main part of the DaemonGuard framework is the Testing Manager process that is responsible for the invocation of the various test daemons, based on the utilization information provided by the hardware instruction counters residing alongside each functional unit within the CPU cores. The main function of the Testing Manager is the periodic checking of pending test requests by any functional unit of any core within the system. Since our aim was to present a proof-of-concept implementation for the DaemonGuard framework, a polling-based approach was employed for simplicity. It should be noted that an interrupt-based implementation would be more efficient. Regardless, the results show that even the polling-based approach incurs near-zero overhead. Irrespective of the implementation details of the Testing Manager, the main goal of this chapter is to demonstrate the potential benefits of selective testing. The test requests are stored into a Pending Tests Queue structure, as depicted in Figure 4.1(b). This memory-mapped software structure is located in shared memory, where every core of the system has direct access. This means that the Testing Manager can run on any available core of the system and it can still have access to the queue.

During each checking period, the Testing Manager checks the queue, it dequeues all pending test requests, and it sends a wake-up signal to the corresponding test daemons. The signal, in this case, is an inter-process communication message and it is facilitated by the OS. Due to this OS-assisted functionality, the proposed testing methodology is minimally intrusive at the system level, and the test programs can run "simultaneously" with the normal applications,

using OS context switching. In fact, depending on the running application, the OS may sometimes be able to completely "hide" the execution of the test daemons. Upon completion of a test session, the corresponding test daemon sends the test result back to the Testing Manager. The complete flow diagram of the DaemonGuard's operation is illustrated in Figure 4.1(b). The high-level pseudo-code of the Testing Manager is given in Algorithm 1. The checking period of the Testing Manager is decided a priori and remains constant during the normal operation of the system.

Algorithm 1 Testing Manager Process for Selective SBST. Periodically checks for pending test requests and invokes test deamons for execution

Input: Period P, Pending Tests Queue (PTQ)

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

4.3.3 Hardware support

The proposed selective testing methodology relies on the run-time gathering of information regarding the utilization of the various functional units within each core of the system. The term "utilization" refers to the number of instructions that have made use of the specific unit during normal operation. In order to collect this data, we assume the presence of a set of hardware instruction counters. There is one such counter for each functional unit within each core. For example, if an executed instruction uses the ALU, the corresponding counter will increase by one. The value of each counter is then used to determine when a functional unit must request a test. When the predefined threshold of T instructions is met, the affected unit places a test request in the Pending Tests Queue of Figure 4.1(b). The Testing Manager process will then invoke the appropriate daemon (e.g., the ALU test daemon) to execute the unit-specific test routine on the corresponding core. After the execution of the test routine finishes, the specific

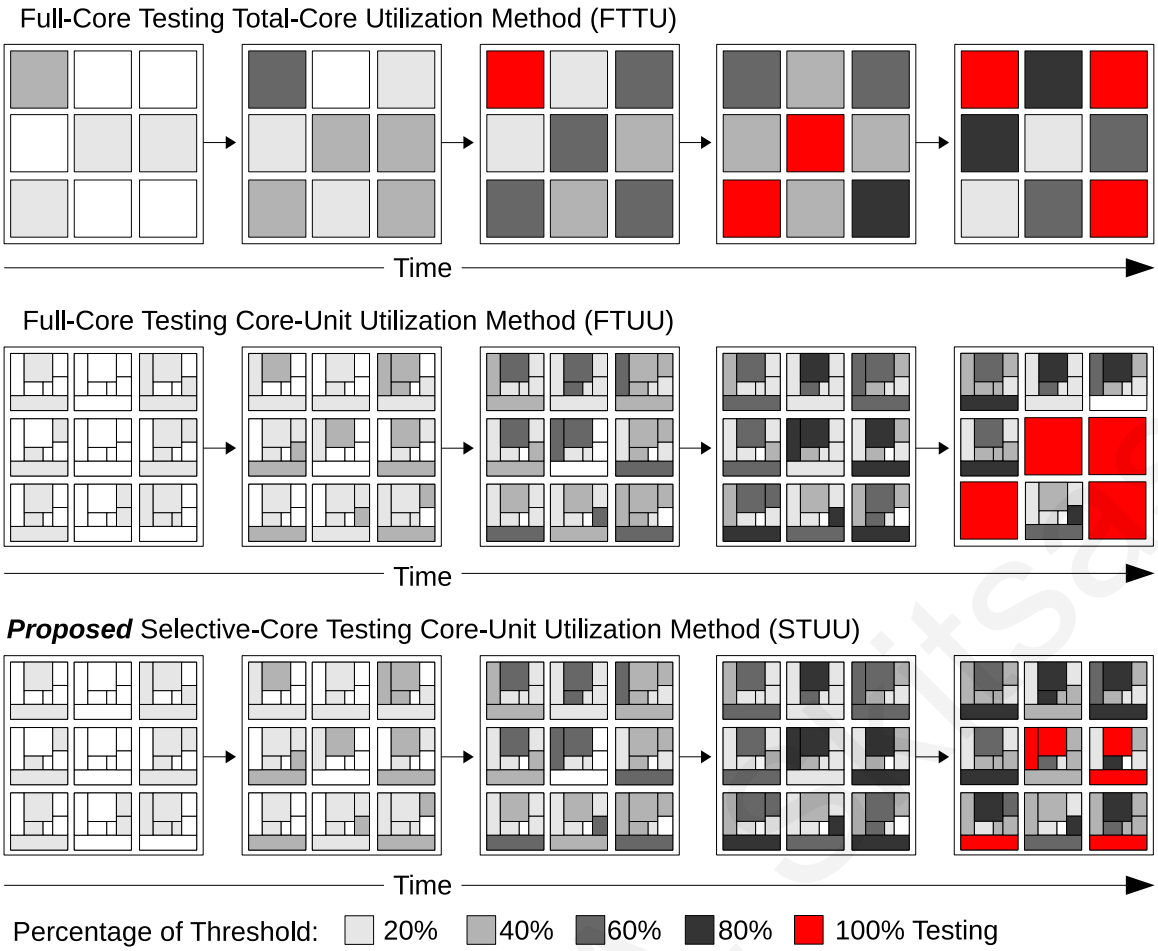


Figure 4.2: Abstract illustration of the differences between the three examined testing methodologies, FT-TU, FT-UU, and ST-UU. The stress on each core or unit (i.e., number of instructions executed) is represented by the color intensity, as a percentage of the threshold required to trigger testing. Red squares/rectangles are cores/units under test.

counter will receive a reset signal from the Testing Manager process to restart counting from zero.

4.4 Proposed selective testing based on functional-unit utilization

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

The motivation of utilization-based self testing is derived by the profiling exercise of Sec-

tion 3.4. We have derived sub-core unit utilization statistics through a series of full-system simulations over the PARSEC benchmark applications [66]. For each benchmark, we periodically collect information pertaining to the number of executed instructions. We classify the instructions based on their type and the corresponding functional unit that is responsible for their execution. The profiling results show that the distribution of the instructions over the various units *within each core* is non-uniform, mainly due to the nature of the running application. This motivates us to explore the potential of *selective testing* at the sub-core granularity by applying appropriate test routines to only the strained units and, thus, reducing the overall testing time overhead.

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

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

A directly comparable testing methodology is the second full-core testing methodology used in our exploration, the *Full-Core Testing based on Core-Unit Utilization (FT-UU)*. Here, the system tracks the number of executed instructions on each functional unit within each core, similar to the proposed DaemonGuard methodology. When the functional unit with the highest utilization within a core exceeds the testing threshold (i.e., the unit executes more instructions

Table 4.1: Simulated system parameters.

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

than the threshold T), the OS invokes a test daemon that tests the *entire* core. Note that after the execution of the full-core testing process, all unit counters will restart counting from zero.

Figure 4.2 illustrates abstractly the differences between the three examined testing methodologies. Let us assume that we have a 9-core CMP. In the case of the FT-TU testing scheme (top part of Figure 4.2), the CMP only observes the number of instructions executed in each core, so the minimum granularity of observation is a single core (represented by each small square in the 9-core CMP). The stress on each core (i.e., number of instructions executed) is represented by the square's color intensity, as a percentage of the threshold required to trigger testing. For example, a light grey core indicates low stress, and as the threshold utilization target is approached (i.e., closer to 100%), the color becomes darker. Once the threshold target is hit, the core transitions to testing mode (indicated by the red color). As time progresses, more squares (cores) become darker in color (i.e., total-core stress increases), and various cores transition into testing mode. The second testing methodology, FT-UU (middle part of Figure 4.2), observes core stress at a sub-core granularity, as indicated by the smaller rectangles, which represent individual functional units within each core. However, testing is still performed on the entire core once the threshold of a functional unit is exceeded. Finally, the proposed ST-UU methodology (bottom part of Figure 4.2) conducts selective testing only on the functional units that exceed their stress threshold. As evident by the smaller red rectangles in the bottom row of Figure 4.2, the proposed methodology provides fine-grained testing and, thus, it avoids overtesting. Even though the same workload is executed in all three cases, the *test-trigger time* and the *test target* are different, based on the methodology employed. The proposed ST-UU approach provides finer-granularity observations and finer-granularity testing sessions.

Table 4.2: The execution times and memory footprints of the employed unit-specific test programs.

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

4.5 Experimental Framework and Results

4.5.1 Evaluation framework

For the evaluation of the proposed testing methodologies, we use a full-system, execution-driven simulation framework based on the Wisconsin GEMS toolset [65], in conjunction with Wind River's Simics [64]. We simulate a 16-core CMP, as presented in Table 4.1. Each core in the system is an in-order-execution UltraSPARC III+ core.

Our approach of selective testing requires one test daemon for each functional unit of each CPU core. We first use the test routines from [14], which is the most recent work that developed testing routines at the functional-unit level. In order to use these routines in our framework, we implemented them as daemons and integrated them within the OS, as descriped in previews sections. These test routines target the OpenSPARC microprocessor [67], which uses the SPARC v9 ISA. The UltraSPARC III+ core in our evaluation framework also uses the SPARC v9 ISA, which means that the test routines of [14] are applicable to our system. The execution times, number of instructions, and memory footprints of these routines are given in Table 4.2. In terms of execution time, two sets of times are reported: one for a microprocessor with a perfect memory system (i.e., without considering any memory latencies), and one for a realistic system employing the memory hierarchy described in Table 4.1. The two sets of execution times are shown to identify the test routines that experience elevated cache misses in the realistic system (i.e., the system used in our simulations).

Full-system, execution-driven simulations are performed using the PARSEC benchmark suite [66]. PARSEC is a benchmark suite of multi-threaded workloads that focus on emerging

Table 4.3: Details of the PARSEC benchmark applications used in our evaluation framework.

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

parallel workloads. We use PARSEC benchmarks for our extensive profiling of the distribution of instructions over the units and the cores of the system. To evaluate the proposed testing methodologies, we use eleven of the benchmarks. Details about the input sizes, number of executed instructions, and execution times are given in Table 4.3. All the benchmarks are configured with 16 threads, since the multi-core setup in our evaluation framework consists of 16 cores.

4.5.2 Impact of the DaemonGuard Framework

As DaemonGuard Framework is mainly based on the execution of processes by the O/S concurrent with the normal workload, a series of experiments are performed in order to investigate the impact of DaemonGuard on overall system performance. It will be demonstrated that the cost of employing an always-active Testing Manager process is very low.

Table 4.4 presents the results of this exploration. The parameters of the simulated machine are shown in Table 4.1 and results are collected when running the PARSEC benchmark applications [66]. In order to speed up the simulation time, the memory system is not considered here, because no test routine is actually executed (which would require a memory access), i.e., the memory system does not affect the experiment under consideration. Note that the simulation experiments for the evaluation of Selective SBST include a complete and realistic memory hierarchy, as will be explained later on. For the purposes of our investigation here, we run two simulations per benchmark: one of a “clean” run of the benchmark without the proposed framework, and one with the full-fledged DaemonGuard framework running on the system. A

Table 4.4: Impact of the DaemonGuard Framework on System Performance, $P = 750K$ cycles.

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

test daemon is loaded for the testing routine of each functional unit within each CPU core. As it can been seen in the table, the overhead imposed by DaemonGuard over all benchmarks is near-negligible, at around 0.23%.

The incurred overhead is attributed to the always-on Testing Manager process, which periodically checks for pending test requests. A significant parameter within the Testing Manager is the period P over which pending test requests are monitored; i.e., period P is the time (in cycles) between each check of the Testing Manager for pending test requests. Period P is determined by the threshold T of committed instructions that a functional unit must exceed in order to initiate testing. This threshold is typically determined by the underlying technology, the criticality of the system, and the application of on-line self-testing. For example, for circuit failure prediction [8, 9, 68], periodic on-line self-testing can be performed less frequently – say once every 10 secs – than in the case of hard failure detection, which can occur as often as once per second, in order to allow for low-cost and low-latency recovery [59]. In our framework, a testing request is initiated by the core, based on workload utilization, when the threshold T is reached. Hence, the monitoring of pending test requests must occur frequently, so as to enable timely detection of test requests. The relationship between P and T is expressed as $P << \tilde{T}$, where $\tilde{T}$ is the estimated time in cycles required to commit T instructions.

For the experiments of Table 4.4, the period P is set to 750K, so the Testing Manager checks for pending test requests every 10 ms. The overhead, in terms of instructions, each

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

4.5.3 Evaluation results of Utilization-Based Selective SBST

We simulate all aforementioned benchmarks according to the testing methodologies described in Section 4.4, using the proposed DaemonGuard framework and the test routines of Table 4.2. Simulations are performed in order to evaluate the benefit of the proposed selective testing policy, ST-UU, as compared to the two full-core testing policies, FT-TU and FT-UU.

One key issue is the selection of an appropriate *testing threshold* T, i.e., the number of executed instructions that trigger the initiation of a test session. As already mentioned, T will be defined based on technology parameters related to aging and wear-out effects, the criticality level of the system, as well as the considered application of on-line self-testing (e.g., failure prediction vs. failure detection). First, we study how different values of T impact the proposed methodology. Specifically, we evaluate the testing-time overhead when modifying the testing threshold. Figure 4.3 shows the results when applying different threshold values. In particular, we performed simulations using threshold values T of 5M and 10M instructions for all the considered benchmarks. The three figures correspond to the three testing methodologies under evaluation. The x-axis of each graph shows the 11 benchmarks and each pair of bars corresponds to the two investigated threshold values. The last group of bars on the x-axis refers to the average results for all the benchmarks. The y-axis shows the extra overhead – in terms of cycles – incurred by each method due to testing. As it can be seen, the overhead is drastically reduced as the threshold increases, since the number of testing sessions also decreases. On average, doubling the threshold yields a reduction in testing overhead of about 40% in all three testing methodologies. This result highlights the significance of the testing threshold: a very low threshold will result in excessive testing and unnecessary stressing of the functional units by the test routines themselves. On the other hand, setting the testing threshold too high may result in late detection of faults.

For the remainder of our experiments, we use $T = 5M$ instructions. In practice, T may

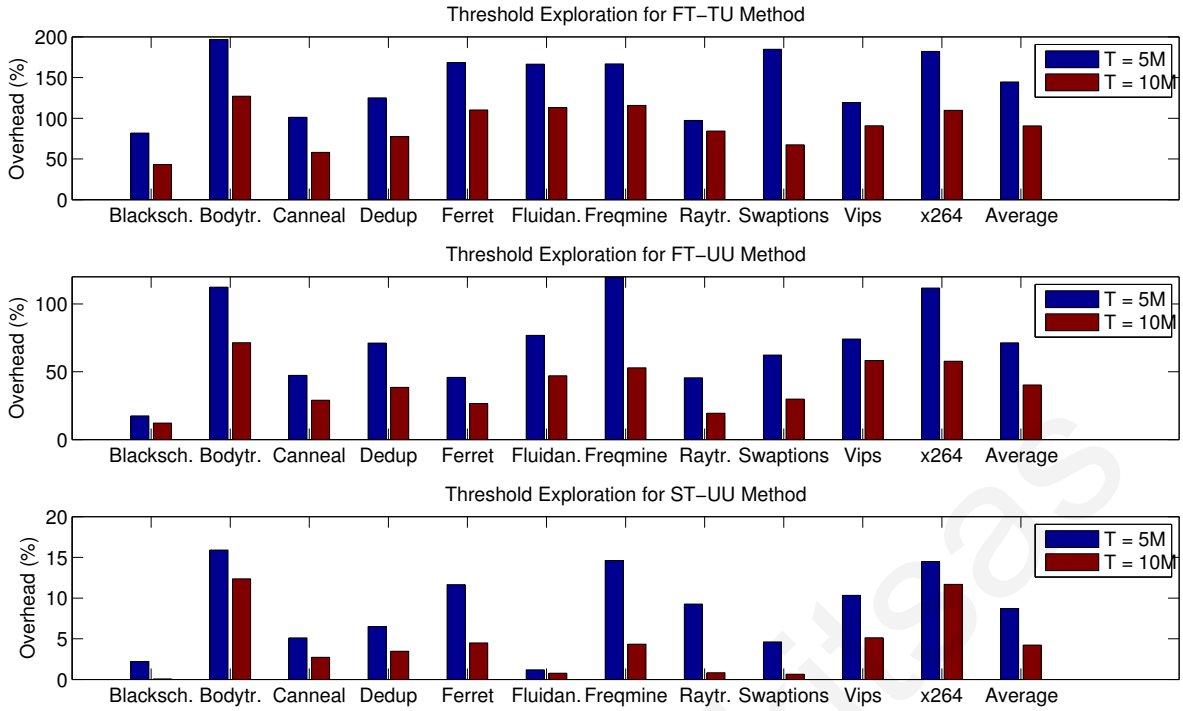


Figure 4.3: Exploring the impact of the *testing threshold T* (the number of executed instructions required to trigger a testing session) on the execution time of the various benchmark applications. The three graphs correspond to the three testing methodologies under evaluation: FT-TU, FT-UU, and the proposed ST-UU (bottom graph).

be larger; however, we limit its value here, in order to be able to invoke the various testing policies an adequate number of times for comparison purposes, while maintaining reasonable simulation times in our full-system, cycle-accurate framework.

Since the main objective of this chapter is to reduce the testing time, we investigate the total execution time (application time plus testing time), in terms of cycles, and the total number of executed instructions in the presence of each testing methodology. Figure 4.4 presents the results over all the considered benchmarks. The top graph of Figure 4.4 presents the testing overhead in terms of total committed instructions under each testing policy, while the bottom graph shows the testing overhead in terms of extra clock cycles. Each triplet of bars represents the three testing methodologies and shows the testing overhead normalized to a system with DaemonGuard loaded but with no tests performed. The last triplet in both graphs shows the average results across all benchmarks. Clearly, the proposed selective testing approach (the right-most bar – ST-UU – in each triplet) yields a significant reduction in the testing cost, both in terms of extra executed instructions and extra cycles needed. On average, the testing overhead is less than 8% when applying the proposed selective testing approach (ST-UU), while full-core testing based on unit-utilization (FT-UU) imposes a 70% overhead.

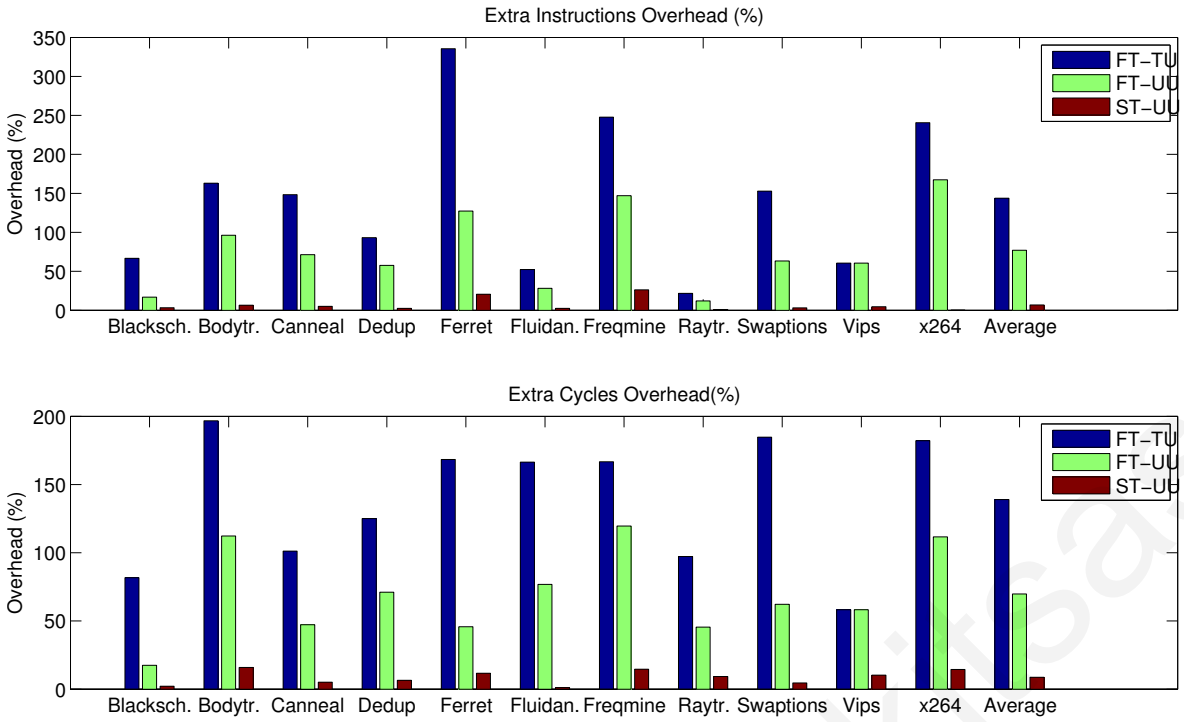


Figure 4.4: The testing overhead imposed by the three testing methodologies across all benchmarks. The top graph shows the testing overhead in terms of extra executed instructions, while the bottom graph shows the overhead in terms of extra cycles needed. The testing routines are executed together with the running applications, whenever a testing session is triggered.

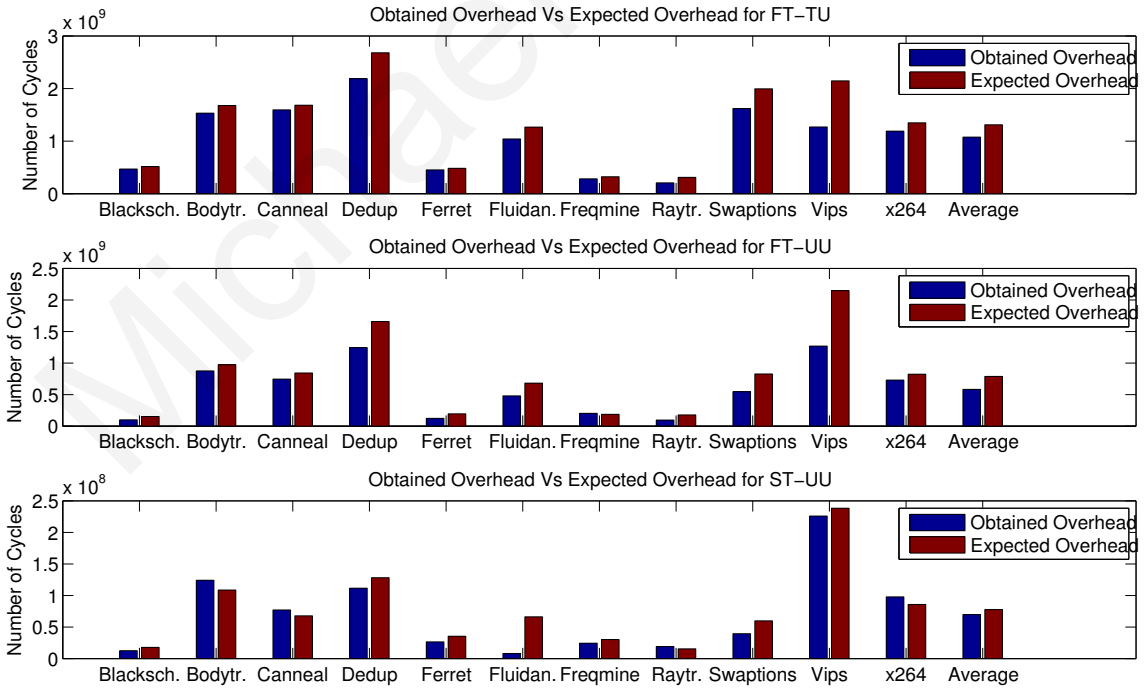


Figure 4.5: A comparison between the obtained testing overhead incurred – in terms of extra cycles needed – and the *expected* overhead. The expected overhead is estimated by multiplying the number of testing sessions performed with the number of cycles that each test routine requires when executed in isolation.

Expected Testing Overhead

We also compare the obtained testing overhead incurred – in terms of extra cycles needed – with the *expected* overhead. The expected overhead is estimated by multiplying the number of performed testing sessions (as reported from the simulations) with the number of cycles that each test routine requires when executed in isolation. The simulation results show that the actual (obtained) overhead is less than the expected, as illustrated in Figure 4.5. Once again, the three graphs correspond to the three testing methodologies under evaluation. The left bar in each pair of bars represents the obtained extra cycles needed due to testing, while the right bar indicates the calculated *expected* cycles. Observe that, in many cases, the obtained overhead is less than the expected. The reason for this discrepancy is attributed to the OS: the OS scheduler re-balances (re-distributes) the application workload during testing sessions, so as to optimize the use of resources. Consequently, testing time can be hidden by the OS's scheduling mechanisms. This is another benefit of the proposed daemon-based SBST methodology: by employing OS-assisted selective testing, we benefit from the inherent capabilities of the OS itself, in terms of scheduling and load-balancing optimizations.

4.6 Concluding Remarks

This chapter proposes Selective SBST, a new approach in SBST that monitors the system utilization at the sub-core granularity and initiates targeted testing of only the over-utilized functional units. Under-utilized units are only sporadically tested in a periodic manner. The proposed methodology is evaluated using full-system, execution-driven simulations over the PARSEC benchmark suite. The results incidate significant reductions in testing overhead of up to 30× when the Selective SBST is used instead of a full-core testing approach. Additionally, the results of the execution of larger test routines with higher demands of memory accesses indicates an increased overhead in terms of execution time. This motivates us, first to investigate the impact of memory hierarchy (i.e. LLC) and then to propose a mechanism in order to further reduce the imposed overheads.

Chapter 5

Cache-Aware Selective SBST

The era of nanoscale technology has ushered designs of unprecedented complexity and immense integration densities. Billions of transistors now populate modern multi-core microprocessor chips and the trend is only expected to grow. Diminutive feature sizes, however, put undue strain on the reliability and long-term endurance of these modern systems. Research by both industry and academia is pointing to the alarming fact that future designs will be increasingly vulnerable to aging and wear-out artifacts. In this chapter, we proposed an improvement of Selective SBST by tracking the testing activity and providing a test data pre-fetch mechanism aiming to further reduce the imposed testing time overheads. In particular, we investigate the impact of the cache hierarchy on the testing process and we develop a cache-aware selective testing methodology that significantly expedites the execution of memory-intensive test programs. We perform a series of experiments using a full-system, execution-driven simulation framework running a commodity operating system, real multi-threaded workloads, and test programs. Our results indicate that the cache-aware testing technique is very effective in exploiting the memory hierarchy to further minimize the testing time.

5.1 Introduction

The proposed Utilization-Based Selective SBST of Chapter 4 seems to be a promising SBST method which aims to reduce the testing time overheads by avoiding the over-testing of under-utilized units. Based on the proposed Selective SBST and the DaemonGuard Framework (described in previous chapters), in this chapter, we proceeded to an enhancement by proposing the Cache-Aware Selective SBST. The DaemonGuard mechanism is able to exploit the memory hierarchy of the CPU to expedite the testing process. While some test programs may be

small and could fit in their entirety within the L1 cache, there are also test programs that are much larger (on the order of MB) and, therefore, tend to stress the system's memory.

In general, on-line self-testing may require thorough testing to account for a variety of failure modes, such as traditional stuck-at and delay faults. In such cases, it may be necessary to load a considerable amount of test patterns from off-chip memory. Given the trend toward many-core microprocessors with ever-increasing hardware complexity, the memory footprint of such SBST programs is also likely to increase. In this chapter, we investigate the impact of the cache hierarchy on the testing time of memory-intensive test programs and demonstrate that the testing coordinator (the *Testing Manager* in our case) cannot be oblivious to the underlying memory sub-system. Hence, the DaemonGuard Framework for Selective SBST is augmented with the capability to perform *cache-aware* selective testing, whereby test sessions are initiated not only based on unit utilization, but also on the recent history of test sessions by other similar units in other cores. Consequently, test programs can benefit from cache-resident blocks, thereby obviating the need for many expensive off-chip memory accesses.

In order to assess the new testing methodology, similar with Selective SBST, we fully implement and evaluate DaemonGuard Framework in a full-system simulation framework based on Simics [64]/ GEMS [65], running a commodity operating system and executing the PARSEC benchmark suite [66] in a multi-core setup. For the evaluation, we compare the testing time overhead of the cache-aware Selective SBST with the non-cache-aware from the Chapter 4. Additionally, in order to assess the efficiency of the proposed approach, we compare the results with the ideal scenario where all the data are located in the L1 cache of the core that functional unit under test belongs. The results of our experiments indicate that the cache-aware selective testing capabilities of DaemonGuard are shown to substantially decrease the execution time of memory-intensive test programs, thus minimizing the testing overhead and its impact on overall system performance.

The work of this chapter has been peer-reviewed and published as a journal paper [49].

5.2 Cache-Aware Selective SBST

The DaemonGuard framework is designed to initiate testing based on the individual unit utilizations. While this philosophy is extremely beneficial in terms of providing fine-grained testing (which is faster and avoids overtesting), the framework seems to ignore a fundamental aspect of the system: the memory hierarchy. The memory hierarchy comes into play when the test programs become large enough that they start to stress the memory sub-system. Some of the

test programs used in this thesis are small enough to fit entirely within the L1 cache of any individual core in the CMP. Hence, other than the unavoidable compulsory misses, those test programs will not experience any other cache misses throughout their execution. However, there are other test programs with larger working sets, which cannot fit within the cache hierarchy. These test programs are *memory-intensive*; they may experience large numbers of cache misses, and, consequently, suffer from very expensive (in terms of wasted stall cycles) off-chip memory accesses.

Moreover, the trend towards larger-scale and more complex (e.g., heterogeneous) CMPs implies an accompanying increase in the size of some of the test programs used in SBST schemes. Some researchers have already reported fairly large (in terms of memory footprint) test programs [3, 42], and it is not unreasonable to expect further increases in the memory footprints of various test programs in the near and distant future. It is, therefore, imperative to investigate the impact of the cache hierarchy on the testing time of memory-intensive test programs, in an effort to devise ways to decrease the execution time of such large test programs.

Motivated precisely by this need to further contain the testing time, we augment the DaemonGuard framework with the capability to be *cache-aware*. Selective testing can now be initiated not only based on unit utilization, but also on the recent history of test sessions by other similar units in other cores. Our simulation results indicate that test-related data (both test patterns in the data segment and instructions in the text segment) remains cached in the hierarchy for a substantial period of time after the test session concludes its execution. This period can be viewed as a *window of opportunity* for other cores to exploit. Remember, that the data required by a test daemon is shared between all the loaded daemons targeting the same functional unit in the various cores. Hence, if the *Testing Manager* could observe the recent testing history on all cores, the DaemonGuard mechanism would be able to exploit the memory hierarchy of the CPU to re-use cache-resident blocks, thereby limiting the need for time-consuming off-chip memory accesses. As a result, the overall execution time of memory-intensive test programs would be significantly reduced. Figure 5.1 presents an abstract example of how cache awareness can benefit the testing process. The top portion of Figure 5.1 illustrates the *worst-case* (in terms of testing overhead) scenario: Core 1 conducts a test session for a particular functional unit; Core 2 conducts the same test for the same type of functional unit (situated in Core 2), but Core 2's test commences at a time beyond the window of opportunity. In this figure, the window of opportunity is denoted as the time between points B and C on the timeline. This worst-case scenario occurs when all the test instructions and data have been evicted from the cache, i.e., it corresponds to a cold run yielding the longest possible test execution time. The

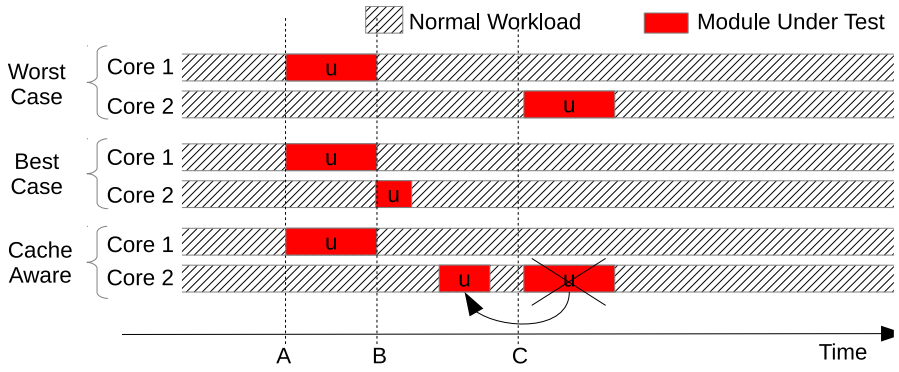


Figure 5.1: An abstract example of how cache awareness can benefit the testing process. The goal is to take advantage of the so called *window of opportunity*, which is a limited period of time after the testing session of a particular type of functional unit in one core in the CMP. During this time window (designated by the time between points B and C on the timeline in this figure), other cores wishing to test the same type of functional unit could take advantage of cache-resident blocks from the other core's testing session.

middle portion of Figure 5.1 illustrates the *best-case* scenario: Core 2's test commences immediately after the end of Core 1's test (or, it could also overlap with Core 1's test); thus, Core 2's test benefits from cache-resident blocks and the total time needed to complete the test session is minimized. The worst- and best-case scenarios constitute the performance *bounds* pertaining to the impact of the cache on the testing time. Finally, the bottom part of Figure 5.1 depicts the solution proposed in this chapter, which triggers earlier testing in Core 2 (i.e., the test is moved within the window of opportunity), in order to take advantage of cache-resident blocks from Core 1's testing session. Obviously, the proposed methodology would fall somewhere between the worst- and best-case performance bounds, as only those tests falling outside but close to the window of opportunity will be triggered earlier, in order to benefit from cache-resident blocks.

Based on this premise, we propose a new cache-aware selective methodology, where we consider *both* the stress on each individual unit *and* the recent history of test sessions targeting the same unit in other cores. In the baseline selective method (i.e., the non-cache-aware one), each core is responsible to request a test when the number of executed instructions on a specific functional unit exceed a threshold T. In the cache-aware version, the cores are no longer responsible to request a test. Instead, every T/d executed instructions on a particular unit of a particular core – where d is a constant integer and determines the granularity of threshold sub-divisions – the core updates the corresponding unit's status by sending a signal to the Testing Manager. This is achieved using a Status Update Queue (SUQ) situated in main memory and shared by all cores. In other words, the signal sent by each core for each of its functional units simply indicates that T/d instructions have been executed on the particular unit since the previous status update. The signal also includes a timestamp, the purpose of which will be

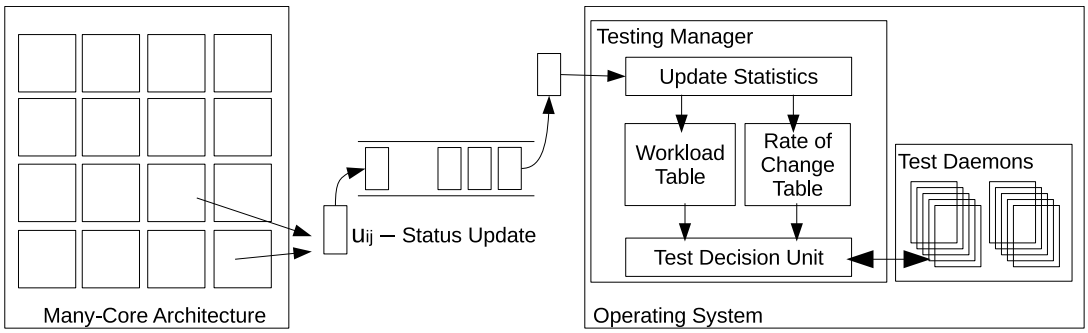


Figure 5.2: A high-level overview of the enhanced – *cache-aware* – DaemonGuard framework. The new structures employed are the *Status Update Queue* (situated in shared main memory) and two statistics tables, the *Workload Table* and the *Rate-of-Change Table* (both residing in main memory, within the address space of the Testing Manager itself).

explained shortly. Figure 5.2 presents a high-level overview of the enhanced – cache-aware – DaemonGuard framework. Thus, each core c_i in the CMP inserts individual status updates for each of its functional units u_j in the Status Update Queue. These update signals are indicated by u_{ij} in Figure 5.2. The coordination of the testing procedures on the entire CMP is now performed by the Testing Manager module, which is modified to support and orchestrate the cache-aware selective testing scheme. Algorithm 2 presents the high-level pseudo-code of the cache-aware Testing Manager OS process.

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

$$K_{ij} = K_{ij} + 1/d \tag{5.1}$$

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

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

Algorithm 2 Testing Manager (Cache-Aware)

Input: Period P, Status Update Queue (SUQ)

```
1:  while True do
2:      while SUQ not Empty do
3:          u_ij ←SUQ.dequeue()
4:          Update Workload Table(u_ij) /* K_ij (Eq. 1) */
5:          Update Rate of Change Table(u_ij) /* R_ij (Eq. 2) */
6:          EC_ij = (1 − K_ij) × T/R_ij /* (Eq. 3) */
7:          if (u_ij has exceeded T) or (EC_ij < L) then
8:              SendSignal(unit(u_ij)) /* trigger test */
9:          end if
10:     end while
11:     while Available Results do
12:         CollectResult(daemon)
13:     end while
14:     Sleep(P)
15: end while
```

$$R_{ij}^{t} = \frac{T/d}{C_{ij}^{t} - C_{ij}^{t-1}} \tag{5.2}$$

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

The Testing Manager (running as a process within the OS of the multi-core system) uses three pieces of information: (a) data from the two above-mentioned statistics tables, (b) the type(s) of units that have just exceeded their threshold T, and (c) all units of the same type(s) that have recently completed their testing session(s). By combining this valuable history information, DaemonGuard becomes cache-aware and triggers the appropriate testing daemon(s). A test daemon for some unit u_{ij} is triggered under any of the following two conditions: (i) threshold T has been reached at unit u_{ij} (this information is contained in the Workload Table),

or (ii) the estimated expected number of clock cycles remaining until the next testing session of unit u_{ij} – denoted by EC_{ij} – is less than L. The term L is the number of cycles required by a functional unit in the system to execute 10% of the instructions (a value chosen empirically here, without loss of generality) of its testing threshold T. Recall that the testing threshold T is a particular number of executed instructions. EC_{ij} is an estimate of the expected remaining time before the threshold of unit u_{ij} is reached, and it is calculated by:

$$EC_{ij} = (1 - K_{ij}) \times T/R_{ij} \tag{5.3}$$

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

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

5.3 Experimental Framework and Evaluation

5.3.1 Evaluation framework

For the evaluation of the proposed testing methodologies, we use a full-system, execution-driven simulation framework based on the Wisconsin GEMS toolset [65], in conjunction with Wind River's Simics [64]. We simulate a 16-core CMP, as presented in Table 4.1. Each core in the system is an in-order-execution UltraSPARC III+ core.

The basic test routines (Test Daemons) for this approach are the same with Selective SBST that are presented in Section 4.5.1. In terms of the memory footprint, Table 4.2 shows the memory capacity consumed by each test program, which includes both the source code of the test program (i.e., instructions in text segment) and the size of the test patterns loaded from memory (i.e., the data segment). Obviously, the test programs targeting the integer units (the

Table 5.1: The execution times and memory footprints of the, *memory-intensive* test programs used to assess the enhanced, *cache-aware* DaemonGuard framework.

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

first four functional units in Table 4.2) are extremely small and can easily fit within the L1 cache of a core. Thus, these test programs are not memory-intensive.

As previously mentioned in Section 5.2, there are larger test programs already employed in multi-core SBST schemes [42, 45]. These programs are considered larger, in the sense that they rely on large numbers of test patterns loaded from off-chip memory. Furthermore, given the continuous increase in both the size and complexity of multi-/many-core microprocessors, it is reasonable to anticipate an equivalent increase in the amount of required test patterns. To account for these trends, we develop a second set of test programs, as shown in Table 5.1. These test programs have the same structure as those in Table 4.2, with the only difference being in the increased amount of loaded test patterns. We primarily change the size of the data segment of the first four test routines of Table 4.2, as these are the ones invoked more frequently under the selective testing scenario proposed in Chapter 4. The last three test routines (the ones for the floating-point units) already have a large data segment (see Column 6 of Table 4.2), so we do not modify these. The second set of these test programs, shown in Table 5.1, will be used to assess the efficacy and efficiency of the cache-aware DaemonGuard framework.

Full-system, execution-driven simulations are performed using the PARSEC benchmark suite. Details about the input sizes, number of executed instructions, and execution times are given in Table 4.3.

Table 5.2: Testing overhead results. The testing overhead is defined as the ratio of the test execution time (i.e., the time expended on testing sessions) over the total execution time of the system (i.e., until the benchmark application completes its execution). The results are averaged over the 16 cores of the system.

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

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

For the evaluation of the cache-aware DaemonGuard framework, we employ the memory-intensive set of test programs, as shown in Table 5.1. The main goal of cache-aware selective testing is to further decrease the testing time by exploiting the cache hierarchy to minimize expensive off-chip memory accesses. In order to assess the efficacy of the proposed cache-aware DaemonGuard mechanism described in Section 5.2, we investigate three different testing mechanisms: (1) the *best-case* scenario depicted in Figure 5.1 (ideal execution), (2) the non-cache-aware DaemonGuard framework of Section 4.5, and (3) the proposed enhanced cache-aware DaemonGuard. The first evaluated mechanism (best-case test scenario) corresponds to the situation where most test instructions and data are cache-resident, i.e., the previous testing session has just finished and its working set is still in the cache. This scenario is the optimal in terms of test execution time – since it minimizes the number of off-chip memory accesses – and it can be viewed as the ideal reference point for the evaluation of cache-aware selective testing. The second evaluated test mechanism (non-cache-aware DaemonGuard) initiates testing based solely on the thresholds of the various functional units, i.e., it is oblivious to the recent testing history of other similar functional units in other cores. Finally, the third mechanism

Table 5.3: The number of test-program-related LLC misses incurred by the three evaluated testing mechanisms. Note that a reduction in LLC misses results in lower test times and lower off-chip memory traffic.

Benchmark	Number of LLC Misses (Testing)			Incr. in LLC Misses		Impr. (%)
	Best-Case (Ideal)	Non-Cache Aware	Cache-Aware	Non-Cache Aware	Cache-Aware	
Blackscholes	135 K	779 K	634 K	5.8×	4.7×	17%
Bodytrack	220 K	1178 K	985 K	5.33×	4.4×	16%
Canneal	478 K	1069 K	821 K	2.23×	1.7×	23%
Dedup	245 K	542 K	428 K	2.21×	1.7×	21%
Ferret	230 K	1271 K	925 K	5.53×	4.02×	27%
Fluidanimate	457 K	1742 K	1114 K	3.8×	2.4×	36%
Freqmine	280 K	1038 K	833 K	3.71×	2.98×	20%
Raytrace	185 K	613 K	510 K	3.31×	2.8×	16%
Swaptions	171 K	710 K	633 K	4.16×	3.7×	11%
Vips	917 K	3146 K	2002 K	3.43×	2.18×	36%
x264	72 K	168 K	159 K	2.33×	2.21×	5%
Average	309 K	1115 K	823 K	3.80×	2.99×	21%

(the proposed cache-aware technique) corresponds to the methodology described in Section 5.2, whereby test sessions can be initiated earlier than usual to ensure that they are executed within the window of opportunity provided by the cache hierarchy.

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

Table 5.2 presents the testing overhead results. The testing overhead is defined as the ratio of the test execution time (i.e., the time expended on testing sessions) over the total execution time of the system (i.e., until the benchmark application completes its execution). The results are averaged over the 16 cores of the system. As can be seen from the results, the proposed cache-aware selective testing methodology reduces the testing overhead incurred by memory-intensive test programs, as compared to the non-cache-aware DaemonGuard of Section 4.5. On average, the cache-aware DaemonGuard reduces the testing overhead from 24% to 12%, as compared to the non-cache-aware technique. This decrease corresponds to a quite significant reduction of 50% in testing overhead. The ideal (best-case) results are also shown as an indication of the theoretical maximum achievable improvement.

Table 5.3 provides further details about the number of test-program-related LLC misses

incurred by the three evaluated testing mechanisms of the cache-aware DaemonGuard. The non-cache-aware DaemonGuard mechanism suffers from a 4× higher, on average, number of LLC misses, as compared to the ideal (best-case) scenario. When employing the cache-aware DaemonGuard, the number of incurred LLC misses related to the test programs is reduced by 21%, on average, as compared to the non-cache-aware mechanism. Note that beyond the substantial savings in testing time, the significant decrease in the number of LLC misses also implies a corresponding decrease in the off-chip network traffic, which further improves the overall system performance.

5.4 Concluding Remarks

This chapter demonstrates the Cache-Aware Selective SBST that significantly reduces the execution of memory-intensive test programs. In Chapter 4 we proposed the Selective SBST where test programs are initiated based on unit utilization. In this chapter, we enhance the Testing Manager process in order to invoke Test Daemons not only based on the utilization, but also on the recent history of test sessions. This will result in the reduction of the execution time of test programs by minimizing the number of off-chip accesses by exploiting cache-resident test related data. Experimental results indicate an approximate 50% reduction in testing overheads compared to non-cache-aware selective testing So far, we have proposed on-line fault detection techniques that are based on the monitoring of the system's activity and perform selective testing. In order to have a complete fault detection solution, we need to include the ability to test the entire system within the same testing session. This approach will set the system off-line as all the processing elements should be tested concurrently. The next chapter investigates different test scheduling policies to maximize the system availability during test given a test latency constrain.

Chapter 6

Optimizing System Availability during SBST

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 chapter, 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.

6.1 Introduction

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 [1, 2], the test scheduling issue becomes even more pertinent. One approach is to periodically initiate testing on *all* system cores simultaneously [12, 14, 69]. 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 [42, 48, 59]. 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). Selective is part of this work and presented in Chapter 4.

In this chapter, 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 chapter 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 experimrental 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.

This work was published and presented in a peer-reviewed conference [46]. A journal version has been submitted and is under review in the Journal of Electronic Testing, Theory and Applications.

6.2 Related Work

Recently, several techniques have investigated the scheduling of test routines in multi-/many-core systems under SBST. Apostolakis et al. [12] 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 [12] is to reduce the total test application time, assuming that all cores are tested simultaneously (i.e., full-system parallel testing).

Haghbayan et al. [48] 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 [14], 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. [59] 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 [44], 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 [44] only examines fully parallel system testing (i.e., testing all cores simultaneously), leading to zero availability during the SBST session.

In [54, 70], 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.

6.3 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

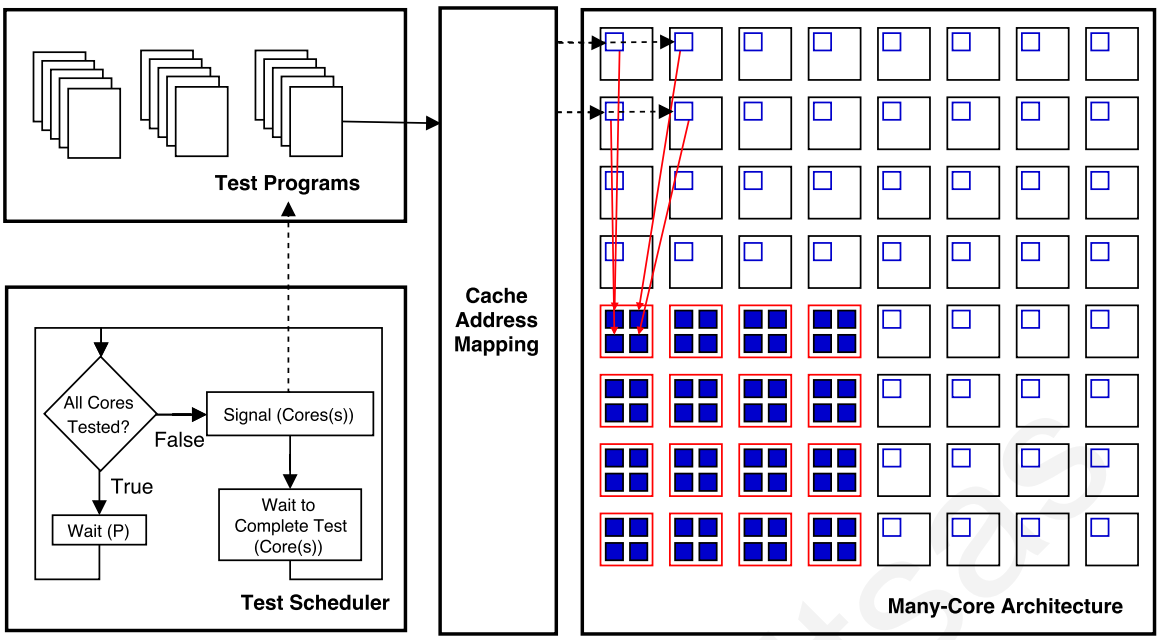


Figure 6.1: Architectural overview of the employed framework. The two main components are (1) the Test Scheduler (an OS process), and (2) the actual Test Programs, which all operate at the OS level. The *Cache Address Mapping* component is responsible for the mapping of physical addresses to the cores 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.

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 investigate 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 chapter, a *cluster* is defined as a group of cores within the many-core system where the 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 adapt the DaemonGuard Framework (Chapter **??**) as abstractly depicted in Fig. 6.1. 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. 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 Figure 6.1) are facilitated by the DaemonGuard Framework.

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 [71]. In [72], 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 work. 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.

6.4 Test-Scheduling Exploration

6.4.1 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 6.4.4.

In this chapter, 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.

6.4.2 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 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.

Algorithm 3 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
```

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).

As mentioned in Section 6.3, the scheduling process is the responsibility of the test scheduler OS process. Algorithm 3 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. 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.

6.4.3 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 6.3), 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.

$$SAT_k = \frac{TL_k \times N - TO_k}{TL_k \times N} \tag{6.1}$$

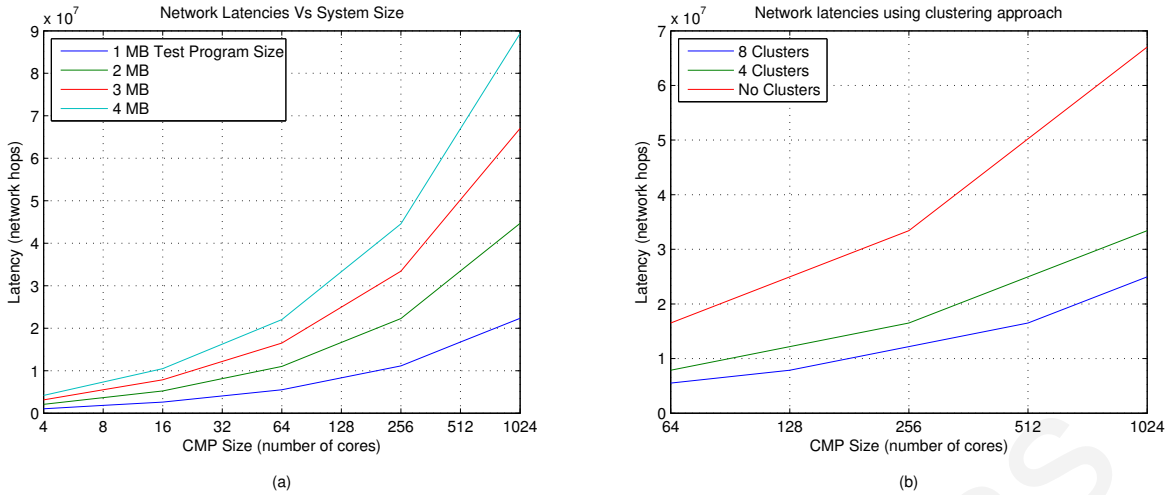


Figure 6.2: A high-level statistical analysis investigating the on-chip network latency (in terms of network hops) as the number of on-chip cores in the CMP increases. Results in (a) the absence of clustering, and (b) in the presence of clustering are depicted. In the latter case, the size of the employed test program is set to 3 MB.

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.

$$SAT_{max} = \max_{k}\{SAT_k\}, k = 1..N \tag{6.2}$$

$$subject\ to\ TL_k < L \tag{6.3}$$

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.

6.4.4 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 6.3. 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 6.4.1, 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 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 6.2(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 6.2(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 chapter. The results in Figure 6.2(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 6.4.2) 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 6.4.2 can be applied to either of the aforementioned two approaches (for the testing of the cores within each cluster).

6.5 Experimental Framework and Results

6.5.1 Evaluation Framework

For the evaluation of the proposed test scheduling techniques, we perform full-system, execution-driven simulations using the Wind River's Simics [64] simulator extended with the Wisconsin GEMS toolset [65] and the GARNET network model [73]. We simulate two multi-core sys-

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 6.1: Simulated system parameters.

tems, as presented in Table 6.1, 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 Test Daemons from DaemonGuard Framework that perform testing at the full-core level, to test each individual core according to the proposed test-scheduling techniques.

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 [66] 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.

6.5.2 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.

Performance of a 16-core CMP

Figure 6.3 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 6.3) 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 6.3. 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 6.3, 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 6.3. 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

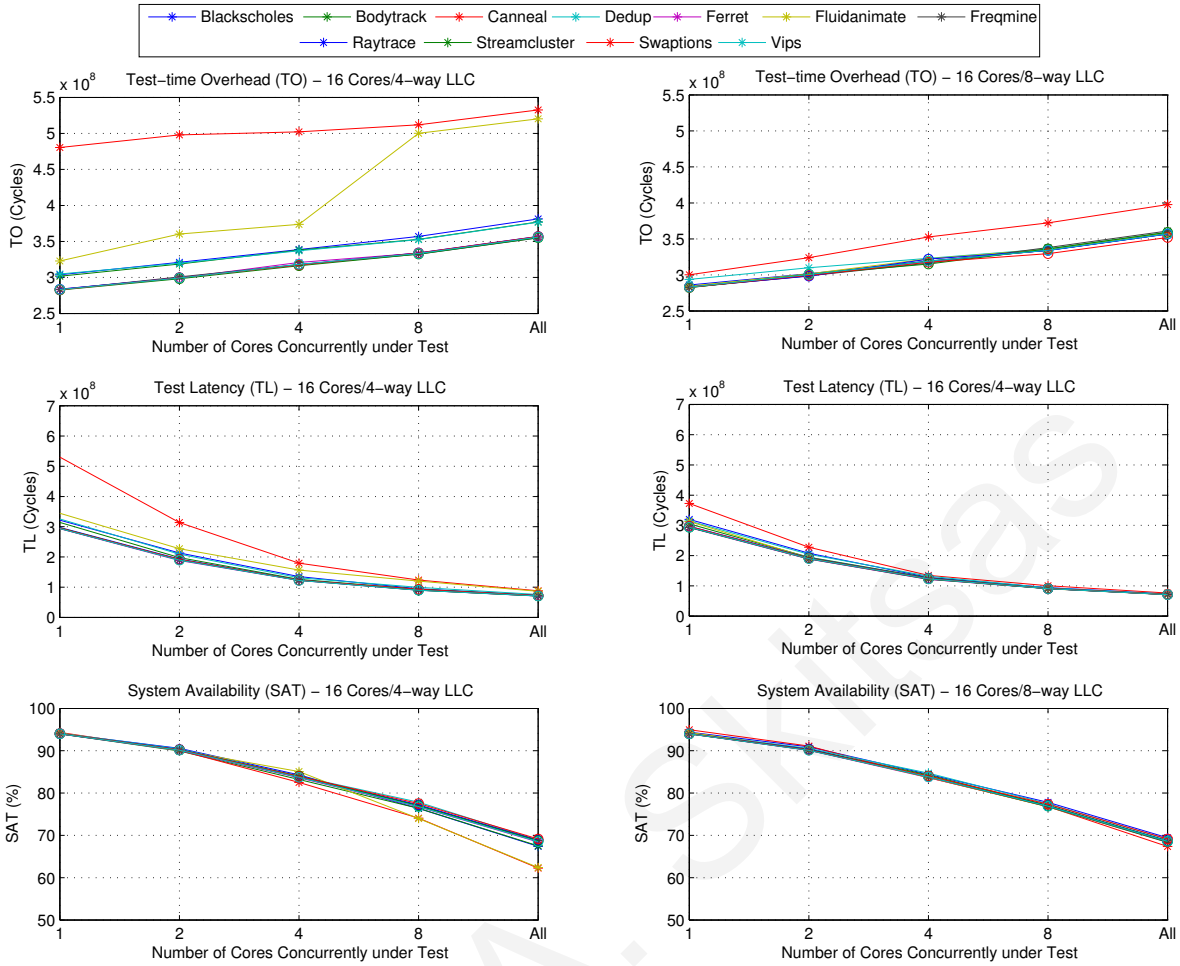


Figure 6.3: The results of the *16-core CMP* system for all examined PARSEC benchmarks. Each row of plots corresponds to one of the three evaluated metrics. The left-column plots corresponds to a 4-way LLC, while the right-column plots correspond to an 8-way LLC. The total LLC size for both setups is the same.

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 6.3) depicts the system availability under test (SAT) for each of the scheduling scenarios under exploration, calculated using Equation (1) (see Section 6.4.3) for both cache setups (4-way and 8-way). The SAT metric combines the two test metrics under consideration and reveals the best 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.

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.

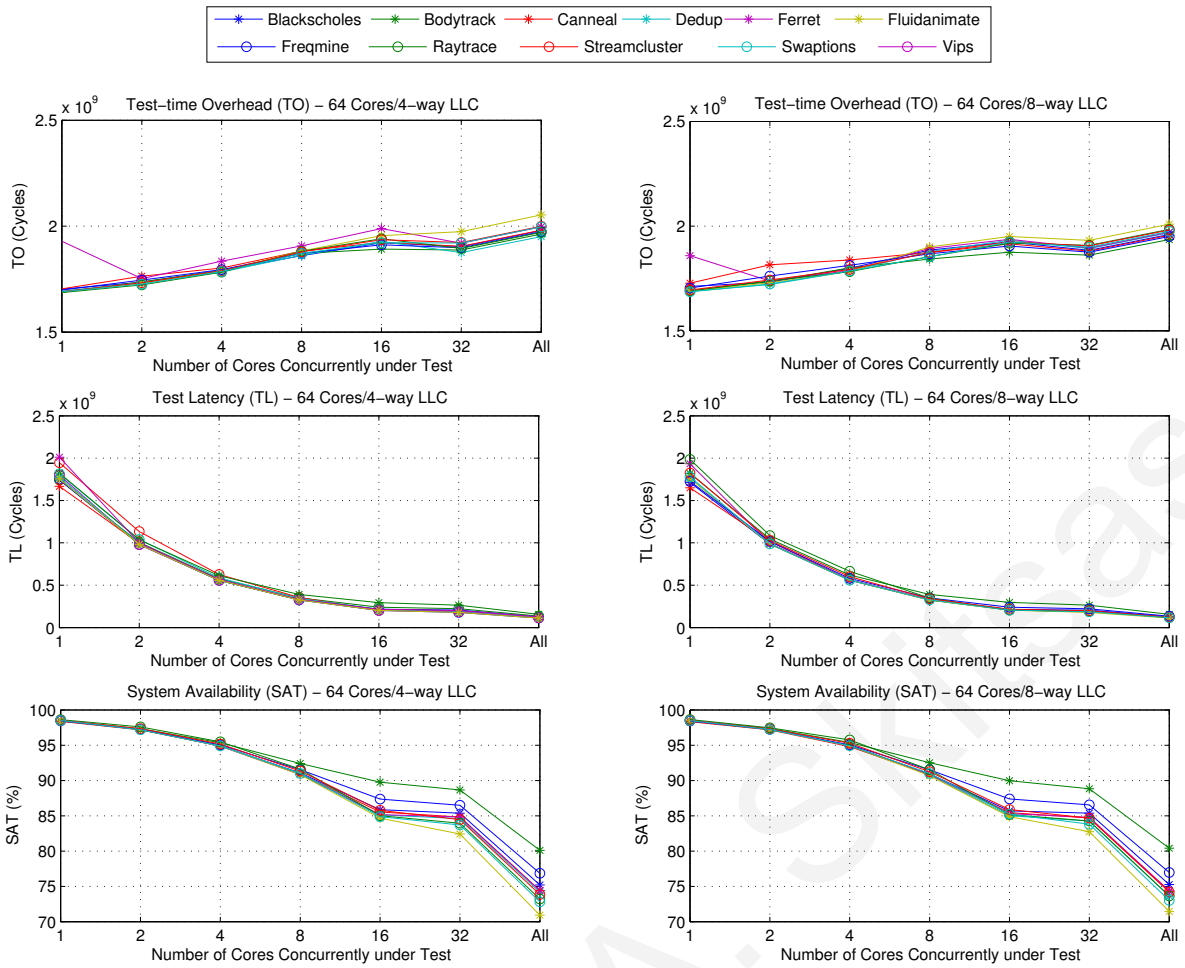


Figure 6.4: The results of the *64-core CMP* system for all examined PARSEC benchmarks. Each row of plots corresponds to one of the three evaluated metrics. The left-column plots corresponds to a 4-way LLC, while the right-column plots correspond to an 8-way LLC. The total LLC size for both setups is the same.

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 6.4 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. The impact of the cache is also demonstrated by doubling

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

Table 6.2: 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.

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 6.2 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 6.2, 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 in Section 6.4.4. The main premise of the clustering approach is to maintain all the test-related data within the vicinity of the cores to be tested.

6.5.3 Evaluating the Clustering Approach

In this sub-section, we evaluate the effectiveness of the clustering approach of Section 6.4.4. 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

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

the L2 banks of the cores comprising the cluster (using a dynamic data mapping mechanism, as mentioned at the end of Section 6.3). 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 6.3 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 6.3 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 6.1). Hence, the clustering approach allows us to contain the TO and scale the investigated test-scheduling policies to arbitrarily large CMP systems.

Figure 6.5 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

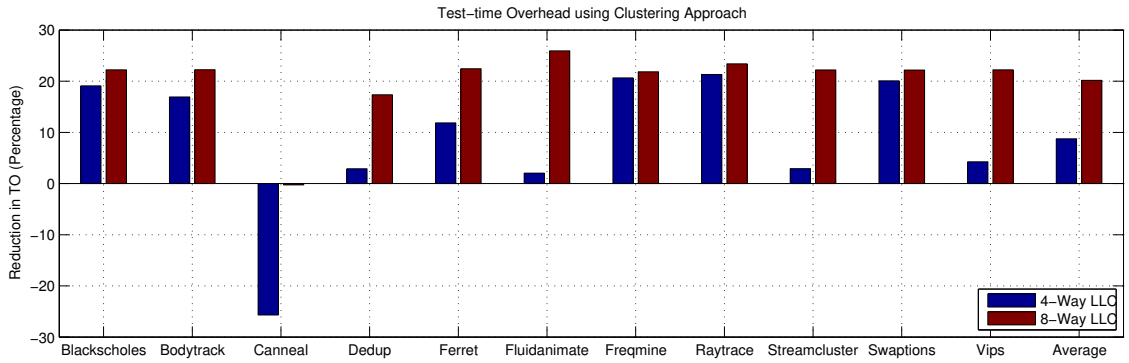


Figure 6.5: An overview of the savings obtained when using the clustering approach. The graph shows the percentage reduction in the test-time overhead when using clustering, as compared to the case *without* clustering. Two different LLC associativity setups are evaluated: 4-way and 8-way.

to our previous experiments, we evaluate two different LLC associativity setups: 4-way and 8-way. As demonstrated by the results in Figure 6.5, 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 6.3), 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.

6.6 Concluding Remarks

This chapter presents an exploration of periodic, on-line SBST scheduling policies based on the number of cores concurrently under test during test sessions. The scope of this exercise is to propose the scheduling methodology that maximizes system availability for running a normal workload under a test latency constraint. As the test target of this approach is the entire system, ensuring the system's scalability is an important requirement. Thus, this chapter proposes a clustering approach in order to maintain the performance of our methodologies while the system scales up in terms of number of cores. 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. The overall goal of this thesis is to develop a dependable system able to tolerate permanent hardware failures encountered during the normal operation of many-core architectures. So far,

we proposed techniques for on-line fault detection. The next chapter deals with mechanisms able to assist with the recovery of the system in the presence of permanent faults.

Chapter 7

System Recovery in the Presence of Faults

In-field on-line testing techniques have recently been proposed for permanent fault detection caused by wear-out/aging-related defects manifesting during the lifetime of a system. Selective Software-Based Self-Testing (SBST) is one such paradigm focusing primarily on the recently stressed functional units of a multicore system at a sub-core granularity, in an attempt to reduce the application performance penalty caused by periodically testing the entire system. In this chapter, we enhance our O/S-enabled framework DeamonGuard for on-demand (selective) SBST to support fault recovery capabilities. Towards this goal, we propose an efficient check pointing and rollback recovery mechanism which, upon fault detection, can restore the system to the most recently valid correct state and resume the normal operation assuming disabling of the faulty core, thereby leading to a healthy (but degraded) system. The work in this chapter concentrates on reducing the number of stored checkpoints required when testing at a sub-core granularity, and minimizing the recovery penalty of such framework. We evaluate and present the overhead of the proposed recovery mechanism. Our results indicate a practical reduction in the number of stored checkpoints as well as a significant improvement in recovery latency for the cases where the faults are correlated with the stressed units.

7.1 Introduction

In previous chapters, we proposed several SBST methodologies in order to efficiently detect permanent failures in modern many-/multi-core systems during the lifetime of the chip. Beyond the detection of faults, modern systems must be enhanced with mechanisms able to self-repair and recover the system to a fault-free state, in order to remain functional despite the presence of permanent faults.

To maintain proper operation of the system, several error recovery techniques have been proposed. These techniques are classified mainly in two categories: (i) Forward Error Recovery (FER), and (ii) Backward Error Recovery (BER). In the first (FER), the usage of redundant hardware is necessary for error detection and recovery. On the other hand, BER requires to store a fault-free state of the system using checkpoints for error recovering. Once an error is detected, the system is able to rollback to the fault-free state and re-execute the affected workload, assuming it supports reconfiguration/fault-containment capabilities to rule out the malfunctioning component. In most cases, BER does not require extra hardware to support the recovery procedure, and the imposed overhead is in execution time, since a subset of the already executed workload needs to be re-executed. There is also some storage overhead for saving the checkpoints.

Several approaches for checkpointing and rollback (during the recovery procedure) have been proposed in the literature, targeting multicore architectures. SafetyNet [74] combines local check-pointing and incremental loggings. In SafetyNet, components coordinate their local checkpoints, in order to represent a consistent global recovery point. Revive [75] uses log-based rollback mechanisms, based on global checkpoints, and enables recovery from permanent faults, while the faulty core is disabled. Revive I/O [76] is an extension of Revive that deals with the output-commit problem. The issue of scalability in checkpointing solutions in large scale parallel computing systems is further addressed in the more recent works of [77] and [78], where hardware-based coordination for local checkpointing and multi-level checkpoint are respectively proposed.

Recovery mechanisms in multicore systems must be incorporated with on-line fault detection schemes. Concurrent methods relying on fault-tolerant mechanisms (i.e., redundancy techniques) [42, 52] and non-concurrent on-line testing, such as Software-Based Self-Testing (SBST) techniques [3,18] could both support recovery mechanisms. In this chapter, we propose a recovery mechanism for a selective SBST technique of a shared-memory multicore system. During selective SBST, the execution of test programs is performed on demand at sub-core granularity. The system monitors the utilization of the functional units of each core and invokes the corresponding test program when a predetermined threshold of executed instructions is reached. Utilization is used as a surrogate measure as it is directly related to the switching frequency and activity factor of the components [61]. Real-time system monitoring and test initiation and execution is overseen by DeamonGuard Framework (Chapter 3). In this chapter, we expand DeamonGuard Framework to support recovery capabilities in the presence of permanent faults, based on checkpointing and rollback.

Once an error is detected by on-demand SBST, the system must be able to recover via roll-back to a fault-free state. This is achieved by the recovery mechanism that is responsible to roll back the system to a valid checkpoint that ensures the correct execution of the workload until the time the checkpoint was captured. Since the considered detection mechanism (selective SBST) performs on-demand testing at sub-core granularity, checkpoints are captured and stored in the system at irregular time intervals (not periodically), which can cause problems with checkpoint consistency. This problem can be trivially resolved with *global* checkpointing (which, in turn, can be enhanced with scalable checkpointing solutions, such as those in [74], [75], [77], [78]). Under global checkpointing, the architectural state of the *entire* system (i.e., all cores) is saved. Note that this solution tends to incur additional overhead, since the number of stored checkpoints grows with both the number of sub-core units and the number of cores in the system. This is an inherent characteristic of on-demand SBST for detection, performed either at the sub-core or at the core level. The first contribution of this chapter is the reduction of the number of total checkpoints in the system at any time, using a dynamic policy which can associate a checkpoint with more than one units/cores according to when their SBST was last executed. The second contribution of this chapter is the optimization of the recovery latency required for rollback to a fault-free checkpoint using a newly proposed algorithm able to find an appropriate checkpoint – called the Most Recently Valid (MRV) checkpoint – aiming to reduce the recovery time overhead. Hence, both types of recovery overheads (checkpoint storage and recovery latency) are examined. It is assumed that once the recovery process is finished, a reconfiguration mechanism can isolate the faulty core so that the system can continue the fault-free operation with a performance degradation, due to the reduced number of functional cores. Reconfiguration is further discussed in the next Section, however, the details of this mechanism are beyond the scope of this thesis.

This work was published and presented in a peer-reviewed conference [**?**].

7.2 General Framework for Fault Detection & Recovery

To tolerate permanent hardware failures encountered during the normal operation of a many-core architecture, the system must be augmented with three salient capabilities: (i) a fault-detection mechanism (without loss of generality, we employ an on-line software-based self-testing mechanism for fault detection); (ii) a recovery technique to restore the system to a correct state after a fault is detected (this constitutes the main focus of this chapter); (iii) a hardware reconfiguration mechanism to keep the system operational by disabling the faulty module(s).

A high-level overview of the general framework used to facilitate fault detection and recovery in a multi-/many-core setup is presented in Chapter 3 and Figure 3.1 . The rest of this section describes in brief the key elements of this framework with focus on the recovery which is the main focus of this chapter.

7.2.1 Fault Detection

The considered shared-memory multi-core system incorporates on-line testing, using the SBST mechanism for permanent fault detection. The assumption is that the system is homogeneous, i.e., all processing cores are identical. Test programs are executed during the lifetime of the system, and they aim to detect faults caused by aging and wear-out artifacts. In particular, we use the recently proposed on-demand selective testing methodology of the DaemonGuard framework (Chapter 4. DaemonGuard Framework employs a new form of SBST, namely selective SBST, which tests individual functional units in each processor core on-demand. Specifically, DaemonGuard performs testing of functional units (at the sub-core granularity), based on each unit's utilization, i.e., how often the unit is used during the execution of various benchmark applications.

The main component of the DaemonGuard framework is the Testing Manager process, which runs at the OS-level and is responsible for the invocation of the various Test Daemons (test programs targeting individual functional units of each core of the system), based on the utilization information provided by hardware instruction counters residing alongside each functional unit within the CPU cores. The main function of the Testing Manager is the checking for pending test requests by any functional unit of any core within the system. When a testing session is completed, the Testing Manager calls the Checkpoint Manager (if the test has passed successfully) to create a system (global) checkpoint and store the current fault-free state. Otherwise, the Recovery Manager is invoked, in order to roll back the system to a previous fault-free state.

While selective SBST has been shown to be extremely efficient in terms of testing-time overhead and avoiding unnecessary over-testing, it suffers from one drawback: it generates significantly more checkpoints. In fact, the checkpoints generated are proportional to the number of individual functional units present in each core (every test session for each functional unit generates a new global checkpoint). Given that each checkpoint incurs both a timing overhead (the time needed to generate the checkpoint) and a storage overhead, it is imperative to minimize the number of generated (and stored) system checkpoints. This is precisely one of the

fundamental objectives of this thesis.

7.2.2 Fault Recovery

The main focus of this chapter is the system recovery process, once a permanent fault has been detected. Hence, a recovery mechanism is introduced, which is able to keep the system operational despite the occurrence of permanent faults. The main components of the proposed recovery mechanism are (a) the Checkpoint Manager, and (b) the Recovery Manager, which are responsible, respectively, for the creation of system checkpoints and determining a valid checkpoint (among the multiple stored ones) to roll back.

The Checkpoint Manager can potentially create a checkpoint after each testing session. If the test passes successfully, then a new checkpoint is created, in order to capture the fault-free state up until the testing time of the specific unit. In this thesis, we use a global checkpoint for the entire system, in order to ensure checkpoint consistency despite the presence of a shared-memory environment. Furthermore, we propose a methodology to reduce the number of checkpoints caused by the multiple testing sessions resulting from the multi-core architecture and the sub-core test granularity considered.

The Recovery Manager is called upon the detection of a permanent fault, and it is responsible to find an appropriate checkpoint, which ensures a consistent, fault-free state to roll back and recover the system. An algorithm that is able to find the Most Recently Valid (MRV) checkpoint is proposed, which aims to reduce the recovery penalty by avoiding roll-backs to the oldest checkpoint.

7.2.3 System Reconfiguration

After the detection of a permanent fault and the subsequent recovery to a fault-free state, the system should be able to isolate the faulty module and reconfigure itself to a fault-free (albeit degraded) operational mode. The granularity of system reconfiguration could vary. For example, reconfiguration could be performed at the core granularity (by disabling the entire faulty core), or at a sub-core granularity (by disabling individual functional units within a core). The latter approach requires diagnosis and localization of the permanent fault(s) [79], and the core must be designed with multiple/spare units, or cross-core redundancy capabilities, in order to disable/bypass faulty modules [80, 81]. The core micro-architecture considered in this chapter does not include any spare functional units, so we assume – without loss of generality – that the reconfiguration mechanism works at the *core granularity* (i.e., entire cores are disabled upon

fault detection), which does not require any diagnosis mechanisms. We assume that the operating system is aware of the reconfiguration policy and, upon rollback, it is able to re-distribute the workload to the fault-free (and still active) cores, and resume execution from the selected checkpoint. Note that the reconfiguration mechanism is beyond the scope of this thesis; the proposed recovery mechanism is orthogonal to the reconfiguration technique employed by the system.

7.3 The Proposed Recovery Mechanism

During SBST, upon completion of each testing session, the system stores its fault-free (assuming the test completed successfully) state information in a checkpoint. This information allows the system to be restored to a fault-free state after the detection of a fault. In multi-core systems, where the testing procedure could be performed at a fine granularity (e.g., core-level or sub-core level), the check-pointing mechanism is more complicated. Three main problems arise, which must be taken into account: (i) the checkpoint consistency, (ii) the number of generated and stored checkpoints, and (iii) ensuring an appropriate fault-free state (i.e., choosing an appropriate checkpoint) for rollback. To resolve the first problem, we consider global checkpoints covering all the cores of the system. Techniques that improve the checkpoint consistency and the scalability of global checkpointing – such as [75, 77, 78] – could be combined with the proposed recovery mechanism. For the second problem, we propose a methodology that reduces the number of checkpoints stored at any given time by the system. For the last problem, we propose a methodology to find the most recent checkpoint that ensures a fault-free state for the system.

7.3.1 Reducing the Number of Checkpoints

At a core-level test granularity, where each core is tested independently, we create a global checkpoint to store the fault-free state. As a result, only the tested core could be characterized as fault-free in the checkpoint, since the remaining cores of the system may not have been tested yet. In order to have a sufficiently robust recovery mechanism, we need one checkpoint for each core of the system. Thus, the number of checkpoints required for full-core-based testing and recovery is equal to the number of cores in the system. In this case, the valid checkpoint to recover after a fault detection is the last captured checkpoint corresponding to the faulty core. In our work, since we consider selective testing at a sub-core granularity, checkpoints are created

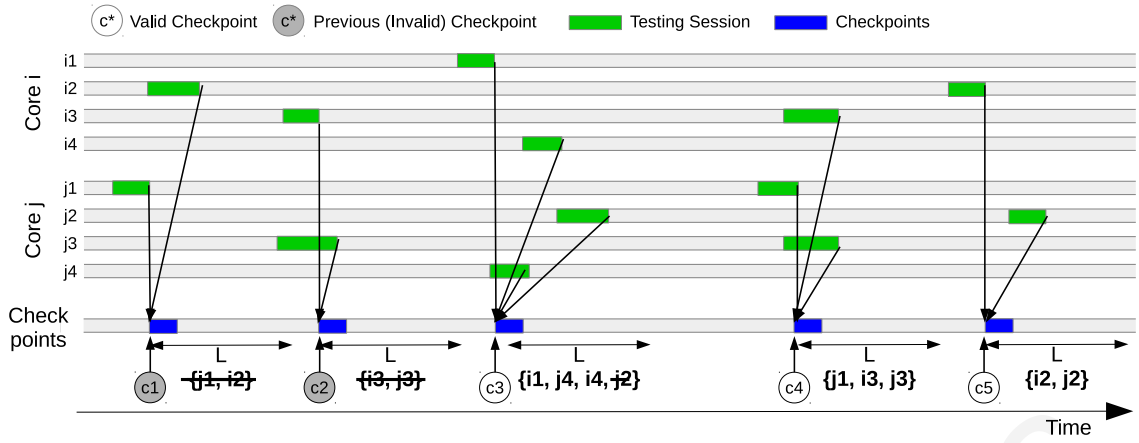


Figure 7.1: An example scenario of the proposed check-pointing system, assuming the use of selective SBST to detect the presence of permanent faults.

at the end of each testing session, targeting a specific unit of a core within the system. As a result of this, the number of stored checkpoints in the system increases, and it is equal to the number of cores in the system multiplied by the number of units considered for testing. To tackle this large number of generated checkpoints, we propose a methodology aiming to reduce the total number of checkpoints.

The main idea of this methodology is to use an existing checkpoint for more than one unit. In particular, units that are tested within a specific number of cycles, L, after the creation of a checkpoint may use the last created checkpoint for roll back (instead of creating a new one). This period can be viewed as a *window of opportunity* to reduce the number of checkpoints. Checkpoints are enhanced with a list of units that are assigned to them, and if any of those units fails a future test, the state will roll back to that particular checkpoint. The Checkpoint Manager is responsible to decide either to create a new checkpoint, or to use an existing one and append the tested unit to the list. In the latter case, the considered unit must be removed from the list of the old checkpoint that is currently assigned to said unit. Checkpoints that do not have any units assigned to them are considered invalid and they discarded from the system. Algorithm 4 describes the proposed methodology.

Figure 7.1 illustrates an example of the proposed check-pointing mechanism. A global checkpoint is captured after the completion of the testing session of a specific functional unit. The example illustrates two cores of the system, i and j, with four units each. Once a unit is tested, the Checkpoint Manager is invoked to decide whether to create a new checkpoint, or to use an existing one (based on the distance, in cycles, from the last checkpoint). The C_1 checkpoint is created after the testing of unit j_1, and, subsequently, unit i_2 is appended to the same checkpoint. The same procedure is applied to the following testing sessions as well. After

the creation of the C_4 and C_5 checkpoints, the units assigned to C_1 have been tested again, so they are appended to the new checkpoint lists. At this point, those units are removed from C_1 and the state of said checkpoint becomes invalid (it is removed from the system).

Algorithm 4 Checkpoint Manager

Input: Tested Unit U

Input: Last Checkpoint C

1: **if** $U.TimeStamp - C.TimeStamp < L$ **then**
2: $\quad C.appendUnit(U)$
3: **else**
4: $\quad NewC = CreateNewCheckpoint()$
5: $\quad NewC.appendUnit(U)$
6: **end if**
7: $LastC = U.LastAssignCheckpoint()$
8: $LastC.removeUnit(U)$
9: $U.updateCheckpoint(NewC)$

7.3.2 Identifying the Most-Recently Valid (MRV) Checkpoint

When a fault is detected and the recovery mechanism is invoked, the system rolls back to a checkpoint that ensures fault-free re-execution of the workload. As mentioned earlier, during selective testing, we hold all the relevant checkpoints of all sub-core units for each core of the system. The example in Figure 7.1 shows that, at any given time, we have 4 (number of units per core) checkpoints for each core of the system. The checkpoint that ensures fault-free state recovery is the oldest checkpoint among all the units of the cores of the system.

It should be noted that despite the unit-based testing at a sub-core granularity assumed in this chapter, the rollback process is performed at the system level, since we use global checkpoints. In order to improve the recovery penalty – by avoiding rollback to the oldest checkpoint – we propose an algorithm that is able to select the Most Recently Valid (MRV) checkpoint for system recovery. The proposed algorithm is described in Algorithm 5.

Algorithm 5 gets as input the faulty unit and the list with the global checkpoints. The first step of the algorithm is to sort the checkpoints according to the capture time (cycle), with the oldest checkpoint being first, and then set the *Cu* checkpoint as the latest valid checkpoint for the unit that has sustained a fault. The second step of the algorithm is to find the MRV

Algorithm 5 Find the Most-Recently Valid (MRV) Checkpoint

Input: Faulty Unit U

Input: List of Checkpoints C

Sort C based on time (Oldest First)

$Cu \leftarrow C.getCheckpoint(U)$

1: **repeat**
2: $Cd_1 \leftarrow C$.remove()
3: $Cd_2 \leftarrow C$.pick()
4: **if** $Cd_1 = Cu$ **then**
5: return Cu
6: **end if**
7: $l \leftarrow Cd_2$.cycle - Cd_1.cycle
8: $t \leftarrow$ AccumulateTestingTime(Cd_1.ListofUnits)
9: **if** $t < l$ **then**
10: $response \leftarrow$ testUnit(Cd_1.ListofUnits)
11: **if** $response = fail$ **then**
12: return Cd_1
13: **end if**
14: **else**
15: return Cd_1
16: **end if**
17: **until** ($True$)

checkpoint to recover the system. The main idea of this procedure is to avoid rollback to the oldest checkpoint, in the case where we can find a more recent checkpoint that still ensures fault-free execution of the affected workload. During the first iteration of the algorithm, we get the oldest checkpoint, Cd_1, and the next one, Cd_2. From the oldest checkpoint, we identify the type of the units assigned to it, and we estimate the required time to test all of them. The estimation is based on a priori profiling of each test program. If the estimated testing overhead is smaller than the distance in time (cycles) of the two consecutive checkpoints, we execute the test programs corresponding to the units assigned to the oldest checkpoint (Cd_1). If the test is successful, we remove Cd_1 from the list, and we repeat this process for the next checkpoints. This procedure is repeated until: (i) the testing overhead is higher than the recovery from an older checkpoint,

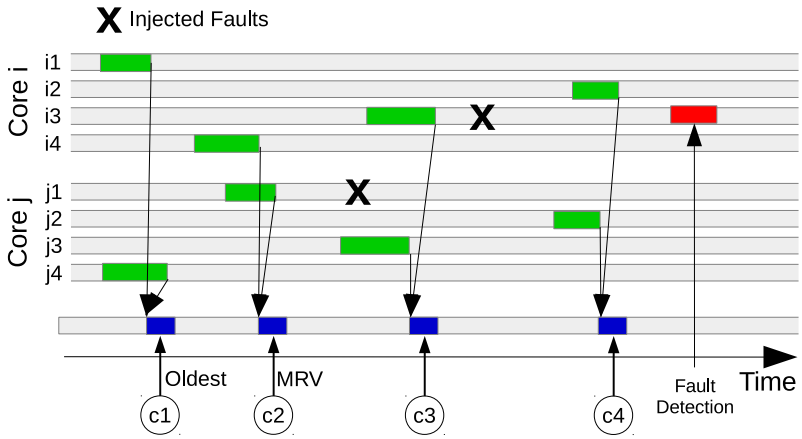


Figure 7.2: An example of the use of the MRV algorithm, which reduces the roll-back recovery penalty.

whereby we return Cd_1 (line 15), or (ii) one of the executed tests fails, whereby the algorithm returns Cd_1 (line 12), or (iii) all the functional units pass their tests, whereby Cu is considered to be the MRV checkpoint for recovery.

Figure 7.2 illustrates an example where the MRV algorithm is applied. In this example, two faults occur, one in each of the two considered cores, i and j. Note that the occurrence of multiple permanent faults (at different locations on the chip) becomes more relevant as the system ages; wear-out and aging effects may cause precipitation of multiple faults in the late stages of the system's lifetime. Thus, it is imperative for any mechanism to cover those situations as well. During the testing session of unit i_3 in the example of Figure 7.2 (red color), a fault is detected and the MRV algorithm is used. The oldest checkpoint in this case is C_1, and the next step of the proposed algorithm is to find a more recent valid checkpoint (if it exists). To do this, we move to the next (chronologically) checkpoint, i.e., C_2, and we initiate testing of the functional units assigned to the C_2 checkpoint. In this particular example, during the testing procedure of the units assigned to C_2 (i.e., units i_4 and j_1), a second fault is detected in unit j_4. As a result of this, C_2 is now considered as the MRV checkpoint for roll-back. Had the second fault not occurred, the MRV checkpoint would be C_3. In both cases, the roll-back to the oldest checkpoint is avoided, thereby reducing the recovery penalty.

Once the system is recovered to a fault-free state, a system reconfiguration must be performed, in order to isolate the faulty core. This process leads to a degraded (but operational) system. As previously mentioned, we assume that the OS would redistribute the system workload (as restored from the checkpoint) to the fault-free cores of the degraded system and resume execution.

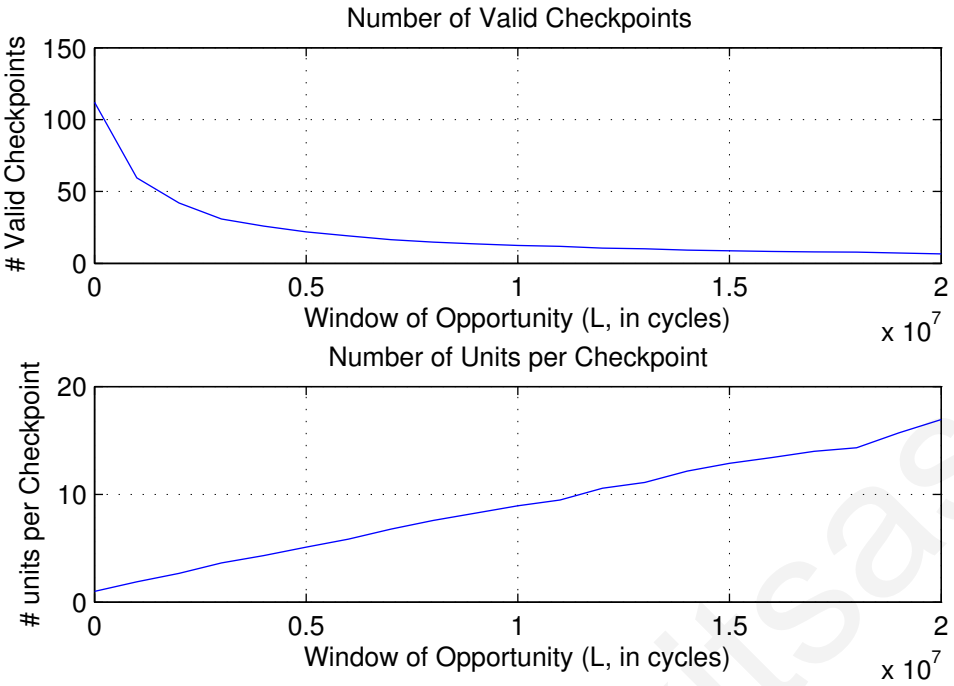


Figure 7.3: Reduction of the number of stored checkpoints by applying the proposed mechanism varying the Window of Opportunity.

7.4 Experimental Results

For the evaluation of the proposed recovery mechanism, we have developed a simulation framework that incorporates unit-based fault injection, a detection mechanism based on selective SBST, and the proposed recovery mechanisms using rollback and check-pointing. We consider a 16-core system with 7 functional units in each core. Each unit is tested by an appropriate software test program with the same characteristics as those described in DaemonGuard Framework and used for Selective SBST. Faults escaping the SBST detection mechanism are not considered in this chapter. The considered workload is synthesized using detailed application profiling of the PARSEC benchmark suite [66].

The total duration of simulations is 20 billion cycles. The testing threshold (utilization metric which triggers selective SBST) is set to 10 million instructions. Under-utilized units are tested periodically, if the executed instructions do not reach the threshold within a specific time period, to ensure that all units will be tested at least once during a simulation process. Multiple fault injections and simulations are performed, in order to investigate the detection and recovery overhead in the presence of faults. Faults are injected in both utilized and under-utilized units. The reported results for each unit are averaged over 10 fault injection campaigns per unit, whereby a fault is injected into the considered unit at a time.

We first study the effectiveness of the proposed policy in reducing the total number of

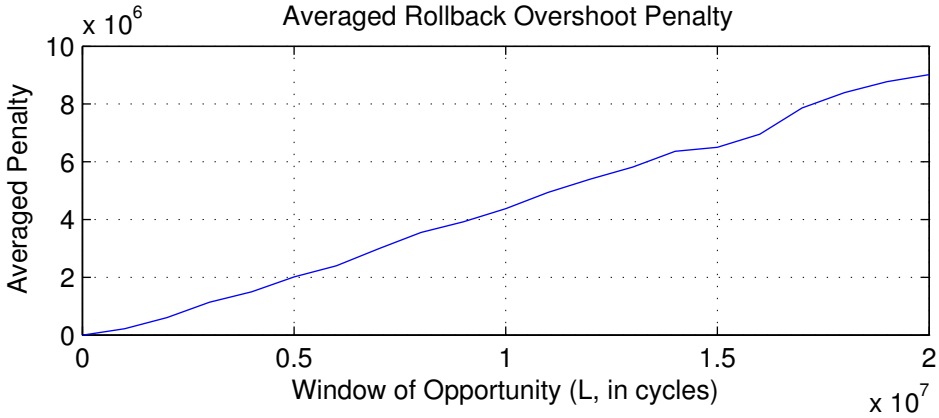


Figure 7.4: The average distance between the checkpoint capture time and the time where a testing session is completed for the units that are appended to an existing checkpoint.

checkpoints kept at any time. Figure 7.3 presents the number of stored valid checkpoints and the number of functional units assigned to each checkpoint at any given time, by varying the *Window of Opportunity* L. As expected, the number of checkpoints is reduced as L increases. Actually, the number of checkpoints is reduced quickly with a small increase in the value of L. Since we do not create checkpoints for each tested unit in the system, we have an extra overhead in terms of recovery latency which is the distance between the checkpoint timestamp and the time where the testing session is completed. However, this extra penalty is limited by the value of period L which is the maximum distance that can appear. Figure 7.4 presents the averaged distance of each functional unit to the checkpoint. For the rest of our results we set the window of opportunity $L = 4M$ cycles where the number of stored checkpoints is about 25% of the initial.

Figure 7.5 presents results for the detection latency and recovery penalty of the two recovery policies, i.e., (i) oldest checkpoint, and (ii) the proposed MRV checkpoint approach. The x-axis shows the 7 considered functional units of the system and the y-axis refers to the latency for three metrics (detection and the two above-mentioned recovery policies), in terms of cycles. The functional units are sorted based on their utilization. The ALU is the highest utilized unit, while MULT (multiplier) is the least utilized unit.

As we can see in Figure 7.5, the latency of the fault detection is increased as the utilization decreases. This is a result of selective testing, as units are tested based on their utilization. Comparing the ALU and MULT units, detection is more than 3× faster in ALU. The recovery penalty incurred when using the oldest check-pointing method is similar for all the units of the system. The reason for this is that irrespective of the faulty unit, the system recovers to the oldest checkpoint, and in most cases the oldest checkpoint is determined by the under-utilized units.

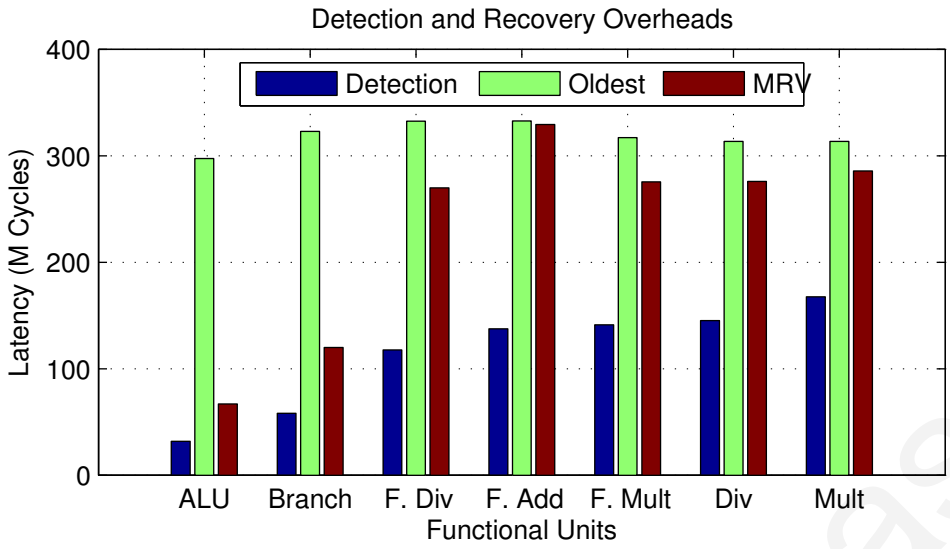


Figure 7.5: Detection and recovery overhead. The results are averaged over several fault injection experiments.

The small variations between the units are related to the detection latency. In the case of highly utilized units, the recovery is slightly better, due to the fast detection of faults. Considering the proposed MRV checkpoint mechanism, we can significantly reduce (up to 4×) the recovery penalty, when the fault occurs in a stressed unit. The improvement of the MRV mechanism for the under-utilized units is smaller, compared to that achieved for the stressed units, since in most cases the most recently valid checkpoint is near the oldest one.

As we consider faults caused by aging and wear-out artifacts, faults are more likely to occur in units with higher stress during the lifetime of the system. Based on this attribute, the MRV check-pointing mechanism can significantly improve the recovery latency.

7.5 Concluding Remarks

This chapter introduced an efficient checkpointing and roll-back recovery mechanism for selective SBST techniques. Selective testing performs on-demand testing at the granularity of functional units based on utilization. Thus, the time interval between two consecutive testing sessions is not constant. This creates challenges related to the number of checkpoints that are required to hold the fault-free state of the system and the roll-back penalty. In this chapter, two mechanisms are proposed in order to meet the challenges. For the evaluation of the proposed techniques, we developed a simulation framework with fault-injection capabilities. Results validate the significant reduction in both the number of checkpoints and the roll-back recovery penalty.

Chapter 8

Conclusions

The main goal of this thesis is to develop a framework able to provide protection against undesired system behavior caused by aging and wear-our effects in shared-memory multi-core systems. On-line testing techniques are employed in modern systems in order to perform dynamic detection of permanent faults during the lifetime of the system. Beyond the fault detection, systems must be enhanced with recovery capabilities in order to keep the system functional in the presence of permanent fault.

The work in Chapters 4 and 5 introduces the notion of Selective SBST as a means to drastically reduce the testing time overhead in multi-/many-core microprocessors. The proposed testing methodology considers system activity at the sub-core granularity and initiates targeted testing of only the over-utilized (and, thus, strained) functional units. Under-utilized units are only sporadically tested. To facilitate this testing regime, we introduce the DaemonGuard Framework for Selective SBST, which enables real-time observation of individual sub-core modules and performs on-demand selective testing of individual functional units. This discriminatory testing approach offers drastic savings in testing time, since the testing phase only executes test routines relevant to the specific functional unit under test. DaemonGuard employs a transparent, minimally-intrusive, and lightweight operating system process that monitors the utilization of individual components at run-time. Whenever a unit requires testing, DaemonGuard invokes OS-residing unit-specific test daemons to execute appropriate test routines.

In Chapter 4, selective testing is compared against two full-core SBST approaches to evaluate the testing time overhead incurred on the system. Our results indicate substantial reductions in testing overhead of up to 30×. Most importantly, the impact of the DaemonGuard framework on system performance is shown to be negligible, thus corroborating our claim that OS-assisted selective SBST is a feasible option for modern microprocessors. Further-

more, the DaemonGuard Framework for Selective SBST is augmented with the capability to exploit the memory hierarchy of the system in order to expedite the testing process. Large, memory-intensive test programs tend to stress the memory sub-system of the CMP. In Chapter 5 we demonstrate that cache-aware selective testing can significantly reduce the execution of memory-intensive test programs, by exploiting cache-resident blocks and minimizing the number of expensive off-chip memory accesses. Thus, the cache-aware DaemonGuard scheme initiates test sessions based not only on unit utilization, but also on the recent history of test sessions by other similar units in other cores. The cache-aware testing scheme is shown to be very effective in exploiting the memory hierarchy to minimize the testing time of memory-intensive test programs.

Unlike selective testing, a different testing policy where all the cores of a multi-core system are required to be tested is evaluated through the implementation of different scheduling policies. The work performed in Chapter 6 is an exploration of periodic, on-line 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.

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, the work in chapter 6 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 enables efficient scalability of the testing process to arbitrarily large systems.

Finally, the work in Chapter 7 introduces an efficient checkpointing and roll-back recovery mechanism for selective SBST techniques. In the presence of permanent faults, the proposed technique is able to effectively recover a shared-memory multi-core system to a fault-free state and keep the entire system operational, with some performance degradation. In terms of checkpointing, a methodology is proposed to substantially reduce the total number of generated and stored checkpoints, which improves the viability of selective SBST. Complementing the checkpointing process, a recovery mechanism identifies the most appropriate checkpoint

for roll-back, in order to minimize the recovery penalty. The proposed methodology is evaluated under extensive fault-injection scenarios targeting both highly-utilized and under-utilized units. Results validate the significant reduction in both the number of checkpoints and the roll-back recovery penalty. Especially in the cases where the faults are more likely to occur in highly-stressed units with higher utilization, the recovery penalty can be dramatically reduced.

Overall, the research within this thesis focuses on the development of mechanisms towards the protection of multi-/many-core systems against undesired behavior. In the area of SBST, this thesis contributes to a significant reduction of testing time overhead taking into advantage the monitoring of system's activity and initiating on demand testing targeting the high-utilized elements. Despite the potential of on-demand selective testing during lifetime, the system must be able to perform self-test targeting its' entirety. Regarding this requirement, our work contributes to the optimization of system availability under a given test latency constraint through test scheduling approaches. Beyond self-testing, this thesis contributes in system recovery after the detection of permanent faults by developing mechanisms for efficient checkpointing and roll-back recovery in systems that perform on-demand testing.

8.1 Future Work

The open research challenges in the area of on-line fault detection towards reliable systems in combination with the outcomes of the proposed methodologies set the basis for further research on top of this thesis. There are several directions that future research can take in order to improve the findings of this thesis. Selective SBST, a hybrid approach that combines testing at sub-core granularity with full-core testing can result in further reductions in test-time overheads and detection latencies. Using larger programs (targeting more than one functional units or the entire core) provides the potential to reduce the overheads imposed by the OS while test daemons are invoked and run (scheduling). Merging the test programs of different functional units is a challenging approach as several parameters such as the utilization of the considered units should be considered. Additionally, test programs that target un-core components can be part of selective SBST. This, necessitates the need to include triggering mechanisms of these programs as are not part of the monitoring activity (functional units).

Cache-aware selective SBST and clustering approach proved able to manage large test programs as well as large systems. In this context, additional techniques can be applied in large test programs in order to further reduce the test-time overheads. A potential approach in this direction is the pipelined execution of test programs. The execution of large programs in seg-

ments (one after other) can reduce the misses of LLC resulting in savings in testing time. A very simple approach towards this extension is to split the test routine in two parts. The scheduler will initiate the execution of the first part for all the cores of the system and then for the second part. All the proposed test scheduling approaches (varying the number of concurrent cores under test) can be applied.

For the last part of this thesis, several directions of future work can be considered. Recovery mechanisms with capabilities to isolate the faulty component and reconfigure the system in order to remain operational can be implemented. Additionally to this, techniques that differentiate transient and permanent errors should be included in the proposed framework. An example is that upon the detection of a fault, the test scheduler can re-execute the same test program. If the fault is also detected, we have a permanent fault and further actions for reconfiguration should be taken. Otherwise, we can assume the detection of a transient fault and we can proceed with the recovery only. Another potential direction of future work and more relevant with the first part of this thesis is that the testing manager of the selective testing can receive feedback from the checkpointing mechanism towards the further reduction of the number of stored checkpoints as well as the reduction of the recovery penalty in the presence of faults.

In this thesis and specifically in the fault detection techniques our goal is to reduce the imposed testing time overheads either by perform on-demand testing or using different scheduling approaches. A potential direction that this work can be further continued is to identify and use different metrics for the assessment and evaluation of testing exercise. A reliability model that includes the test quality (i.e. test coverages), system characteristics and requirements (i.e. critical or non-critical systems), detection latencies can be integrated to the DaemonGuard Framework in order to trigger and assess the testing procedure.

An alternative approach to addressing the issue of aging and wear-out can be implemented by exploiting the already proposed DaemonGuard Framework. So far, we deal with aging by providing mechanisms able to perform on-line detection of permanent faults during the lifetime of the system. Tackling the issue from a different perspective is to provide solutions able to predict or slow down the aging phenomenon. Such techniques can be implemented and orchestrated by the DaemonGuard Framework.

To sum up, further development of the DaemonGuard Framework can lead to a complete simulation framework that can support detection, prevention and recovery of permanent faults due to aging and wear-out phenomenon on top of the modern architectures. The modular and scalable design of DaemonGuard allows the integration of different self-testing approaches targeting multi-core systems.

Bibliography

[1] S. Borkar, "Thousand core chips: A technology perspective," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 746–749. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278667

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

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

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

[5] M. Nicolaidis and Y. Zorian, "On-line testing for vlsi - a compendium of approaches," *Journal of Electronic Testing*, vol. 12, no. 1-2, pp. 7–20, February 1998.

[6] M. Agarwal, B. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *25th IEEE VLSI Test Symposium, 2007.*, May 2007, pp. 277–286.

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

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

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

[10] D. Sylvester, D. Blaauw, and E. Karl, "Elastic: An adaptive self-healing architecture for unpredictable silicon," *Design Test of Computers, IEEE*, vol. 23, no. 6, pp. 484–490, June 2006.

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

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

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

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

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

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

[17] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on VLSI Systems*, vol. 16, no. 11, pp. 1441 –1453, nov. 2008.

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

[19] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 88–99, Jan 2005.

[20] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *IEEE International Test Conference (ITC), 1998,*, October 1998, pp. 990–999.

[21] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: Building high availability systems with commodity multi-core processors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 470–481. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250720

[22] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, June 2007, pp. 317–326.

[23] H. T. Li, C. Y. Chou, Y. T. Hsieh, W. C. Chu, and A. Y. Wu, "Variation-aware reliable many-core system design by exploiting inherent core redundancy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2803–2816, Oct 2017.

[24] E. Rotenberg, "Ar-smt: a microarchitectural approach to fault tolerance in microprocessors," in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, June 1999, pp. 84–91.

[25] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 25–36.

[26] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99–110.

[27] C. Wang, H. s. Kim, Y. Wu, and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection," in *International Symposium on Code Generation and Optimization (CGO'07)*, March 2007, pp. 244–258.

[28] K. H. Chen, J. J. Chen, F. Kriebel, S. Rehman, M. Shafique, and J. Henkel, "Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3441–3455, November 2016.

[29] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: Communication-optimised multi-threaded error-detection technique," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: ACM, 2016, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/2968455.2968508

[30] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[31] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 243–254.

[32] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Efficient software-based fault tolerance approach on multicore platforms," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 921–926.

[33] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 25:1–25:17. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901339

[34] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 196–207.

[35] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, December 2007, pp. 210–222.

[36] A. Meixner and D. J. Sorin, "Error detection using dynamic dataflow verification," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, Sept 2007, pp. 104–118.

[37] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, February 2007, pp. 169–180.

[38] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, July 2006.

[39] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser.

ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736063

[40] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346315

[41] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168868

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

[43] K. J. Lee, P. H. Tang, and M. A. Kochte, "An on-chip self-test architecture with test patterns recorded in scan chains," in *2016 IEEE International Test Conference (ITC)*, November 2016, pp. 1–10.

[44] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulos, "Accelerated online error detection in many-core microprocessor architectures," in *VLSI Test Symposium (VTS), 2014 IEEE 32nd*, April 2014, pp. 1–6.

[45] M. Kaliorakis, N. Foutris, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Online error detection in multiprocessor chips: A test scheduling study," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, July 2013, pp. 169–172.

[46] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "Exploration of system availability during software-based self-testing in many-core systems under test latency constraints," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, October 2014, pp. 33–39.

[47] S. Gupta, A. Ansari, S. Feng, and S. Mahlke, "Adaptive online testing for efficient hard fault detection," in *Proceedings of the 2009 IEEE International Conference on Computer Design*, ser. ICCD'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 343–349. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792354.1792420

[48] M.-H. Haghbayan, A.-M. Rahmani, A. Miele, M. Fattah, J. Plosila, P. Liljeberg, and H. Tenhunen, "A power-aware approach for online test scheduling in many-core architectures," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 730–743, 2016.

[49] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "Daemonguard: Enabling o/s-orchestrated fine-grained software-based selective-testing in multi-/many-core microprocessors," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1453–1466, May 2016.

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

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

[52] O. Khan and S. Kundu, "Hardware/software codesign architecture for online testing in chip multiprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 714 –727, Sept - Oct 2011.

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

[54] A. Kamran and Z. Navabi, "Hardware acceleration of online error detection in many-core processors," *Canadian Journal of Electrical and Computer Engineering*, vol. 38, no. 2, pp. 143–153, 2015.

[55] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-sbst methodology for efficient testing of processor cores," *IEEE Design Test of Computers*, vol. 25, no. 1, pp. 64–75, January 2008.

[56] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, October 2016.

[57] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, A. Sansonetti, and G. Squillero, "Software-based self-test techniques for dual-issue embedded processors," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[58] D. Sabena, M. S. Reorda, and L. Sterpone, "On the automatic generation of optimized software-based self-test programs for vliw processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 813–823, April 2014.

[59] Y. Li, O. Mutlu, and S. Mitra, "Operating system scheduling for efficient online self-test in robust systems," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 201–208. [Online]. Available: http://doi.acm.org/10.1145/1687399.1687436

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

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

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

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

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

[65] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's

general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1105734.1105747

[66] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[67] "Opensparc t1 microarchitecture specification," August 2006.

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

[69] A. Kamran and Z. Navabi, "Stochastic testing of processing cores in a many-core architecture," *Integration, the VLSI Journal*, vol. 55, pp. 183–193, 2016.

[70] ——, "Self-healing many-core architecture: Analysis and evaluation," *Hindawi Publishing Corporation, VLSI Design*, vol. 2016, 2016.

[71] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, vol. 00, pp. 250–261, 2009.

[72] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736045

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

[74] D. Sorin, M. Martin, M. Hill, and D. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 123–134.

[75] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02, 2002, pp. 111–122.

[76] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "Revivei/o: Efficient handling of i/o in highlyavailable rollback-recovery servers," in *In HPCA*, 2006, pp. 203–214.

[77] R. Agarwal, P. Garg, and J. Torrellas, "Rebound: Scalable checkpointing for coherent shared memory," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011, pp. 153–164.

[78] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, November 2010, pp. 1–11.

[79] M. Schölzel, T. Koal, and H. T. Vierhaus, "An adaptive self-test routine for in-field diagnosis of permanent faults in simple risc cores," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2012, pp. 312–317.

[80] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 43–51.

[81] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, USA: ACM, 2009, pp. 93–104.