University of Cyprus

Department of Electrical and Computer Engineering

Parallel Fault Simulation and Test Generation Automation Processes for Chip Multiprocessors

Stavros Hadjitheophanous

A Dissertation Submitted to the University of Cyprus in Partial Fulfillment

of the Requirements for the Degree of Doctor of Philosophy

July, 2018

VALIDATION PAGE

Stavros Hadjitheophanous

Parallel Fault Simulation and Test Generation Automation Processes for Chip Multiprocessors

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 July 20, 2018 by the members of the Examination Committee.

Committee Chair ______________________________
Dr. Chrysostomos Nicopoulos

Research Supervisor ______________________________
Dr. Maria K. Michael

Committee Member ______________________________
Dr. Theocharis Theocharides

Committee Member ______________________________
Dr. Xrysovalantis Kavousianos

Committee Member ______________________________
Dr. Stelios N. Neophytou

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

Ζούμε στη εποχή των αρχιτεκτονικών πολλαπλών πυρήνων όπου προϊόντα καθημερινής χρήσης όπως τα έξυπνα τηλέφωνα, wearable's, τάμπλετς ακόμα και αυτοκίνητα εμπεριέχουν ολοκληρωμένα κυκλώματα πολλαπλών πυρήνων. Ο αριθμός των ενσωματωμένων πυρήνων ποικίλει από μερικές δεκάδες σε συσκευές χαμηλών προδιαγραφών σε μερικές εκατοντάδες στους υπερυπολογιστές. Η διαχρονική σμίκρυνση της τεχνολογίας και οι προβλέψεις για ακόμα περεταίρω αύξηση στον αριθμό των πυρήνων ανα τσιπ έχουν θέσει του πολυπύρηνους επεξεργαστές στο επικέντρο του ενδιαφέροντος από διάφορα ερευνητικά προγράμματα.

Οι πολύ-επεξεργαστές έχουν την δυνατότητα να εκτελούν πολλαπλές εντολές παράλληλα σε διαφορετικούς πυρήνες μειώνοντας σημαντικά τον συνολικό χρόνο εκτέλεσης του προγράμματος. Επιπρόσθετα, προσφέρουν μεγάλη επεξεργαστική δύναμη, γρήγορη και ομοιόμορφη πρόσβαση ενσωματωμένης μνήμης και έξυπνους μηχανισμούς ενδο-επικοινωνίας μεταξύ των πυρήνων και αποφυγής συγκρούσεων στην κοινή μνήμη (shared memory coherency) καθιστώντας τους ιδανικούς για πολλές εφαρμογές όπως εφαρμογές επεξεργασίας ψηφιακού σήματος, ενσωματωμένων συστημάτων, δικτύων, μονάδων επεξεργασίας γραφικών (GPUs) και πολλά αλλά.

Θεμελιώδης και επεξεργαστικά δύσκολα προβλήματα αυτόματου έλεγχου όπως το πρόβλημα της προσομοίωσης σφαλμάτων (fault simulation) και παραγωγής διανυσμάτων δοκιμής (test generation) μπορούν να εκμεταλλευτούν τις τελευταίες εξελίξεις στο τομέα των πολυπύρηνων επεξεργαστών για να προσφέρουν πιο αποτελεσματικές λύσεις. Η δομή των αλγορίθμων αλλά και η στοχευμένη εκτέλεση των προγραμμάτων μπορούν να επηρεάσουν σημαντικά τη απόδοση των πολυεπεξεργαστών. Σύμφωνα με τον νόμο του Amdahl η επιτάχυνση που επιτυγχάνετε μέσω του παραλληλισμού είναι άμεσα συνδεδεμένη με το ποσοστό του κώδικα που μπορεί να παραλληλοποιηθεί.

Για την παραλληλοποιήση βασικών προβλήματων αυτόματου ελέγχου οι μηχανικοί έχουν την τάση να βασίζονται σε εκλεπτυσμένους μεταγλωττιστές και αυτόματα εργαλεία παραλληλοποίησης κώδικα τα οποία δεν εκμεταλλεύονται πλήρως τους επεξεργαστικούς πόρους

της εκάστοτε αρχιτεκτονικής. Τα συγκεκριμένα προβλήματα λόγω την αυξημένης τους πολυπλοκότητας και της δυναμικής τους φύσης είναι πολύ δύσκολο να προβλεφθεί η ροή εκτέλεσης τους και ως αποτέλεσμα οι μεταγλωττιστές και τα αυτόματα εργαλεία παραλληλοποίησης του κώδικα δεν μπορούν να δουλέψουν αποδοτικά και συχνά καταλήγουν σε μηβέλτιστες λύσεις.

Οι ευκαιρίες που προσφέρονται από την εξέλιξη των πολυπύρηνων επεξεργαστών δημιουργούν μεγαλύτερες προκλήσεις για την παραλληλοποίηση δύσκολων προβλημάτων αυτόματου έλεγχου. Διάφοροι παραδοσιακοί αλγόριθμοι θα πρέπει να ξανασχεδιαστούν λαμβάνοντας υπόψη όλα τα πλεονεκτήματα και μειονεκτήματα που προσφέρονται από τη ανάπτυξη των πολυπύρηνων επεξεργαστών. Αυτή η διατριβή μελετά την επίδραση που έχει ο διαχωρισμός του φόρτου εργασίας κατά την παραλληλοποίηση δύσκολων προβλημάτων αυτόματου ελέγχου ο οποίος μπορεί να έχει σημαντική επιρροή στην αποδοτικότητα αλλά και στην ποιότοιτα των αποτελεσμάτων τους. Επιπρόσθετα αναλύονται διάφορες μεθόδοι παραλληλοποίησης οι οποίες στοχεύουν να βελτιώσουν σημαντικά την απόδοση των προβλημάτων προσομοίωσης σφαλμάτων (fault simulation) και παραγωγής διανυσμάτων δοκιμής (test generation), βασισμένοι στις πολυπύρηνες ομογενείς αρχιτεκτονικές κοινής μνήμης. Οι προτεινόμενες μέθοδοι είναι ικανές να διατηρούν την επεκτασιμότητα τους καθώς αυξάνετε ο αριθμός των επεξεργαστών που χρησιμοποιούνται καθώς επίσης να διατηρούν και την καλή ποιότητα των αποτελεσμάτων. Επίσης, η διατριβή προτείνει μια τροποποιημένη παράλληλη μέθοδο την παραγωγή διανυσμάτων δοκιμής για το πρόβλημα των πολλαπλών ανιχνεύσεων (n-detect) test set. Τα αποτελέσματα των πειραμάτων επιβεβαιώνουν τις μεγάλες δυνατότητες των προτεινόμενων λύσεων.

Επιπρόσθετα, η παρούσα διατριβή μελετά αλγόριθμους για την βελτίωση της αξιοπιστίας σε μικρό-επεξεργαστές πολλαπλών πυρήνων (CMP). Η παρατεταμένη καταπόνηση ενός συγκεκριμένου μέρους των επεξεργαστών πολλαπλών πυρήνων συνδέεται με τη αυξανόμενη ευαισθησία στην φθορά. Ενώ μια αποτυχία σε ένα τμήμα του τσιπ μπορεί να μην είναι απαραιτήτως καταστροφική για ολόκληρο το σύστημα, εντούτοις, ακόμη και μια απλή φθορά σε ένα κρίσιμο μονοπάτι για παράδειγμα σε συστήματα δίκτυων σε τσιπ (NoC) ή σε ένα από τους επεξεργαστές πυρήνα μπορεί να θέσει εκτός λειτουργιάς ολόκληρο το CMP. Διάφορα διανύσματα δοκιμής μπορούν να δημιουργηθούν κατά την διάρκεια του σχεδιασμού του CMP και μπορούν να εφαρμοστούν κατά τη διάρκεια αδράνειας της λειτουργιάς του για να βοηθήσουν στην παράταση της διάρκειας ζωής του. Στην παρούσα διατριβή παρουσιάζεται μια νέα τεχνική παραγωγής διανυσμάτων δοκιμής (exercise vectors) βασισμένη κυρίως σε τεχνικές αυτόματης παραγωγής διανυσμάτων δοκιμής (ATPG) η οποία μπορεί να παράξει ένα μικρό αριθμό διανυσμάτων που βοηθούν στην βελτίωση της αξιοπιστίας των CMP.

Abstract

Electronic devices are experiencing the era of multi-core architectures where even everyday life products like smartphones, wearables, tablets or even cars, houses many processors in a single chip. The number of processing units called cores are increasing from tens for low-end devices to hundreds in supercomputers. Technology shrinking, as well as the industry trends and the market size forecasts, suggest that those numbers will continue to grow. Multi-cores processors have the ability to run multiple instructions on different processing units at the same time and as a result to speed-up the overall execution time. Also, they offer a huge amount of processing power, fast and uniform on-chip memory, advance inter-core communication methods and shared memory coherency that can be utilized by a variety of application domains such as general-purpose computers, digital signal processing, embedded systems, networks and GPUs.

Computationally intensive fundamental test automation problems such as fault simulation and test generation can benefit from those developments. However, the performance gained by the use of multi-cores is wildly depended on the software algorithms used and the corresponding implementation. Based on Amdahl's law the limitation on the speed-up gain is strongly related to the percentage of the software that can concurrently run on multiple processing units. Engineers opt to rely on automatic parallelization tools consisting of sophisticated schedulers and compilers for the parallelization of the test automation problems which can limit the efficiency of the parallelized processes. Due to the complexity and the dynamic nature of those processes general purpose automatic parallelization tools cannot predict a priori the execution flow of the test automation algorithms, thus they can lead to local optimal solutions.

All those new potentials open up a significant research topic focusing around parallelization of fundamental test automation processes, where traditional algorithms are re-visited taking into account all the new challenges. The present thesis investigates the impact of partitioning in parallel test automation processes which can significantly enhance the performance of fault simulation and test generation processes by utilizing parallelization concepts for shared memory on-chip homogeneous architectures. The developed methods are able to maintain the scalability of the algorithms as the number of processing cores utilized is increased and at the same time avoid the test inflation problem. The parallel test pattern generation methodology is also extended to generate multiple-detect (n-detect) test sets. Experimental results validate the great potentials from the parallelization of the test automation processes.

Moreover, the thesis investigates algorithms for the improvement of the reliability in chip-multiprocessors (CMPs). Prolonged operational stress of a specific part of the logic is linked with increased susceptibility to wearout and failures. While a failure in a part of the chip might not be necessarily catastrophic for the whole system, even a single failure in a critical component like the inter-processor Network-on-Chip (NoC) fabric or core processor can cause a severe problem in CMP. Exercise vectors can be generated in the design phase, stored and utilized by CMP during idle times to prolong its lifetime. The thesis presents a novel vector generation technique based on ATPG techniques that generates a compact number of vectors that can improve the CMPs reliability.

Acknowledgments

Probably the hardest part of the thesis because it brings so many thoughts, feelings, memories about the whole journey. A journey that definitely taught me a lot and hopefully it will continue to do it in the upcoming journeys. Anything that I would write for my advisors Dr. Maria Michael and Dr. Stelios Neophytou will not be enough to express my gratitude, so I will only write a huge thank you!, for everything!.

Also, I would like to acknowledge University of Cyprus and 'KIOS Research and Innovation Center of Excellence' for providing an ideal environment for my studies. A special thanks to my colleagues for their inspiration and support all this time. I will not name each one of them since the contribution from everyone either small or big was important for me and deeply appreciated.

Last but not least, I would like to thank my friends and family, for believing in me and with their endless encouragement I was able to reach to the point of defending my Ph.D. Mary and Sofocle hope that I will be able to offer everything you did and doing for me and Nicolas, to my children someday. Zoe this is for you, for all the patience you had all this time and for making our life more beautiful, let's enjoy it. Thank you.

Publications

Thesis Journal Articles

1. **S. Hadjitheophanous**, S. N. Neophytou and M. K. Michael, "Exploiting Shared-Memory to Steer Scalability of Fault Simulation using Multicore Systems", accepted for publication, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), pp.1-14, July 2018, doi:10.1109/TCAD.2018.2855131.

2. H. Kim, S. B. Boga, A. Vitkovskiy, **S. Hadjitheophanous**, P. V. Gratz, V. Soteriou and M. K. Michael, "Use it or Lose it: Proactive, Deterministic Longevity in Future Chip Multiprocessors", in *ACM Transactions on Design Automation of Electronic Systems* (TODAES), vol.20, no.4, pp.1-26, Sept. 2015, doi:10.1145/2770873.

Thesis Conference Papers

1. P. M. Reddy, **S. Hadjitheophanous**, V.Soteriou, P. V. Gratz and M. K. Michael, "Minimal Exercise Vector Generation for Reliability Improvement", in *IEEE On-Line Testing and Robust System Design* (IOLTS), pp. 113-119, July 2017,
doi:10.1109/IOLTS.2017.8046205.

2. **S. Hadjitheophanous**, S. N. Neophytou and M. K. Michael, "Scalable Parallel Fault Simulation for Shared-Memory Multiprocessor Systems", in *IEEE VLSI Test Symposium* (VTS), pp.1-6, April 2016, doi:10.1109/VTS.2016.7477313.

3. **S. Hadjitheophanous**, S. N. Neophytou and M. K. Michael, "Utilizing Shared Memory Multi-cores to Speed-up the ATPG process", in *IEEE European Test Symposium* (ETS), pp.1-6, May 2016, doi:10.1109/ETS.2016.7519328.

4. S. N. Neophytou, **S. Hadjitheophanous** and M. K. Michael, "On the Impact of Fault List Partitioning in Parallel Implementations for Dynamic Test Compaction Considering Multicore Systems", in *IEEE International Design & Test Symposium* (IDT), pp.1-6, Dec. 2013, doi:10.1109/IDT.2013.6727082.

5. I. Voyiatzis, S. N. Neophytou, M. K. Michael, **S. Hadjitheophanous**, C. Sgouropoulou and C. Efstathiou, “Test set embedding into accumulator-generated sequences targeting hard-to-detect faults”, in *IEEE International Design & Test Symposium* (IDT), pp.1-2, Dec. 2013, doi:10.1109/IDT.2013.6727147.

Thesis Unpublished Journal Articles

1. **S. Hadjitheophanous**, S. N. Neophytou and M. K. Michael, “Maintaining Scalability of Test Generation in Multi-core Shared Memory Systems”, in *IEEE Transactions on Very Large Scale Integration Systems* (TVLSI), pp.1-12, submitted in July 2018.

2. P. M. Reddy, **S. Hadjitheophanous**, V.Soteriou, P. V. Gratz and M. K. Michael, “Exploiting Minimal Exercise Vector Generation Methods for Enhancing Reliability of Multiprocessors”, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), pp.1-10, to be submitted, 2018.

Other Related Publications

1. T. Charalambous, E. Klerides, W. Wiesemann, A. Vassiliou, **S. Hadjitheophanous** and K. M. Deliparaschos, “On the Minimum Latency Transmission Scheduling in Wireless Networks with Power Control under SINR Constraints”, in *IEEE Transactions on Emerging Telecommunications Technologies* (ETT), vol.26, no.3, pp.367-379, March 2015, doi:https://doi.org/10.1002/ett.2616.

2. C. Ttofis, **S. Hadjitheophanous**, A. S. Georghiades and T. Theocharides, “Edge-Directed Hardware Architecture for Real-Time Disparity Map Computation”, in *IEEE Transactions on Computers* (TC), vol.62, no.4, pp.690-704, Jan. 2012, doi:10.1109/TC.2012.32.

3. **S. Hadjitheophanous**, C. Ttofis, A.S. Georghiades and T. Theocharides, “Towards Hardware Stereoscopic 3D Reconstruction: A Real-Time FPGA Computation of the Disparity Map”, in *Design Automation and Test in Europe* (DATE), pp.1743-1748, March 2010, doi:978-3-9810801-6-2.

Table of Contents

Main Notations

ATPG:	Automatic Test Pattern Generation
s-a-0/*sa0*:	line stuck-at-0
s-a-1/*sa1*:	line stuck-at-1
FFR:	Fan-out-Free Regions
PPSFP:	Parallel Pattern Single Fault Propagation
SIMD:	Single Instruction Multiple Data
T:	Test set
F:	Fault List
m:	Number of available processing cores
w:	Processor word size
$\lvert T\rvert$:	Test set size
$\lvert F\rvert$:	Number of faults in fault list
HCI:	Hot-Carrier Injection
BTI:	Bias Temperature Instability
NBTI:	Negative Bias Temperature Instability
CUT:	Circuit-Under-Test
GPU:	Graphic Processing Units
CMPs:	Chip Multi-Processors
NoC:	Network-on-Chip
CMOS:	Complementary Metal-Oxide-Semiconductor
NP:	Non-deterministic Polynomial-time
VLSI:	Very Large Scale Integration
DFS:	Depth First Search
MUX:	Multiplexer
DAG:	Directed Acyclic Graph
TG:	Test Generation
I/O:	Input or Output
EDA:	Electronic Design Automation

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Technology shrinking in the integrated circuit manufacturing process allowed the implementation of multiple processing units (cores) on a single chip as well as large amounts of on-chip memory. These developments offer extensive processing power that can be used in various computationally intensive problems including popular electronic design automation processes. The distributed fashion of this processing power guides towards the development of parallel methodologies that scale well as the number of cores per chip are expected to increase beyond two dozens to hundreds. Multi-core architectures can significantly accelerate the performance of well-established design and test automation processes, yet parallelization is not straight forward and can certainly affect the quality of the obtained results. Decomposition, parallel execution and re-composition of the problem must be mindful in favor of parallelism without compromising the quality of the results with respect the existing non-parallel solutions.

Sophisticated compilers offers a lot of parallelization capabilities (like dynamic scheduling of tasks) and test automation engineers may opt to rely on automatic parallelization of fundamental test automation methodologies. However, due to the complexity and the nature of the problems, the efficiency of the obtained solutions cannot be guaranteed. Compilation and scheduling can have a great impact on how the problem is partitioned and processed by the individual cores which can affect the efficiency and the quality of the parallel solutions. Traditionally, parallel solutions involves three phases: (i) a decomposition step, (ii) parallel execution, and (iii) a re-composition step to construct the overall solution. Due to the dynamic nature of the problems each one of the steps has its own challenges. Except from the obvious additional work needed for issuing, partition and re-composition of the parallel threads there are many other challenges during the parallel execution phase such as shared memory access, communication, synchronization and race conditions among the processing units.

Before the multi-core era parallel test automation attempts mainly focused in one of the two different approaches: (i) bit level parallelism of different components and (ii) distribution of components among multiple processing units, not physically at the same chip (in most cases not even at the same machine). The latter approach suffered from prohibitive communication overhead and therefore the corresponding algorithms were design to avoid communication. On the other hand, bit-level parallelism was constrained by the machine's word size. All those factors must be taken into account for a scalable parallel solution and this thesis summarizes the key ideas proposed and ongoing research towards that direction.

The knowledge gained from the exploration of parallel test automation algorithms for CMPs is utilized in another research area, investigating CMPs reliability. Unfortunately, deep sub-micron CMOS process technology is marred by increasing susceptibility to wearout. Prolonged operational stress gives rise to accelerated wearout and failures, due to several physical failure mechanisms, including Hot Carrier Injection (HCI) and Negative Bias Temperature Instability (NBTI). Each failure mechanism correlates with different usage-based stresses, all of which can eventually generate permanent faults. While some of the faults may not necessarily be catastrophic for the system and could only affect a part of CMP, a single fault in a critical component could render the entire chip useless. Such examples include a failure in CMP's core processor or inter-processor NoC which could lead to protocol-level deadlocks, or even partition away vital components such as the memory controller or other critical I/O. A wearout-decelerating scheme involves the utilization of deterministically generated exercise vectors which can activate the wearout-sensitive components during CMPs idle periods.

1.2 Thesis Objectives

The main objectives of this thesis are to:

- Investigate parallel fault simulation for shared memory on-chip homogeneous CMPs, that is able of maintaining its scalability as the number of processing cores utilized increases.

- Explore parallel test pattern generation for shared memory on-chip homogeneous CMPs, that takes advantage of low cost shared memory communication scheme and targets to minimize the test inflation problem by explicitly avoiding duplicate work.
- Extent the parallel test pattern generation method to efficiently generate multiple-detect (*n*-detect) test sets.
- Investigate reliability techniques where deterministically generated exercise vectors can be utilized to prolong CMPs lifetime.

1.3 Thesis Contributions

The contributions of this thesis are:

- **Impact of Input Partitioning in Fundamental Parallel Test Automation Processes**: Investigate the impact of partitioning regarding the efficiency and the quality of the parallel solutions for fundamental test automation processes. More precisely the impact of fault partitioning and test partitioning are examined for fault simulation and test compaction problems with respect to the overall execution time as well as the compaction efficiency. Experimental results demonstrate the importance of partitioning in both the scalability and test inflation of the parallel solutions.
- **Parallel Fault Simulation**:
  Parallel fault simulation methodology in explored for shared-memory multi-core systems, which optimizes along three parallelization dimensions by taking advantage of high degrees of freedom allowed in the basic/core process. Fault dropping is explored across the cores by explicitly avoiding concurrent simulation of the same fault in more than one core. Guided fault partitioning, efficient workload balancing and utilization of shared resources ensures minimal idle time until the completion of the entire process. The speed-up achieved is comparable to state-of-the-art works and most importantly, scalable, ensuring further performance improvement as the number of processing cores will increase beyond few decades.
- **Test Generation Parallelization**:
  Parallel test generation method for shared-memory on-chip multi-core environments is investigated geared towards high speed-up and test inflation containment. The parallel method is generic and can be applied to previously proposed (not parallel) tech-

niques. The low cost of communication via the shared memory, inherent in the underlying architecture is utilized to coordinate the main steps of the ATPG in order to avoid redundant work and dynamically allocate the workload. At the same time, memory contention caused by multiple cores (threads) accessing shared data is minimized. The obtained experimental results demonstrate the effectiveness of parallel approach in speeding-up the ATPG process and provide comparisons with relevant recent works. Moreover, the parallel method is extended to generate multiple-detect (n-detect) test sets with even better results.

- **Compact Exercise Vector Generation for Improved Reliability**:
  The thesis presents an investigation for a proactive technique, designed to decelerate the effects of aging in the critical components of CMPs such as core processors or NoC. Compact deterministically generated (based on ATPG techniques) exercise vectors are utilized by CMPs during idle times for aging mitigation. Experimental results for NoC and a complex superscalar processor using real benchmarks indicate that reliability can be significantly improved with a small and compact number of exercise vectors.

1.4 Thesis Outline

The outline of this thesis is illustrated in Fig. 1.1. Chapter 2 presents a review of state-of-the-art works for the topics under investigation of this thesis. Chapter 3 investigates the impact of partitioning on fault simulation and test pattern generation parallelization problems and presents findings that guide the work presented in Chapters 4 and 5. Chapters 4 and 5 highlight in details the explored methods and present relevant experimentation for fault simulation and test generation problem, respectively. Chapter 6 incorporates findings from Chapters 4 and 5 to extent the method of 5 to a parallel multiple-detect (n-detect) test pattern generation method. Chapter 7 presents a compact exercise vector generation technique targeting reliability improvement. Parallelizing this process is not the target in the thesis. Instead, the goal is to improve the reliability of the underlying architecture, which is on-chip shared memory Chip Multiprocessors. Finally, Chapter 8 presents the concluding remarks and future work.

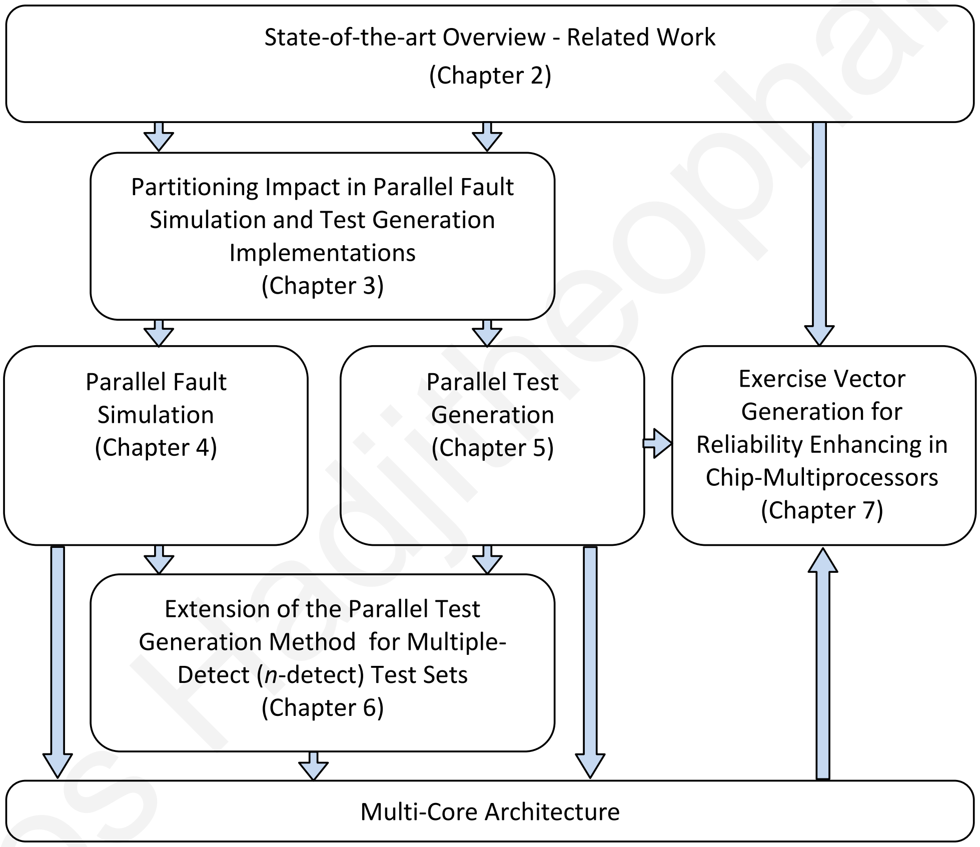

State-of-the-art Overview - Related Work
(Chapter 2)
Partitioning Impact in Parallel Fault Simulation and Test Generation Implementations
(Chapter 3)
Parallel Fault Simulation
(Chapter 4)
Parallel Test Generation
(Chapter 5)
Exercise Vector Generation for Reliability Enhancing in Chip-Multiprocessors
(Chapter 7)
Extension of the Parallel Test Generation Method for Multiple-Detect (n-detect) Test Sets
(Chapter 6)
Multi-Core Architecture

Figure 1.1: Thesis Overview

Chapter 2

Background Knowledge and State-of-the-Art Overview

Over the last several years, o variety of parallel attempts have been proposed investigating traditional test automation problems like the ones examined in this thesis. This chapter summarize related approaches and drafting the most common research directions for the parallel fault simulation (Section 2.2) and parallel test generation (Section 2.3) problems. Also, several techniques have been proposed targeting aging mitigation for multi-core designs. Section 2.4 summarize the main works focusing on reliability enhancement of the underline architecture (presented in Section 2.1) explored in this thesis.

2.1 Underline Architecture of CMPs

Chip multiprocessors offer many opportunities yet there are also many challenges that need to be taken under investigation. A multi-core multiprocessor implements multiprocessing in a single physical chip where designers may couple cores in a multi-core chip tightly or loosely.

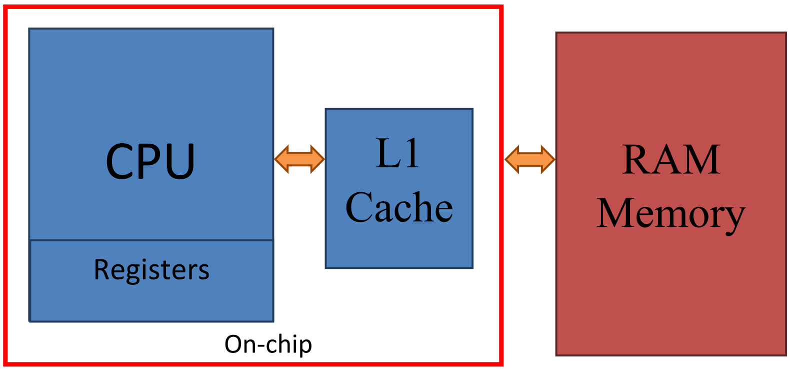


Figure 2.1: Uni-processor architecture.

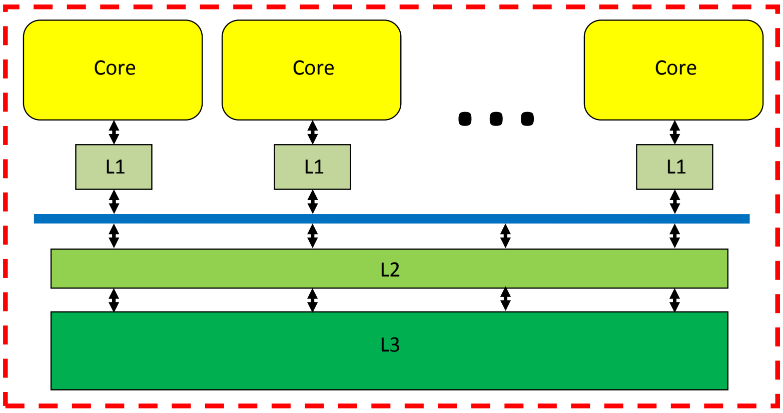


Figure 2.2: On-Chip multiprocessor architecture.

Cores may or may not share caches, and they may utilize message passing or shared-memory inter-core communication techniques. There are several network topologies utilized to interconnect the cores such as the traditional bus, ring, two-dimensional mech or crossbar. Homogeneous multi-core systems consist of only identical cores while heterogeneous multi-core architectures have cores that are not identical (such AMD accelerated processing units that cores do not share even the same instruction set, ARM big. Little where the heterogeneous cores share the same instruction set).

Chip Multiprocessors or CMPs consist nowadays the most popular way to build a high-performance microprocessors for a variety of reasons. The trend of huge and complex uniprocessors has reach a limit for performance scalability mainly due to the limitation on super-scalar instruction techniques. Technology shrinking lead to the compaction of enormous number or transistors per chip resulting in prohibitive power dissipation, costly to design and complex to debug processors. CMPs have recently become a better alternative where the processor die consist of multiple, relatively simpler interconnected processor cores. From the moment a core is designed it can be easily stamped down into more copies to fill the chip area. Fig. 2.2 illustrates an high level example of a CMPs architecture. The low cost inter-processor communication latency between the CMPs' cores can be utilized by multiple threads spreading across the cores to support the computation power needs of many applications [6].

Typical uniprocessors have the following structure (Fig. 2.1) consisting of the CPU, the registers and the cache. Cache is fast small local memory that holds recently used data and instructions. Typical system may be consist of multiple levels of cache. For the case of multiple processors per chip cache coherence problem must be taken under consideration. Memory coherence is a desirable condition where corresponding memory location locations

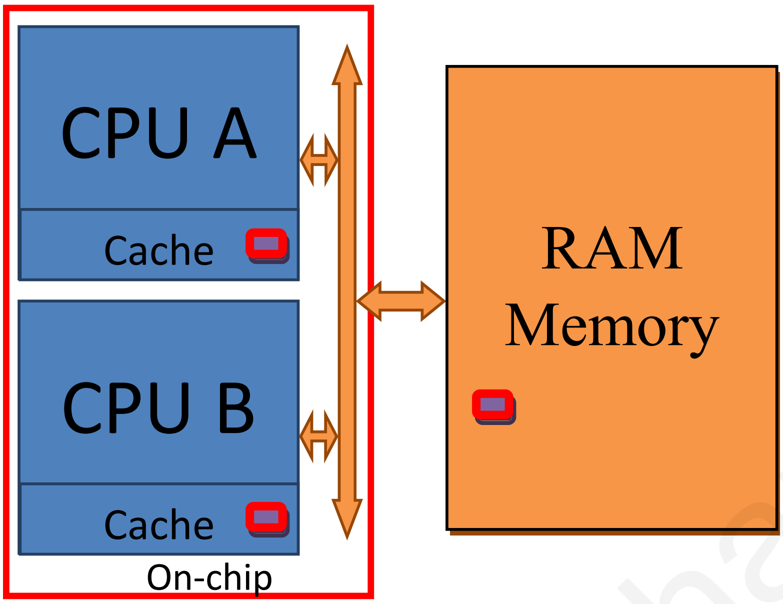


Figure 2.3: Memory coherence problem in multi-processors.

for each processor always contain the same data value in the cash. Without it the running programs will be affected and the results will be wrong. Processors share variables and one of them can write a value while a different processor may be reading the value (Fig. 2.3). One way to alleviate the problem is using a so-called memory coherence protocol that is responsible for notifying all the processors for changes on shared values, thereby ensuring that all copies of the data remain consistent.

2.2 Parallel Fault Simulation

Fault simulation is a fundamental process in Test Automation area mainly targeting the calculation of the fault coverage of a given test set. It can be used either as a standalone tool or as part of algorithms developed for relevant problems such as: test generation, fault diagnosis, techniques for fault tolerant design ([7, 8]). As a result, any performance acceleration of the fault simulation process will also contribute in the performance of many other tools. The most popular techniques can be classify into four main categories based on the underline architecture used for the solution: distributed architectures, vector processors, GPUs and general purpose multiprocessors.

2.2.1 Solutions for Distributed Architectures

Early fault simulation parallelization attempts (distributed solutions) mainly focus on workload partitioning assisted by message passing communication. The performance of those approaches where limited by the high message delivery delays imposed by the distributed inter-chip communication model [9–12]. As a result, communication was intentionally kept minimal and, in essence, the cores where operating independently on a part of the entire problem solution space with communication occurring rarely or even just for the overall solution composition. One of the first attempts was presented in [9] where a hardware accelerator named *MARS* consisting of 15 processing elements was configured in a pipeline. The concurrent fault simulation algorithm proposed was requiring a large amount of memory, which made it impractical to use for large designs. In [10] a dynamic, two-dimensional parallel technique was proposed that extended bit-parallelism of faults to patterns, in order to address large communication delays regarding the faults dropped. The goal was to utilize multiple execution in different processing units by performing multiple fault propagation at the same time. The work of [11] proposed two gate-parallel algorithms for the connection machine that employed 16 processing elements and an elaborated routing network where a message could be sent in 12 routing steps. [12] proposed a fault-disjoint partitioning of the workload in multiple processors (up to 32) based on a static analysis of the distribution of activity in fault simulation. This method performed independent pattern simulations without requiring any communication between processors. The performance of all of the above approaches was limited by the high message delivery delays imposed by the distributed inter-chip communication model. As a result, communication was intentionally kept minimal and, in essence, the processing elements were operating independently on a part of the entire problem solution space with communication occurring rarely or even at the end of processing just for the composition of the complete solution.

2.2.2 Solutions for Vector Processors

Solutions designed for vector processors [11–15] as well as solutions designed for distributed architectures exploit parallelism across three different dimensions: data, algorithmic and structural. Initial parallelization attempts relied on static partitioning of the fault list and/or the test set to compensate for the high communication cost [16, 17].

2.2.3 Solutions for GPUs

The development of multi/many core architectures, necessitate the revisiting of the most effective (yet computationally intensive) techniques in order to be adjusted to these new architectures. Recently, Graphic Processing Units (GPUs) have been exploited [18–24], where the existence of many-cores on GPUs is utilized to create multiple threads in a single instruction multiple data (SIMD) fashion with substantial speed-ups. The work of [19] presented an event-driven logic simulator accelerated by a GPU. When considering gate level netlists, this method achieved an average 3× speedup over traditional serial simulators. [20] proposed an algorithm to map many of the optimizations of serial techniques for fault simulation to the SIMD paradigm achieving a speed-up of 16× with respect to the best serial approaches. [18, 24] proposed timing-aware fault simulation of small delay faults using a data-parallel approach on a GPU with 2880 processing units. The evaluation of the methods was carried out using 10.240 random input stimuli and showed significant improvement in run-time. The authors of [23] exploited GPU's parallel capabilities in order to calculate the *n*-detect fault coverage of a given test with a single traversal of the circuit's netlist. The reported results on a GPU with 128 stream processors showed time reduction by a factor of 25× when compared to a commercial tool. In [22] a fault simulation process is used to accelerate the calculation of the fault table which tabulates all fault detections by each test pattern considered. The algorithm matches pattern parallel simulation with the SIMD-based architecture of a GPU achieving 15× speed-up over the execution of a state-of-the-art serial tool. [21] proposed a method to exploit parallelism in 3 different dimensions i.e., algorithm, model and data. At the same time, it minimized communication between the host processor and the GPU by utilizing the individual's device memory as much as possible.

2.2.4 Solutions for General-Purpose Shared-memory On-chip Multiprocessors

On the other hand, recent general-purpose shared-memory on-chip multiprocessors offer a fast, asynchronous and high capacity memory, in contrast to the SIMD paradigm of GPUs, which can leverage the inter-core communication overhead. A small number of such approaches have been proposed for the highly related problem of test pattern generation [3, 25, 26].

For the specific problem of the parallelization of fault simulation, only the recent work of [1] considers general-purpose shared-memory multiprocessors (as we do in this work).

For the specific problem of the parallelization of fault simulation, only the recent work of [1] considers general-purpose shared-memory multiprocessors (as we do in this work). The proposed technique exploits parallelism both using multiple pattern reasoning and compiled computing model distribution. Moreover, it focuses on utilizing powerful techniques for critical path tracing [27] to accelerate fault propagation. However, because the core process of [1] is highly optimized, the exploitation of parallelization techniques is limited. For this reason and in contrast to the methods we propose in this thesis([28,29]), [1] fails to maintain speed-up gains when the number of processing cores utilized increases. Parallel Pattern Single Fault Propagation (PPSFP) is a popular concept for fault simulation. Exact critical path tracing in multi-core environments has been widely used in combinational and full scan-path circuits for fault simulation. Many researches incorporated PPSFP with sophisticated techniques such as critical path tracing [27,30], dominator concepts [30,31] and stem region [32] with the goal to speed-up the fault simulation procedure. Critical path tracing methods avoid fault simulation for faults within Fan-out-Free Regions (FFR) [27,30]. [33] proposed a modified critical path tracing technique the excludes fault simulation for fanout stems and used various rules to check exactness of critical path tracing beyond the FFRs in linear time. However, the rule based consent cannot be used for concurrent simulation of multiple patterns. Works [34, 35] provide extensions on the work of [33] going beyond the FFRs regions and using stuck-at faults respectively.

Multi-core architectures offer room for improvement for most of the established serial designs and test automation methodologies; however, this improvement depends on the underlining architecture (number of cores and memory availability) as well as on the ability of the proposed approach to scale with respect to the underlining architecture.

2.3 Parallel Test Pattern Generation

Automatic Test Pattern Generation (ATPG), a well-known NP-hard problem, becomes more demanding as devices under test are becoming larger and more complicated and as emerging defects require new fault models of higher complexity. While previously proposed procedures are very effective, see the recent works in [36–38], among many others, they are inherently non-parallel and thus, cannot rely on automatic parallelization using sophisticated compilers. Proper problem decomposition, workload distribution and final test set re-composition are essential to guarantee the quality of the results while maintaining fault coverage. Since, typically, each core does not consider the entire search space, parallel approaches tend to choose local optimal solutions resulting in test set increase [39], known as the *test inflation* problem.

Parallel ATPG has been studied before the on-chip multicore era, by either applying bit level parallelism or distributing ATPG components among multiple processing units, not physically on the same chip [39, 40]. These approaches were designed to avoid/minimize communication overhead and were constrained by the machine's word size. In current on chip multicore architectures with shared memory, on-chip communication is much faster, significantly reducing the cost of inter-core communication. Furthermore, high level of memory coherency is guaranteed and the number of available cores keeps increasing. These new developments and trends motivate towards the investigation of parallel ATPG approaches capable of achieving speed-up scalability as the number of on-chip cores increases, while overcoming new challenges such as shared memory contention, as well as efficient workload distribution parallel threads.

Recent works on ATPG parallelization for on-chip multi-core environments exploit a variety and, often mixture, of parallelism dimensions such as fault parallelism, structural (circuit) parallelism, and algorithmic (including search-space) parallelism. Moreover, the goal of utilizing parallelism often varies. For example, [25] exploits algorithmic parallelism via SAT solver parallelism for maximizing fault coverage with limited speed-up with respect to the corresponding serial process. Similarly, [41] applies bit-level parallelism to generate multiple test patterns concurrently that meet different quality metrics to achieve higher physical-aware n-detect coverage. Static fault parallelism is explicitly considered in [42] using a master-slave architecture to reduce inter-process communication which achieves sub-linear speedup up to 8 cores but suffers from increased test set sizes (test inflation).

Parallelization speed-up rates and test set inflation are investigated in the recent work of [2] which also considers a shared memory architecture model. Shared memory is utilized as an extremely low latency communication mean with high capacity to leverage synchronization and communication of the process. The work in [4] proposes a low communication circular pipeline parallel ATPG procedure which emulates the deterministic execution of a serial ATPG in order to be able to reproduce the same test set every time the parallel algorithm is executed. This leads to limitations in speedup scalability. The series of works in [3, 5, 43] target both parallelization speedup and test inflation minimization strategies, incorporated in state-of-the-art commercial tools. In particular, [5] achieves high speed-up by applying dynamic fault partitioning and depth-first-search based compaction in a shared memory architecture. [43] extends [5] to be used in distributed multi-core hybrid architectures, while [3] incorporates a copy-on write technique for private data protection in order to reduce memory locking when the same part of the memory is used concurrently by more than one cores. Similarly to the above approaches, the exploration of the present thesis targets to achieving high degree of speed-up, as the number of available cores increases, and at the same time limiting test set inflation.

2.3.1 Parallel GPUs Approaches

Some parallel approaches have also been proposed targeting GPUs based architectures. In contrast to the fault simulation problem where the GPU model can be very effective due to its concurrent nature which can directly adopt the single instruction multiple data (SIMD) approach of GPUs [21, 24], GPU-based ATPG has received limited attention [44, 45]. This is mainly attributed to the architecture's memory limitation which leads to unacceptable test set size increase.

2.3.2 Test Compaction Approaches

Two main approaches for test compaction are followed: (i) static compaction which corresponds to the approach where compaction is applied on top of ATPG, while, (ii) on dynamic compaction compaction is applied together with ATPG. For static compaction (i) previously proposed methods follow three main directions, i.e., pattern merging, bit fixing and pattern reordering. Techniques proposed in [46], [47] rely on merging compatible test patterns by utilizing don't care values. Patterns are considered compatible if they do not have conflicting bit values for the same input. Merging does not necessarily referred to combining two patterns into one, but can also be extended to combining three patterns into two. [48] tries to combine bit fixing/alternate with fault simulation and fault dropping in order to catch faults not considered when generating the specific pattern. The target is to eliminate patterns that after fault dropping do not detect faults anymore. Test compaction in the popular ATPG work [49] uses reverse order fault simulation to eliminate tests that do not really contribute to fault coverage.

On the contrary, in dynamic techniques (ii) the compaction is either done as part of the ATPG process [50–52] or starting from a given test set and using an ATPG procedure [47,53]. The works in the former approach invoke different heuristics during test generation in order to meet a pre-computed test set size that is close to the theoretical lower bound that can be estimated solving the problem of maximum independent fault set [54]. The latter approach iterates among the elements of the given test set and performs systematic pattern replacement (using ATPG) with others that detect at least the same faults and either detect some extra faults or have more don't cares. This replacement together with fault dropping and pattern merging result in small test set sizes.

2.3.3 Parallel *n*-detect Test Generation

Extending a test generation method to complex (e.g., defect-aware) fault models can affect the scalability of the methodology as the fault list partitioning will not result in mutually exclusive sub-lists. Another approach to ensure high defect coverage in the manufacturing processes is to generate tests based on simple fault models where each fault is targeted more than one times. Such test sets are known as n-detect test sets (each fault targeted n times) and have been experimentally shown to improve the defect coverage at the expense of an increase test set size [55–60]. Many of these works conclude that the multiple-detections for the same fault should not be generated in an unconstrained manner [55, 57–59]; rather they should have significant difference among them in order to provide high defect coverage. The work of [60] proposed a method to generated compact stuck-at test sets that offer high defect coverage using on a new output deviation-based metric. The work of [61] provided a theoretical evidence that the size of an n-detect test is lower bounded by n times the lower bound of a single-detect test set. Many works have been proposed [62–64] to compact a given n-detect test set to meet the lower bound described in [61] since the vast majority of n-detect test generation methods have been shown to produce much larger test sets.

2.4 Aging Mitigation in Multi-Core Designs

Moore's Law scaling is continuing to yield even higher transistor density with each succeeding process generation, leading to today's multi-core Chip Multi-Processors (CMPs) with tens or even hundreds of interconnected cores or tiles. Unfortunately, deep sub-micron CMOS process technology is marred by increasing susceptibility to wearout. Prolonged operational stress on today's multi-core CMPs gives rise to accelerated wearout and failure, due to several physical failure mechanisms, such as HCI and NBTI. NBTI and HCI are dominant wearout effects and have thus been more intensively studied [65]. Unfortunately, the vast majority of the reported models lack important details, such as values for various constants, measurement conditions, detailed explanation of parameters, etc. Thus, it becomes fairly challenging to employ the existing frameworks, in the context of microarchitecture, to perform meaningful aging effect calculations.

Various techniques have been proposed to mitigate the aging effect in processor core architectures. Among those proposed mechanisms, [66] analyzed the effects of BTI on the clock distribution network with clock gating features in a microprocessor, and then proposed two BTI-Gater cells, to balance delay degradation on the gated clock branch. This technique requires a software sleep scheduling wrapper that works in conjunction with the BTI-gater cells to reduce aging. [67] proposed the Internal Node Control (INC) scheme to reduce the impact of static NBTI on circuits with frequently idle functional units such as adders, subtractors and shifters. INC placements allow outputs of an INC-modified gate to be forced to specific values during sleep mode e.g. exercise various paths to combat NBTI. Those works inspire the investigation of the present thesis targeting to suggest a way to generate exercise vectors and periodically insert them to balance the duty cycle of an NoC router which is a critical component to CMPs.

2.4.1 Chip-Microprocessor Reliability

.

Aging effects are studied under stress conditions to derive relevant micro-architectural models, with the latter being more realistic for high-frequency long-term CMOS operation [65, 68]. Some representative NBTI alleviation techniques from the microprocessors related with the work presented in the present thesis (Section 7.3) include, [69]. [69] introduces the NBTI-aware processor architecture called Penelope and proposed a number of techniques to combat NBTI in various components, including a mechanism that writes special values in memory cells in order to keep the duty cycle at an ideal 50%. [70] suggested the Colt duty cycle equalizer which balances the duty cycle by alternating true and one's complement data representations, while [71] proposed to generate idle periods for BTI recovery by power gating most of the components in a single core processor system. Next, [72] reduced aging in micro-processor pipelines, by replacing the traditional design-time time-balancing pipeline scheme with MTTF-balanced pipelines, also at design-time. The same authors proposed a technique to alleviate NBTI [73] where instructions are classified based upon their execution criticality, directing each into a specialized functional unit, so as to balance the duty cycle in each functional unit by leveraging one at the expense of the other, while on the middleware level, the same method leverages specialized NOP instructions to achieve maximum NBTI relaxation in processors [74].

2.4.2 Architectural-Level Techniques

Further works focus upon architectural-level reliability models. Research in [75] proposed such a model of a processor core, which considers a set of failure mechanisms, assuming uniform failure rates across specific components, however restricting the accuracy of the model when extended to the entire chip. [76] further develop this concept and introduce *effective defect density* and *effective stress condition* coefficients that weigh the failure impact across the chip area and run-time respectively. Last, [77] propose an NBTI mitigation method by rejuvenating the logic along NBTI-critical paths that are first identified hierarchically. Using SPICE simulations a detailed model for computing NBTI-induced delays is first captured at the gate level, and fed as input to an evolutionary algorithm to extract critical circuit usage patterns and thereafter create periodic rejuvenating stimuli. However, the effectiveness of the bit patterns in the generated vectors in mitigating NBTI was not evaluated while no discussion of the periodicity of their application was presented. Contrary to their efforts in optimizing the convergence of the evolutionary algorithm, we have proposed deterministic vector generation algorithm through principles of path delay test model and presented optimization techniques to reduce the hardware overhead. [78] proposed and evaluated the vector exercising technique, considered in this thesis, for NoC router. However, its vector generation algorithm is limited to stuck-at fault activation concepts only, leading to larger vector sets and considerably higher hardware overheads for general designs.

2.4.3 Aging in NoC Domain

Aging has been also examined in the NoC domain. [79] propose routing algorithms to mitigate multiple aging mechanisms. They also point out that NBTI plays a major role in NoC router aging, and their routing techniques balance the traffic load across the network to level-out the aging rates among the routers. The approach is reasonable, in that they force the network traffic to detour through routers of low utilization which, on the contrary, accelerates NBTI-caused aging. However, they use these routing techniques for the opposite effect; the routing algorithms are actually designed to reduce the workload onto the routers which exhibit high utilization, which as we show here are not actually the routers likely to exhibit the most stress-related aging. [80] propose a similar technique to the one investigate by this thesis 7, in that it inserts special values to idle arbiters to mitigate NBTI. However, they propose this technique to make arbiters less frequently utilized so as to give these routers a chance to recover from the effects of NBTI, which is actually not necessary applicable to frequently utilized circuits.

In these previous NoC-oriented studies, it is assumed that the NBTI stress time is proportional to the router utilization, however, on the contrary, in the thesis 7.2 we prove that this is not the actual case. Through detailed, gate-level analysis, not found in earlier works, this thesis demonstrates that the duty cycle becomes more skewed when the NoC router is actually under-utilized and not when it is highly- or over-utilized (Chapter 7).

Chapter 3

Impact of Partitioning in Parallel Fault Simulation and Test Generation

This chapter outlines concepts and motivation for the efficient decomposition of fundamental test automation problems such as Fault Simulation and Test Pattern Generation in order to explore the parallelism without compromising the quality of the results with respect the existing non-parallel solutions. Section 3.2 highlights the important role of partitioning in parallel fault simulation problem, while 3.3 summarize the important role of partitioning in parallel test generation problem. A general parallel test set compaction framework (Section 3.3.1) developed to investigate the impact of various fault partitioning techniques. Results clearly highlight the importance of partitioning for the scalability efficiency and the quality of the results for the problems under consideration.

3.1 Preliminaries

The developments of the past decade in multi-core architectures together with the technology shrinking allowed the realization of (even low-end) computing systems with multiple processing units with sharing memory, at different levels and with different schemes. In order to fully exploit the parallelism offered by these systems, the software model used should be modified to fit the new structures. On the other hand, special purpose applications that consider difficult problems such as those examined in EDA are constrained by the per-processor speed improvement freeze. The latter makes the transition towards parallel approaches for these problems a necessity. The rapid evolution of multi-processor systems will soon allow the realization of architectures with more than twenty cores (known as many-cores), available first for scientific applications. Since most EDA problems are NP-hard and the existing solutions involve processing-demanding heuristics, many-core architectures give rise to a very promising potential.

Test automation engineers may count on automatic parallelization of traditional methodologies using modern compiling approaches together with dynamic scheduling of tasks [81], [82]. However, this does not guarantee the higher efficiency of the obtained parallel procedure mainly because of the generic nature of the compiling tools. Compilation and scheduling can have a great impact on how the problem is partitioned and passed to the individual processing units. For example, for the ATPG problem the selection of the circuit's netlist partitioning technique may give high variations in test generation time depending on the distribution of easy-to-detect or redundant faults among processing units. The quality of the obtained result can be also affected since the algorithmic approaches tend to choose a local optimal when they don't consider the entire netlist [39, 40, 83]. In any case, custom parallelization and/or from scratch development of a parallel solution can benefit even more from automatic parallelization tools run on top of them.

Several researchers have examined parallel solutions for test automation problems, with the higher attention given in fault simulation and ATPG [5, 40, 41, 83]. The older techniques [40, 83] do not consider shared memory, rather distributed memory with the processing units not physically being on the same chip. While inter-processor communication is the main bottleneck for these systems, there are a number of issues regarding the problem decomposition and recomposition that remain equally important to shared memory architectures. Even for distributed structures like these, interesting solutions have been proposed for the techniques' main steps such as fault list partitioning, circuit partitioning, search space partitioning as well as final solution composition. Newer techniques considering shared memory architectures [5,41] have mainly utilized the inter-core communication reduced cost on their benefit in order to archive dynamic load balancing among cores and/or broadcast indeterminate results achieving reduced overall processing time and avoiding getting trapped at local optima.

3.2 Partitioning impact in Parallel Fault Simulation

Multiprocessing architectures can accelerate the performance of fault simulation process. The accelerating factor depends on the scale of the underlying architecture (i.e. the number of processing elements) and, perhaps more importantly, on the scalability of the implemented

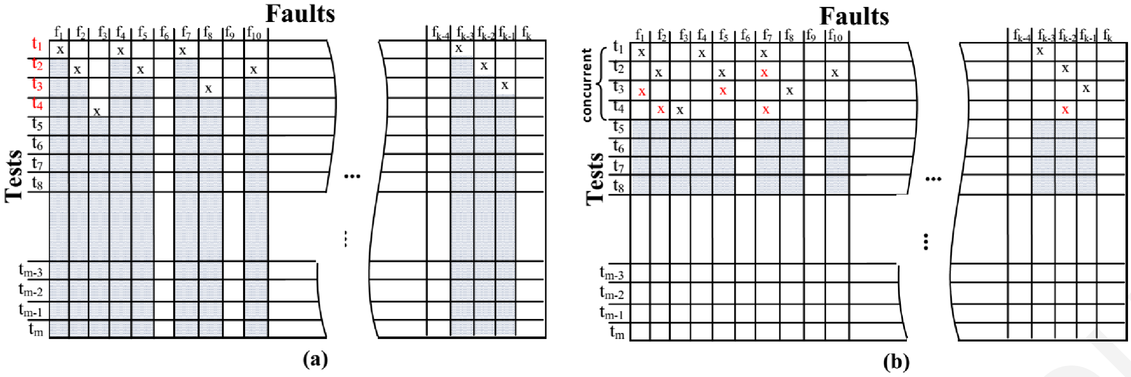

Faults
Tests
concurrent
f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 f_{10}
f_{k-4} f_{k-3} f_{k-2} f_{k-1} f_k
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
t_{m-3} t_{m-2} t_{m-1} t_m
x
(a)
(b)

Figure 3.1: Fault simulation example: (a) Serial fault simulation with fault dropping, (b) Straightforward parallelization 4 cores.

parallel algorithm(s) with respect to the architecture. Among the most important factors is the partitioning of the fault list.

Fig. 3.1 is used to highlight the main challenges for fault simulation parallelization. The Fig. 3.1(a) shows the faults detected after the first four tests (t_1-t_4) are simulated in a serial fashion. Patterned cells indicate no need for simulating the corresponding combination, due to fault dropping (e.g., f_4 is detected by t_1 and not considered by any other test thereafter). Again, this process characteristic can affect parallel execution since two different tests may simulate the same fault(s) at the same time in two different cores. When both tests detect the common fault(s), neither of them can take advantage of fault dropping as the two processing cores operate independently. Fig. 3.1(b) shows how the serial case of Fig. 3.1(a) could be executed on a 4-core system after a straightforward parallelization where each core considers a different test. Cells with a red x show combinations examined in the parallel but not in the serial execution. In such cases, achieving the potential 4× speed-up is not possible as the total workload in the parallel execution is higher than the one of the serial case. One way to minimize this extra workload is to rely on the shared memory for communicating as fast as possible the detection of a fault so that it is dropped from further consideration. However, this can potentially increase the execution time due to the memory coherence mechanism. When two or more cores access the shared fault list to mark the same fault(s) (or faults within the same memory block) memory blocks should be moved from local caches to main memory and vice-versa to keep local cache copies consistent with the content of main memory. In addition, it can create race conditions requiring memory locks to synchronize the shared recourses. For example, in Fig. 3.1(b), marking fault f_7 during the concurrent execution gives rise to such undesired situation. If all three cores simulating t_1, t_2, and t_4 attempt to mark f_7 at the same time, delays are introduced due to memory locks. Furthermore, if multiple cores mark f_7 within a short interval, delays are introduced in order to keep local cache copies consistent in all cores. This issue intensifies in approaches where the basic process considers concurrently a group of faults, such as in the case of bit-parallel simulation. This situation cannot be completely prevented but it should be minimized in order to achieve maximal speed-up. Hence, the same fault should not be concurrently considered in more than one core, even for different tests.

The above guidelines correspond to the two dimensions of parallelism: (i) fault list partitioning and, (ii) test set partitioning. In both cases partitioning comprises the decomposition step responsible of distributing workload among the cores. For the case in (i), cores consider the global test set T and can potentially simulate concurrently the same test but for different faults in F. This requires multiple simulations of the fault-free values for the same test, adding overhead to the approach. Partitioning the test set instead, as in case (ii), allows for a global fault list F in the shared memory where the main overhead is to preserve its coherence. In this case, there is no guaranty that two different tests in two different cores cannot simulate the same fault. While approach (i) adds prohibitive overhead to the process, approach (ii) adds only a small overhead allowing for exploiting more speed-up. More discussion and all relevant experimentation for the parallelization dimensions and how they affect the fault simulation parallelization efficiency are presented Chapter 4.

3.3 Partitioning impact in Parallel Test Generation

In this chapter we investigate the impact of fault list partitioning on test set compaction for multi-core architectures [84]. Specifically, a highly efficient non-parallel test set compaction algorithm is executed on sub-sets of the examined circuit's fault list running on different on-chip processing cores simultaneously. The experimentation is exploring two different parameters of the problem: i) the fault list partitioning approach and ii) the number of the fault sub-lists allowed. The obtained results show smaller compaction of the considered test sets as the number of sub-lists (each corresponding to a processing core) is increased. At the same time, the processing time is significantly reduced suggesting a close to linear (to the number of cores used) improvement. For comparison purposes, we also present results (i.e., final test set sizes and CPU time required) for a non-partitioned approach that each different core considers the entire fault list. For this case, the compaction results are closer to the serial execution while the CPU times are not as good as the partitioning approach.

Test compaction

Test set compaction is proven to be NP-hard [54] and is a significant component of the vast majority of ATPG methodologies targeting one of the most important problems of EDA [85]. The problem examined in this thesis and publish in [84] is that of obtaining reduced size test set given a desired fault coverage, under a specific fault model for the circuit-under-test (CUT). Since test compaction is a well studied problem we here give an abstracted definition of the problem in tight relation with the ATPG problem to help understand previous works as well as better define the problem considered in this thesis.

Definition 1: Given a circuit C, a fault list F (under a specific fault model M) and a desired fault coverage $fc(\leq 1)$, an ATPG process D gives a set of input patterns T called test set. When the patterns of T are applied to C they give different than the expected output value(s) in the presence of a single one of at least fc$\cdot|F|$ faults as defined in fault model M.

Definition 2: Given a circuit C, a fault list F, a desired fault coverage fc and an ATPG process D that gives test set T, a compaction process Z applied in conjunction with D gives a test set T' with at least fc fault coverage and with $|T'| < |T|$. While according to this definition a compaction process suffices to result in barely smaller size test set, practically the size of T' should be considerably smaller in order to make sense to develop and apply such a process.

While all these techniques are highly efficient they can benefit from parallel execution. However, converting them to parallel implementations is not a trivial process and has to be done based on each algorithm's characteristics via for example search space or algorithm partitioning. The focus of this thesis is more generic and can be applied to previously proposed (not parallel) test compaction methods. The idea is to partition the fault list based on the available processing units and measure the effect of this partitioning both in the execution time as well as the compaction quality. For this, we consider the modified version of the problem below:

Definition 3. Given a CUT C, a fault list F, an ATPG process D and a compaction process Z that given a test set T with fc fault coverage and a set of processing units U, divide F in $|U|$ sub-lists F_1, F_2,...$F|_U|$ such that upon application of D and Z on each F_i test sets T_i are obtained with $\bigcup T_i$ having fc fault coverage and $\sum |T_i| = |T|$.

While the desired solution to the problem utilizes the multiple processing units, the main ATPG and compaction processes D and Z remain unchanged. The compaction method Z is just applied to a restricted sized input. Since the new definition does not rely on the nature of D and Z the problem is generic focusing only on the fault list division and its solution can be used to convert compaction algorithms to parallel. The selection of the fault list division technique can affect both the efficiency of the solution as well as the quality of the compaction achieved. For example, if fault list is partitioned into $|U|$ sub-lists and algorithms are applied in different processing units for each sub-list, we expect to get execution time $|U|$ times smaller than that of the single processing unit case. However, the compaction quality would be naturally compromised since the compaction process does not consider the complete fault list, as it does at the single unit case, leading to local optimal solutions where $\sum |T_i| = |T|$.

3.3.1 Fault List Partitioning for Dynamic Test Set Compaction

We evaluate the impact of fault list division as a plug-and-play procedure in the process of converting a serial algorithm into a parallel one. For this we have used the efficient dynamic compaction method of [86] on top of two simple, yet extreme division approaches; the first one partitions the fault list into a number of sub-lists equal to the number of available processing units (cores); the second one considers the entire fault list (kept in the shared memory) in all cores. This section describes these two fault list overlap extremes (former approach has no overlap, latter has 100% overlap). For the partitioning approach different techniques are described. The second part of this section gives a brief description of the compaction algorithm used and a detailed explanation of the parallel scheme used for the study.

Fault List Division

The full overlap approach of fault division is straight-forward: all cores see the entire fault list (as does the non-parallel algorithm). Yet appropriate shared memory locking should be used in order to avoid consideration of the same fault from different cores. Since the compaction method of [18] is dynamic, considering the same fault by multiple cores will give higher final test set size. The worse case scenario where all cores considers all faults at the same time in the list is obviously meaningless to examine as it is identical to have the same execution multiple times. Thus, the fault list is centrally maintained and cores obtain faults to consider in a first-come-first-serve fashion, marking them appropriately in order to be dropped from further consideration.

For the non-overlap division, three different partitioning techniques are under investigation for fault distribution across the cores (n) considered processing units. Fig. 3.2 describes the three fault list partitioning techniques planned, i.e., i) in order (topological) partitioning, ii) modulus-n partitioning and iii) random partitioning. In all three cases the n faults in F are considered in a topological circuit order (upper part of Fig. 3.2). When considering *in order*

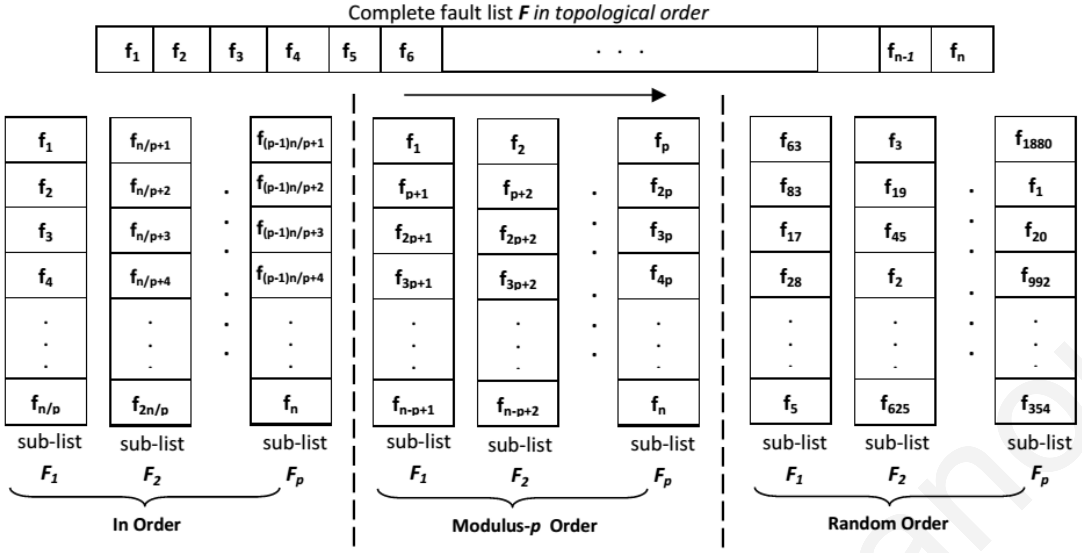

Complete fault list F in topological order
f1 f2 f3 f4 f5 f6 . . . fn-1 fn
f1 f2 f3 f4 . . . fn/p
fn/p+1 fn/p+2 fn/p+3 fn/p+4 . . . f2n/p
f(p-1)n/p+1 f(p-1)n/p+2 f(p-1)n/p+3 f(p-1)n/p+4 . . . fn
sub-list F1 sub-list F2 sub-list Fp
In Order
f1 fp+1 f2p+1 f3p+1 . . . fn-p+1
f2 fp+2 f2p+2 f3p+2 . . . fn-p+2
fp f2p f3p f4p . . . fn
sub-list F1 sub-list F2 sub-list Fp
Modulus-p Order
f63 f83 f17 f28 . . . f5
f3 f19 f45 f2 . . . f625
f1880 f1 f20 f992 . . . f354
sub-list F1 sub-list F2 sub-list Fp
Random Order

Figure 3.2: Fault list partitioning methods under investigation

partitioning, the topological order is followed for the partitioning. Specifically, fault sub-list F_1 holds the first $|F|/n$ faults found in topological order, F_2 the next $|F|/n$ faults, and so on (left column of Fig. 3.2). In *modulus-n* order the faults are partitioned in a modulus fashion, using the topological order. Hence, fault list F_1 gets the first fault f_1, F_2 gets f_2 and so on with F_n getting fault f_n. The next fault (f_{n+1}) is included to F_1, f_{n+2} to F_2 and so on (center column of Fig. 3.2). In *random order*, faults are randomly selected to be included to each sub-list each of which contains (as in the previous two cases) $|F|/n$ faults (right column of Fig. 3.2). Each one of these fault sub-lists is then assigned to a processing unit for executing the test set compaction process.

General Parallel Compaction Framework

The general parallel test set compaction framework developed to investigate the various fault partitioning techniques, utilizes the dynamic test set compaction process proposed in [86] on the individual fault sublist proposed above. This process has been shown to be very successful for generating compact test sets with low number of specified bits, in a non-parallel framework. Recall, that the objective is to keep the original test compaction process intact in order to measure the effect of fault list division. The compaction process of [86] targets a set of faults at a time with the explicit purpose of reducing as much as possible the initial test set, without any loss in fault coverage. In essence, the test set compaction finds pairwise fault compatibilities, in an iterative manner, so as to find a minimum number of compatible sets of faults. Each such set is used to generate a test that detects all its contained faults. In this context two or more faults are considered compatible if and only if they can be tested by

```
parallel_dyn_compaction($C, F, p$)

Inputs:  circuit $C$, fault list $F$, available cores $p$
Output:  compacted test set $T$
01: $S(F)$=static_partition_($F$, $p$, method)
02: % method: in order, modulus-p, random, none
03: $S(T) = \emptyset,\ \ T = \emptyset$
04: for each $F_i \in S(F), i = 1, 2, ...p$
05: do in parallel % start of parallel execution
06:     $T_i$ = ser_ATPG($F_i$)
07:     $T_i'$ = ser_dyn_comp($F_i, T_i$)
08:     $S(T)$ = $S(T) + T_i'$
09: % end of parallel execution
10: for each $T_i \in S(T), i = 1, 2, ...p$
11: do
12:     $T$ = $T \bigcup T_i$
13: return $T$
```

Figure 3.3: General parallel compaction framework

a common test.

The general parallel framework proposed for the investigation of the various partitioning techniques is detailed in Fig. 3.3. First the fault division procedure is invoked to produce p different fault sub-lists, where p is the number of available processing units (cores). The fault lists are saved in set collection $S(F)$. Lines 04 to 08 applies the serial test compaction algorithm (line 07) on top a trivial ATPG process (line 06) for each list in $S(F)$ in a different core, in parallel. The ATPG process of line 06 corresponds to a simple test-per-fault test generation in order to minimize its effect to the overall process and focus on the test set compaction process. At this point, note that the dynamic compaction inherently preserves the fault coverage. We avoid to include a constraint on the fault coverage in order to keep the framework simple. The test-per-fault procedure implies a 100% fault coverage. The resulted test set T_i' from each core is saved in set collection $S(T)$ kept in shared memory. Lines 10 to 12 take the union of all test sets in $S(T)$ in order to provide a test set of 100% fault coverage on F (duplicates are eliminated). Recall that according to Definition 3 a valid solution of the defined problem will give a T with size equal to the size of the test obtained when the same serial process is applied on F. Obviously, we expect the fault list partitioning to give an increased sized T.

The Dynamic Compaction Framework (Fig. 3.3) can be easily adjusted to accommodate the parallel execution of the majority of static or dynamic test set compaction procedures. Moreover, the framework allows for the unbiased evaluation of the proposed fault partitioning

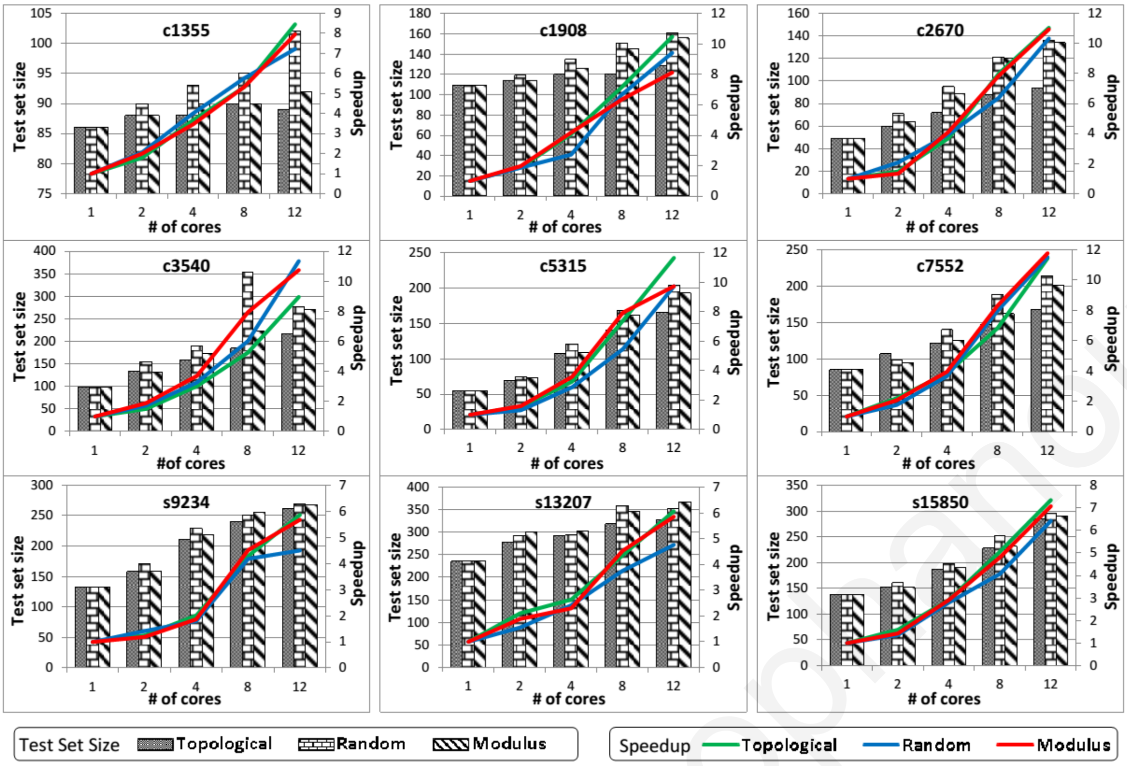


Figure 3.4: Exploration of various partitionings and their affect in test set size and speed-up

techniques, as it does not utilize any knowledge on the internal structure or the characteristics of the considered compaction algorithm.

3.3.2 Experimental Results

The proposed algorithm has been implemented using C++ language and run on a 12-core 1,798 GHz AMD Opteron 6166 HE, running Linux with 32GBs of RAM. The test generation and test compaction processes used are from an in house BDD-based tool described in [86]. The parallelization framework proposed in Section III was implemented using the OpenMP parallel programming model. We experimented with all ISCAS'85 and ISCAS'89 benchmarks circuits, since BDD order files are available only for these circuits. Fig. 3.4 illustrates results for the three partitioning methods described in Fig. 3.2 for 9 representative circuits for different number of cores, namely 2, 4, 8 and 12. The bars report test sets sizes obtained for the three different partitioning methods as shown in the legend below and recorded in the left axis of each chart. As expected, partitioning the fault list has a considerable impact on the quality of the compaction giving larger test sets. This is attributed to the fault sub-list considered by each core that tends to lead the algorithm to local optimal solutions. In some cases the impact is much larger than expected resulting in unacceptable test set sizes (see c3540 and c5315 for example). Out of the three partitioning techniques considered, the

TABLE I. MEASURING THE EFFECT OF FAULT PARTITIONING TO TEST SET COMPACTION FOR 12 CORES

		Non-parallel		In-order Part.			Random Part.			Modulus-p Part.			No Partitioning		
Circuit	**Faults**	$\|T\|$	**spdup**	$\|T\|$	**inc**	**spdup**	$\|T\|$	**inc**	**spdup**	$\|T\|$	**inc**	**spdup**	$\|T\|$	**inc**	**spdup**
s1196	1334	127	1×	191	150%	7.6×	218	172%	6.8×	220	173%	6.4×	129	102%	2.8×
s1238	1390	143	1×	199	139%	5.3×	219	153%	4.1×	223	156%	5.8×	143	100%	2.8×
s1423	1513	30	1×	65	217%	8.8×	86	287%	8.0×	88	293%	8.9×	38	127%	1.4×
s1494	1680	114	1×	139	122%	7.7×	154	135%	5.0×	152	133%	6.8×	117	103%	5.3×
s9234	7274	132	1×	261	198%	5.9×	269	204%	4.6×	268	203%	5.7×	141	107%	1.2×
s13207	10456	236	1×	328	139%	6.0×	352	149%	4.7×	367	156%	5.8×	237	100%	1.3×
s15850	12150	138	1×	285	207%	7.3×	295	214%	6.4×	290	210%	7.0×	149	108%	2.2×
c880	994	21	1×	56	267%	9.5×	77	367%	8.0×	70	333%	7.8×	28	133%	2.1×
c1355	1618	86	1×	89	103%	8.5×	102	119%	7.2×	92	107%	8.0×	87	101%	1.3×
c1908	2056	109	1×	129	118%	10.5×	161	148%	9.4×	156	143%	8.1×	113	104%	1.5×
c2670	2954	49	1×	94	192%	11.0×	136	278%	10.3×	134	273%	10.9×	58	118%	1.5×
c3540	3742	101	1×	217	215%	9.0×	277	274%	11.3×	271	268%	10.7×	108	107%	3.2×
c5315	6016	55	1×	166	302%	11.7×	204	371%	9.7×	193	351%	9.7×	60	109%	2.8×
c7552	8080	85	1×	167	196%	11.4×	214	252%	11.5×	201	236%	11.7×	93	109%	2.8×
		Average:			183%	8.6×		223%	7.6×		217%	8.1×		109%	2.3×

one putting the neighboring faults in the same sub-list (in-order partitioning) always gives smaller test sets. The random partitioning is the one that gives more diverse results in terms of test set sizes. The colored lines in Fig. 3.4 present the corresponding speedup with respect to the non-parallel execution of the algorithm (right axis of each chart). At this point, note that in order to minimize the impact of the parallelization overhead code and obtain comparable results, the non-parallel execution of the algorithm refers to executing the algorithm in Fig. 3.3 with p = 1, i.e., one available processing unit and not the actual non-parallel execution. The main observation related to our motivation is the fact that indeed the increase in the number of processing units gives higher speedups (smaller execution times), yet the speedup in most cases is smaller that the number of cores. The main reason for this is that dividing the fault list into sub-list of equal size does not ensure equal load for each core. Changing the partitioning method can slightly affect the speedup, yet in none of the cases examined the speedup follows the increase in the number of cores. Close monitoring of the execution shows that the deviation of CPU times among cores can be large. A possible solution to this would be a dynamic fault partitioning approach where the fault list is partitioned during the evolution of the algorithm, balancing the load of each processing unit.

Table I reports the obtained results for the three partitioning techniques as well as for the non-partitioning parallel execution based on the framework described in Fig. 3.3. After the circuit's name and the size of the entire fault list we show results for the 1-core execution of the algorithm, i.e., the test set size and the CPU requirements. Since this is a comparative study, we use this as a reference for the fault division approaches examined. Next, we present results for the parallel executions using the framework of Fig. 3.3 with 12 processing units under the different fault division approaches. Thus, the faults are divided in 12 sub-lists and assigned to the available cores. We show results for the size of the obtained test sets (|T|), its relative size with respect to the 1-core execution (inc) as well as the corresponding speedup achieved (spdup). Columns 5-7 report the results for the in-order topological partitioning, columns 8-10 for the random partitioning and columns 11-13 for the modulus-p partitioning. The last three columns report the same results for the non-partitioning case. The results confirm that partitioning the fault list using the topological order gives the smaller test set sizes among the three partitioning techniques. At the same time it also favors execution time as it achieves the highest speedup with respect to the non-parallel execution for the majority of the considered circuits. While the speedup is considerable for all three cases, the quality of the compaction is greatly impacted and this cannot be reversed by using an appropriate fault list partitioning technique. On the other hand, the case where the entire fault list is considered by all cores (no partitioning) gives test set sizes very close to the nonparallel execution of the algorithm; however the speedup is small (2.3× on average although 12 cores were used) and is only attributed to the inter-core communication for avoiding considering a fault multiple times. In summary, examining these two extreme cases for fault division (full and no overlap between the sub-lists) gives good results in either the test set size or the execution time, but not both. Since these two outputs are shown to scale well (see also Fig. 3.4) a meaningful approach is to explore hybrid approaches before looking into from-scratch development of parallel techniques.

3.4 Challenges and Findings

The importance of efficient decomposition on parallelization of fundamental test automation problems is evaluated and important findings are taken under consideration for the research topics of this thesis. Particularly, for fault simulation problem partitioning of faults or tests to the available cores can significantly affect the speed-up scalability. Fig. 3.1 b) highlights the redundant work of a straightforward parallel solution compared to a typical non-parallel fault simulation. Moreover, for the case where faults are distributed to the available cores the parallel algorithms needs to take measures to avoid concurrent simulations of the same test for different faults. Otherwise the potential benefit from fault dropping is lost.

For the parallel test generation problem various fault-list division approaches and a non-partition approach where investigated. The three fault-list partitionings investigated (In-order, Modulus-p and Random order) indicated a slightly performance gain from the In-order (topological sort) partitioning mainly due the utilization of structural similarities among the faults. A comparison among the fault list partitioning approach and the approach where the fault list is shared among the available cores showed that partitioning could offer good speed-up results, while shared fault-list offers negligible test inflation. Since both approaches fail to offer significant speed-up gain without compromising the quality of the results a hybrid approach needs to be considered. consideration.

3.5 Chapter Summary

This Chapter presents the main challenges of partitioning in parallel fault simulation problem (3.2). Moreover, a general parallel framework (Section 3.3.1) for evaluating the fault list division for parallel execution of test set compaction algorithm is presented. Two different fault division approaches covering the two extremes was investigated. One, fully partitions the fault list while, the other considers the entire list. Both approaches fail to benefit from the speedup without compromising the quality of the compaction achieved.

Chapter 4

Parallel Fault Simulation for Shared-Memory On-chip Multiprocessor Architectures

This chapter outlines basic concepts and terminology (4.1) for the fault simulation problem. Section 4.2 presents in details the parallel fault simulation methodology explored in this thesis, along with the experimental results. Results indicate the significance of the proposed parallelization attempt on modern On-chip Multiprocessor architectures.

4.1 Preliminaries and Basic Concepts

A brief overview of the basic concepts, notations and terminology are presented in the section that would be useful for the reader in the following sections.

Definition 4.1.1. ***Defect****: A defect is the unintended difference between the implemented hardware and its intended design. Defects can occur either during manufacture or during the use of devices.*

Definition 4.1.2. ***Error****: is an effect of a defect. Usually is a wrong output signal produced by a defective system. Nano-scale circuit manufacturing processes with the nowadays high performance requirements may involve different types of variations (small deviations in physical shape of gates or interconnections) [87]. These deviations can cause many types of faults.*

Definition 4.1.3. ***Fault****: is a representation of a defect and are represented by fault models.*

4.1.1 Fault Models

Defects can be numerous on nowadays circuits and often not able to be analyzed, so models are used to represent a fault that is likely to occur in a circuit. Fault models are used for simulation, thus expensive hardware circuit manufacturing is avoided. Fault tests are derived based on fault models and are used to identify those faults. Also, test generation tools can have limited number of modeled faults (targets) for testing, while, fault simulation tools are used to evaluate the effectiveness of the tests generated (fault coverage). There are different type of faults like single/multiple stuck-at faults, CMOS stuck-open, bridging faults, structural faults, segment-delay fault, transition faults among others. Many fault models exists in the literature. Stuck-at fault is the most popular among them where stuck-at fault is modeled by assigning a fixed (*0* or *1*) value to a signal line in the circuit. A signal line is an input or an output of a logic gate or a flip-flop. The single stuck-at faults model consist of two faults per line, stuck-at-1 (commonly written as s-a-1 or *sal*) and stuck-at-0 (commonly written as s-a-0 or *sa0*). A line with a *sa0* fault will always will have logic value '0' irrespective of the normal line value. In general, several stuck-at faults can be simultaneously present in the circuit. A circuit with *k* lines can have 3^k-1 possible stuck line combinations. This is because each line can be in one of the three states: *sa0*, *sa1*, or fault-free. More details regarding the fault models can be found on [88].

4.1.2 Logic and Fault Simulation

In the test automation area logic simulation serves two purposes. It can either be used to verify the correctness of design according to specifications (true-value simulation), or can be used to evaluate tests (simulation in the presence of faults). True-value simulator computes the response for a given input assuming that no faults exist in the design (specification). Given that the responses are known if any errors are identified then the design needs to be changed until the responses will match the specifications. On the other hand, fault simulation is a fundamental process in EDA which can be either used to determine the coverage of a given set of (test) vectors for a given list of faults, either as part of algorithms developed for other processing demanding methodologies targeting relevant problems such as: test generation, fault diagnosis, techniques for fault tolerant design [7, 85, 89]. **Fault coverage** is defined as the ratio of the number of detected faults compared to the total number of faults in the fault list.

Initially, all vectors are simulated in true-value simulation mode and stored for a typical

Table 4.1: Notations used in Chapter 4

C:	Netlist of the circuit
UF:	Undetected Faults
RT:	Remaining (non-released) Tests
F_w:	List with w faults selected for simulation
T_w:	Group of *w* tests
T_i:	Test set partition assigned to core *i*
F_i:	Fault list partition assigned to core *i*
Q_i:	Local (in core) queue for core *i*
F_{di}:	Local (in core) list for core *i* holding detected faults
di:	Locally detected faults
TFM:	Shared tests and faults map, consisting of tests and corresponding non-simulated faults

serial fault simulator in its simplest form. Then, faults are iteratively introduced in the circuit (by modifying the circuit's description). Vectors are then simulated and the output values are dynamically compared with the stored true-value responses and if the comparison indicates a difference then this is also an indication that a fault is detected. A complete fault simulation methodology considers, in the worst case, each fault in a given fault list F for every vector in the given test set T and, hence, the well-known worst case complexity of a serial method is $O(m\cdot(k+1).k)$, where m is the number of tests in T, k is the number of faults in F, and k is the number of nodes in a circuit's netlist.

Bit-parallelism

model of parallelization can speed-up the fault simulation time by w times, where w corresponds to the machine's word size, using bit-parallelism of logical operations. This allows simultaneous simulation of w faults or tests. Due to its widespread usage there exists many more intelligent forms of fault simulators based on reasoning such as the deductive [90], concurrent [91] and differential fault simulators [92]. Those simulators allow to collect all detectable faults by a single run of the given test pattern, however, they cannot produce reasoning for many test patterns in parallel.

4.1.3 Motivation and Considerations for Parallelization

Fault simulation is the problem of identifying the percentage of modeled faults detected, when applying a given set of test patterns at the inputs of an integrated circuit. This percentage is known as the fault coverage of the circuit under test considering a fault list (F) generated for a specific fault model. The set of input patterns to be tested, known as the test set (T) can be generated using a deterministic or a random test generation process [7]. In this thesis we consider the problem of parallelizing the fault simulation under a shared-memory multi-core system so that the obtained parallelization speedup, when compared to a serial version of fault simulation, is maximized.

In general, parallelization methods follow a three step approach where the given problem is (i) appropriately partitioned (problem decomposition), (ii) in parallel solved for each partition (parallel execution) and (iii) reassembled to form a final solution (solution re-composition). Apart from the inherent overhead incurred in issuing and, synchronizing multiple threads, decomposition/re-composition steps involve other challenges such as communication overhead, conflicting data in shared-memory and race conditions between processing units. All those challenges must be taken into account from the proposed parallel solution.

Parallelization of the fault simulation process is not a straightforward task as it relies, among others, on appropriate bookkeeping to avoid duplication of work as well as well-scheduled utilization of the available cores. Automatic parallelization tools and sophisticated compilers cannot provide the desirable speed-up since the problem is not inherently parallelizable. As it is presented in Chapter 3 for computational intensive tasks the problem partitioning is of significant importance. One approach is to rely on standard parallelization tools that partition the problem under consideration following generic partitioning rules. However, this results in local optimal solutions due to the limited consideration of the problem's entire solution space. This can severely affect the effectiveness of the proposed solution, especially for architectures with a large number of processing cores. Thus, the methodology proposed here explicitly addresses problem decomposition as well as the final result assembly. At the same time, it proposes solutions for two critical processes used to resolve the *challenges* in partitioning the problem, i.e., fault dropping and workload balancing. The following section discusses attributes of these challenges and their effect on the efficiency of the entire parallel fault simulation process.

Efficient fault dropping: Fault dropping can considerably affect the efficiency of a fault simulation method. For a complete fault simulation method each test from a given test set

(*T*) must be simulated for every fault in the considered fault list (*F*). Hence, the well-known worst case complexity of a serial method is O(|*T*|·(|*F*|+1)·m), where m is the number of nodes in the netlist, implying a circuit traversal for each fault-test combination. Detected faults are discarded (dropped) from *F* and are not taken into further consideration by any of the subsequent tests and, as a result, the actual runtime is improved considerably below this bound. However, for parallel fault simulation solutions the effectiveness of the fault dropping is not effortlessly maintained. Inefficient parallel solutions can result in *two or more tests that are simulated for the same fault more than once*, increasing in this way the overall workload. Moreover, as opposed to serial solutions, where detected faults are immediate dropped from further consideration, fault dropping in parallel solutions might also be delayed (memory locking, coherency) or tests/faults could be reassigned among cores, destructing the bookkeeping. When two (or more tests) detect the same fault(s), then, the potential benefit from fault dropping is small. In this sense simulating the same fault in parallel should be avoided at the extend possible.

Effective partitioning: The well known three step parallelization approach consisting of (i) problem partitioning, (ii) parallel solution for each partition and (iii) reassembled to form a final solution in not straightforward to be followed for fault simulation as discussed in Section 3.2. Appropriate bookkeeping for decomposition and re-composition steps along with well-scheduled utilization of available cores and shared memory during parallel execution are very important for the efficiency of the parallel method. Problem decomposition, either by partitioning of faults or by partitioning of tests (or both faults and tests), and distribution among available cores are very popular parallelization methods [39, 40]. Even though static partitioning can contribute in speed-up gain, it is not adequate to provide a scalable solution because of the overhead imposed by the constraint that *no test should be simulated for the same fault more than once*.

We explain this with the illustration of Fig.4.1. The fault simulation search space is represented as a table, where rows correspond to tests to be simulated and columns to the faults considered. Fig.4.1(a) shows how it is explored by a typical serial fault simulation procedure that simulates tests one by one for all faults in a predefined order (i.e., t_1, t_2,..., t_9). A complete fault simulation should examine all cells in the table. An **x** mark indicates that the corresponding test detects the corresponding fault (e.g. t_1 detects f_2 and f_6); any additional simulations (and possible detections) of an already detected fault is considered as redundant work for traditional fault simulation process e.g., where only the calculation of fault coverage is required. Hence, a fault detected by a test is dropped from further consideration (shaded

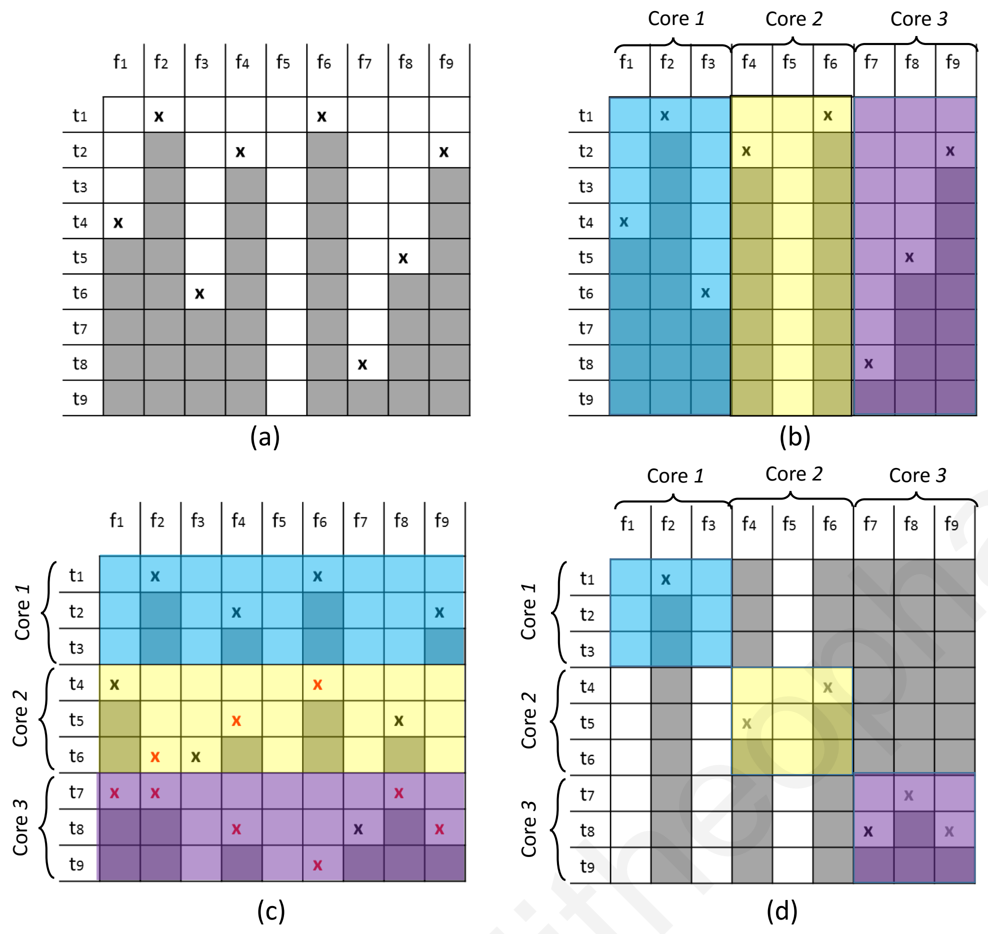


Figure 4.1: Impact of fault dropping in different approaches (a) serial fault simulation, (b) static fault partitioning parallelization, (c) static test partitioning parallelization, (d) hybrid test and fault partitioning parallelization (snapshot, not shown complete here).

cells). For the example in Fig.4.1(a) 43 out of 81 cells are not examined, because of fault dropping.

Fig.4.1(b)-(d) illustrate various partitioning options for parallel fault simulation. In Fig.4.1(b) the faults are distributed across the available cores (fault partitioning) and simulated for all the tests. Thus, each core (represented with different color) simulates all the tests for a fraction of all faults ($|F|/m$). While this approach is complete and the total number of cells examined is the same as in Fig.4.1(a), it is not scalable as expected. Since the percentage of dropped faults is different for the different cores, some cores terminate faster than others. For example, *Core 1* becomes idle after 11 simulations in total (non-shaded cells), *Core 2* after 12 and *Core 3* after 15 simulations. Hence, the total execution time is limited by the slowest core (lowest percentage of dropped faults) as the idle cores' processing power is not utilized. In the example of Fig.4.1(a) the total execution time is 38 time units (white cells), assuming a uniform 1 time unit execution for each test-fault pair examined. A scalable parallelization with 3 cores should provide 3× speed-up i.e., 12.67 time units. Instead, the approach of Fig.4.1(b) is executed in 15 time units (waiting for *Core 3* to finish execution) resulting in only 2.53× speedup.

A second partitioning option is when the tests are equally distributed to the available cores while all faults are considered by all cores (test partitioning) as shown in Fig.4.1(c). This way, each core has to simulate only a fraction of the available tests ($|T|/m$), for all faults in *F*. The non-shaded cells in Fig.4.1(c) but shaded in Fig.4.1(a) indicate unnecessary simulations (22 cells) while a red × indicates additional fault detection (not present in the serial approach). These additional detections occur since multiple tests are concurrently simulated for the same faults (e.g. f_6 is detected by t_1, t_4 and t_9 in all 3 cores). The total number of simulations is 60 instead of 38 in Fig.4.1(a), affecting significantly the expected speed-up. For this example, the execution time is 21 time units (execution time of Cores 1 and 2) resulting in a 1.81× speed-up instead of the 3× expected. Even for the case where each core broadcasts the identified detections (i.e., perform fault dropping across cores via the shared memory), experimentation showed considerable overhead due to undesired situations such as inconsistent race conditions, memory locks, and synchronization.

A hybrid approach combining the two approaches to explicitly avoid potentially unnecessary work is shown in Fig.4.1(d). Each core is assigned a subset of the fault list of size ($|F|/m$) and a subset of the test patterns of size ($|T|/m$) (test and fault partitioning) at one time. This process is iterated until the entire search space is examined. The faults detected at each core are dropped at the end of each iteration. This hybrid partitioning allows for fault dropping to be communicated more frequently and for better workload balancing. For these reasons it has been adopted in the proposed method. More details are provided in Section 4.2.

Superfluous work avoidance: The efficiency of parallel solutions is very sensitive to the uncertainty introduced by fault dropping, shared memory access time, race conditions, synchronization and the shared memory coherence mechanism. Parallel methodologies inherently suffer from increased total workload with respect to serial approaches. The limited view of the entire problem in each core can result in processing that is not necessary because concurrent processing nullifies its contribution to the final solution. Detecting a fault once is sufficient for the problem under consideration and any further consideration of detected faults in any core is considered unnecessary processing. In this work, simulating a fault concurrently in two or more cores for different tests is referred to as *superfluous workload*.

For example, in the serial approach of Fig.4.1(a) the shaded cells denote workload that does not require to be executed due to fault dropping. For the parallel approach of Fig. 4.1(c), where the same fault is considered in more than one cores at the same time, many of these superfluous simulations are executed, increasing the overall executed workload. Superfluous workload is the main reason why test partitioning has a larger impact on the speed-up than fault partitioning. The proposed work incorporates a number of techniques to minimize superfluous workload.

Shared memory utilization: Parallel approaches relying on shared memory are by nature highly dependent on the memory structure. Memory can be effectively used as the main intercommunication mean between processing cores, hence any inefficient usage or inappropriate architecture can severely affect the methodology's performance. Fault simulation is not an exception, since each core must communicate its detections as soon as possible to allow for effective fault dropping. Shared memory access requires memory locking and synchronization in order to avoid race conditions and ensure the memory coherency. The idle processing times imposed by memory locking and synchronization introduce delays, which can increase the overall runtime. This is the case, for example, when the number of faults dropped at one time is large (i.e., high fault dropping rate). Communicating dropped faults very frequently (e.g. on every detection) can increase the memory accesses and the corresponding idle time considerably. This increase has even larger impact on memory hierarchies that combine shared (main memory) and local memories (on-core caches), like the one considered here, where a large number of writes from local to shared memory (and vice versa) are performed. In this work we consider systems with multiple identical cores (homogeneous), each of which posses at least one level of private cache and at least one level of shared memory.

To address this issue, the parallel fault simulation method under investigation proceeds in three phases each utilizing the shared memory structure in a different way. Firstly, the faults are partitioned into fault sub-lists and each core simulates its own partition of tests independently for its private sub-list. Faults dropped need only to be communicated at the end of this phase, resulting in a single shared memory locking that limits core idle time. After a high percentage of (easy-to-detect) faults is dropped, this mutually exclusive approach is not effective any more. Static partitioning of the remaining hard-to-detect faults results in simulation times that are highly different among cores. This can lead to unbalanced workload which in turn results to underutilization of several cores that may become idle. Thus, a dynamic approach follows in which a small number of faults are dynamically distributed to each core at a time. These faults are simulated in parallel by each core (via a bit-parallel simulation process) and the shared fault list is updated immediately upon simulation. This second phase does not explicitly avoid race conditions as the first one. However, it takes advantage of the different execution times among cores to ensure they access the memory in distinct times; yet its dynamic nature does not result in idle cores, unless all the tests assigned to a core have been examined. A last phase is invoked to utilize the idle cores' processing power which may result to more frequent race conditions (both for faults and tests). To minimize these as much as possible this third phase makes usage of a special data structure called Test and Fault Map (*TFM*) where a record is maintained for each test that contains all the undetected faults not yet simulated for this test. Each record can be accessed separately i.e., without the need of locking the entire map.

4.2 Exploiting Shared-Memory to Steer Scalability of Fault Simulation using Multicore Systems

The proposed parallel fault simulation method takes into account all challenges discussed in Section 4.1.3. This section describes in detail the parallel fault simulation method under investigation and presents optimizations incorporated to provide a scalable and well balanced parallel solution (4.2.2). Necessary notation is given in Table 4.1.

4.2.1 Parallel 3-phase Methodology

The proposed method consists of three phases, each one utilizing the available cores and the shared memory in a different manner (Fig.4.2). Initially, a pre-processing step is invoked to favor assigning at the same core, faults with higher likelihood of being detected by the same test. This step is based on constructing fault sublists by a DFS traversal of the circuit's netlist. The traversal begins at a fault site and groups of faults discovered until a branch is reached. Beginning from the branch a new group is formed following the same approach. When a primary output is reached the current group is also completed and a new traversal begins from a new fault site. This process terminates when each fault is contained in a group. After fault grouping a bin-packing heuristic is followed in order to create equal sized sub-lists to be allocated to cores. Larger groups are put first in bins while smaller groups are used to fill the bins up to size $|F|/m$, where $|F|$ is the total number of faults and m the number of available processing cores.

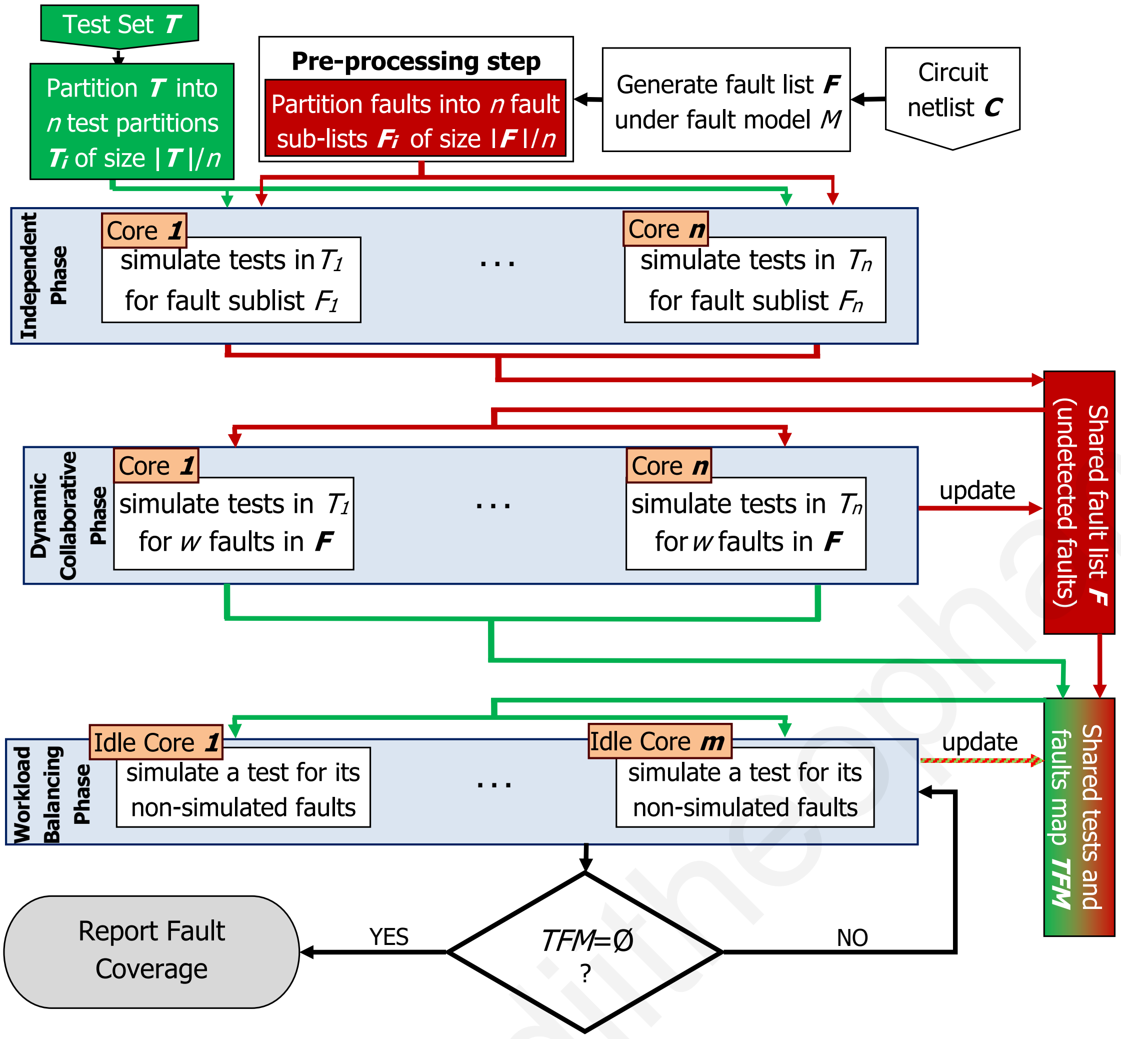

Test Set *T*
Partition *T* into *n* test partitions *T_i* of size |*T*|/*n*
Pre-processing step
Partition faults into *n* fault sub-lists *F_i* of size |*F*|/*n*
Generate fault list *F* under fault model *M*
Circuit netlist *C*
Independent Phase
Core 1
simulate tests in T1 for fault sublist F1
Core n
simulate tests in Tn for fault sublist Fn
Shared fault list F (undetected faults)
Dynamic Collaborative Phase
simulate tests in T1 for w faults in F
simulate tests in Tn for w faults in F
update
Shared tests and faults map TFM
Workload Balancing Phase
Idle Core 1
simulate a test for its non-simulated faults
Idle Core m
update
Report Fault Coverage
YES
TFM=Ø ?
NO

Figure 4.2: Flowchart of the proposed parallel fault simulation method.

This sorted-fault partitioning is used to evenly distribute workload among cores and, at the same time, provides a better starting point for the first phase of the method than a non-sorted approach. The latter has been confirmed by extensive experimentation, the corresponding setup of which has been presented in Section 3.3 and [84] together with relevant results. [84] also describes alternative sorted-fault partitioning approaches such as Breadth-First-Search-based, Reverse Order and Random Selection. The proposed pre-processing approach was selected due to its simplicity and its minimal impact on the total runtime. Keeping its runtime small is highly important for this step as it is the only part of the proposed method that does not run in parallel (its output is an input to problem decomposition). Experimentation has shown that alternative partitioning approaches can affect the overall speedup by 4-5% i.e., reduce it by ~1×.

After the initial pre-processing step, three parallel (Independent, Dynamic Collaborative and Workload balancing) phases take place (Fig.4.2). Initially, at the *Independent Phase*,

Table 4.2: Addressing the identified parallelization challenges in the proposed method

<table>
<tr><th>Parallelization Challenge</th><th>Independent Phase</th><th>Dynamic Collaborative Phase</th><th>Workload Balancing Phase</th></tr>
<tr><td><b>Efficient Fault Dropping</b></td><td>Mutually exclusive fault sub-lists per core. Faults are immediately dropped from local sub-list. The global fault list is updated once at the end of the phase.</td><td colspan="2">Global fault list is updated immediately after detection. Dropped faults are not further considered.</td></tr>
<tr><td><b>Effective Partitioning</b></td><td>DFS-based sorting of faults (pre-processing). Fault expected to be detected by the same test(s) are placed on the same sub-list.</td><td colspan="2">DFS-based sorting of faults (pre-processing). Fault expected to be detected by the same test(s) are simulated together in a bit-parallel fashion.</td></tr>
<tr><td><b>Superfluous Work Avoidance</b></td><td>Faults and Tests are partitioned into mutually exclusive sub-lists. No two cores simulate the same fault even for different tests.</td><td>Faults are marked in the global sublist. No fault is simulated at different cores at the same time.</td><td>Superfluous work avoidance not targeted.</td></tr>
<tr><td><b>Shared Memory Utilization</b></td><td>Sub-lists are kept in core's private cache. Shared memory is only updated at the end of the phase.</td><td colspan="2">A group of faults are simulated together. They are implicitly transfered in the simulating core's private cache.</td></tr>
</table>

tests and faults are equally and statically partitioned among the available cores following a superfluous workload avoidance approach and targeting large number of (mostly easy-to-detect) faults. In this phase cores are working on mutually exclusive search space, as seen in Fig.4.1(d), and are expected to have similar runtime since they are assigned an equal workload. A high percentage of easy-to-detect faults are efficiently dropped per core while, the shared fault list is only updated at the end of the phase avoiding excessive shared memory access for fault dropping, due to the large number of faults being dropped. Despite the obvious benefit of the workload distribution and redundant work avoidance, static partitioning may still result in imbalanced executions among the various cores due to the possibility of different fault dropping ratios among the cores. Hence the proposed methodology goes beyond this static approach, by supplementing it with two dynamic phases that are triggered using specific metrics.

In the *Dynamic Collaborative Phase* that follows, each core simulates the same tests as in the previous phase. The remaining faults are dynamically allocated to all available cores requiring appropriate bookkeeping in order to simulate every test for all undetected faults and also avoid concurrent simulation of the same fault. Faults that are under processing are marked and revisited later exploring the high-speed shared memory for inter-core communication in order to avoid superfluous work. The final *Workload Balancing Phase* redistributes the remaining workload across cores by moving tests from busy cores to others that have already finished processing their initially assigned tests (idle cores). Appropriate bookkeeping using a Test and Fault Map guarantees that a test is simulated for each fault at most once. The following subsections describe in detail each one of these phases while Table 4.2 summarizes how each one of the parallelization challenges presented in Section 4.1.3 is addressed in each phase.

Three phase methodology example

The proposed method highlighted on Fig. 4.2 is presented in the following section with an example. During phase 1, the existence of easy-to-detect faults allows for considerable fault dropping within each individual fault group of $|F|/n$ faults. The pre-processing partitioning step is illustrated in Fig. 4.3(a). T is also partitioned in groups of $|T|/n$ tests, and allocated to the core of the same color which in turn considers only the faults of the same color. The resulting parallelization setup and execution for phase 1 is illustrated in Fig. 4.3(b). The simulation follows a standard event-driven approach based on DFS traversals of C. Each traversal considers sub-groups of faults, within its group ($|F|/n$ faults), in a fault bit-parallel fashion with size equal to the machine word length w. Hence, w faults are sent for simulation at a time. The process is further enhanced by a test bit-parallel technique to calculate the fault-free values of w tests concurrently. The first phase terminates when every core simulates all of its $|T|/n$ tests for all $|F|/n$ faults considered or already detected by a test in its group. This

Algorithm 1 *Independent Phase*

```
Inputs: Test set partition $T_i$, fault list partition $F_i$
Outputs: Updated shared fault list $F$

For all tests $\in T_i$
  $T_w$ = select $w$ tests $\in T_i$
  bit-parallel-simulation($T_w$, fault-free)
$F_{di}$ = $\emptyset$
For each test $t_k \in T_i$
  $F_w$ = select $w$ faults $\in F_i$
  $F_{det}$ = bit-parallel-simulation($t_k$, $F_w$)
  $F_i$ = $F_i \setminus F_w$
  $F_{di}$ = $F_{di} \bigcup F_{det}$
  if ($\frac{|F_{det}|}{|F_i|}$ < DropRate)
    break;
lock-shared($F$)
$F = F \setminus F_{di}$
unlock-shared($F$)
return
```

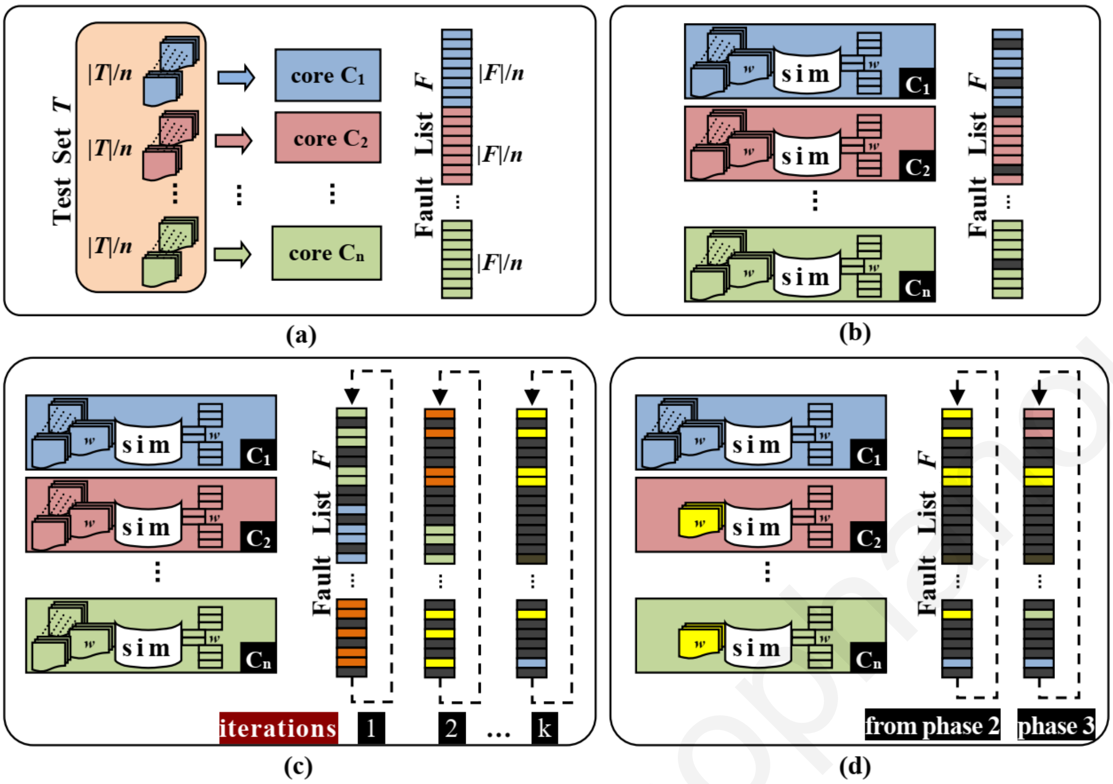


Figure 4.3: Three phase methodology example: (a), Static test partitioning and initial fault list assignment, (b) phase 1, (c) phase 2, and (d) phase 3.

latter case is indicated by the gray entries in the fault list in Fig. 4.3(b). The faults detected at each core are dropped from the global fault list only at the end of phase 1 to avoid unnecessary writes to the shared memory. Since there is no overlap between the faults considered by each core, this write-back scheme is very efficient.

In phase 2 each core can potentially examine all undetected faults not considered by the core during phase 1. Let us denote this list of faults by UF1. Faults in *UF*1 are allocated to each core, in groups of *w* faults, in an iterative dynamic fashion so that no fault is simulated concurrently by multiple cores. During phase 2, global fault dropping is enabled for each simulation step by all cores. Hence, *UF*1′ indicates the dynamically updated list of undetected faults considered by each core during phase 2 (as shown in Fig. 2). Appropriate bookkeeping ensures that no test is simulated twice for the same fault. In each iteration, the process for each core traverses *UF*1′ starting from the first undetected fault not already allocated to some other core, and selects the next *w* faults to simulate. This iterative phase 2 is illustrated in Fig. 4.3(c), where a core considers faults of the same color as itself. For example, during iteration 1, the blue core considers the blue faults in Fig. 4.3(c), which start at the next undetected fault after the blue faults of Fig. 4.3(b). The traversal of *UF*1′ is done in a circular manner. For example, during iteration 1 the green core considers the first group of faults in the list since in phase 1 it considered the last group of faults. This iterative process ensures that a core does not consider a fault it has already considered in a previous iteration or in phase 1. A core process in phase 2 terminates when all allocated tests are simulated for all undetected faults. This phase is sensitive to the uncertainty introduced by fault dropping and may experience imbalanced workload distribution. This case is illustrated in iteration k in Fig. 4.3(c).

A third phase (phase 3) is employed to alleviate this situation and to ensure that the overall workload, among all three phases, is balanced. Specifically, workload from cores still busy in phase 2 is re-distributed to idle cores. In the example of Fig. 4.3(d), only the yellow and blue cores are still busy during phase 2. The red and green cores, which are idle, trigger phase 3 during which workload from the yellow core is redistributed to the two idle cores, along with undetected faults ($UF2$ in Fig. 2) not yet simulated by the yellow core. Redistribution of workload continues until all tests are simulated for all faults or 100% fault coverage is reached. More details for each phase are presented in the following subsections.

(1) Independent Phase

$|T|/m$ tests and $|F|/m$ faults are statically assigned at each core. The first $|T|/m$ tests from T are assigned to Core *1* (T_1), the following $|T|/m$ to Core *2* (T_2) and so on. In the same rationale, $|F|/m$ faults are assigned to each core (F_1 to Core *1*, F_2 to Core *2*, etc). Fig.4.1(d) shows this initial allocation resulting in equal initial workload among cores. Detected faults are stored in local (per core) fault lists (F_{di}), avoiding shared memory contention which can be caused when the global fault list F in the shared memory is updated very frequently by many cores. This is often the case in this phase where a large number of easy-to-detect faults are detected. Hence, each core is independently working on faults from its local fault list (F_i) avoiding communication and time consuming synchronization with other cores. When each core terminates the shared memory is updated by dropping all detected faults from the global fault list F.

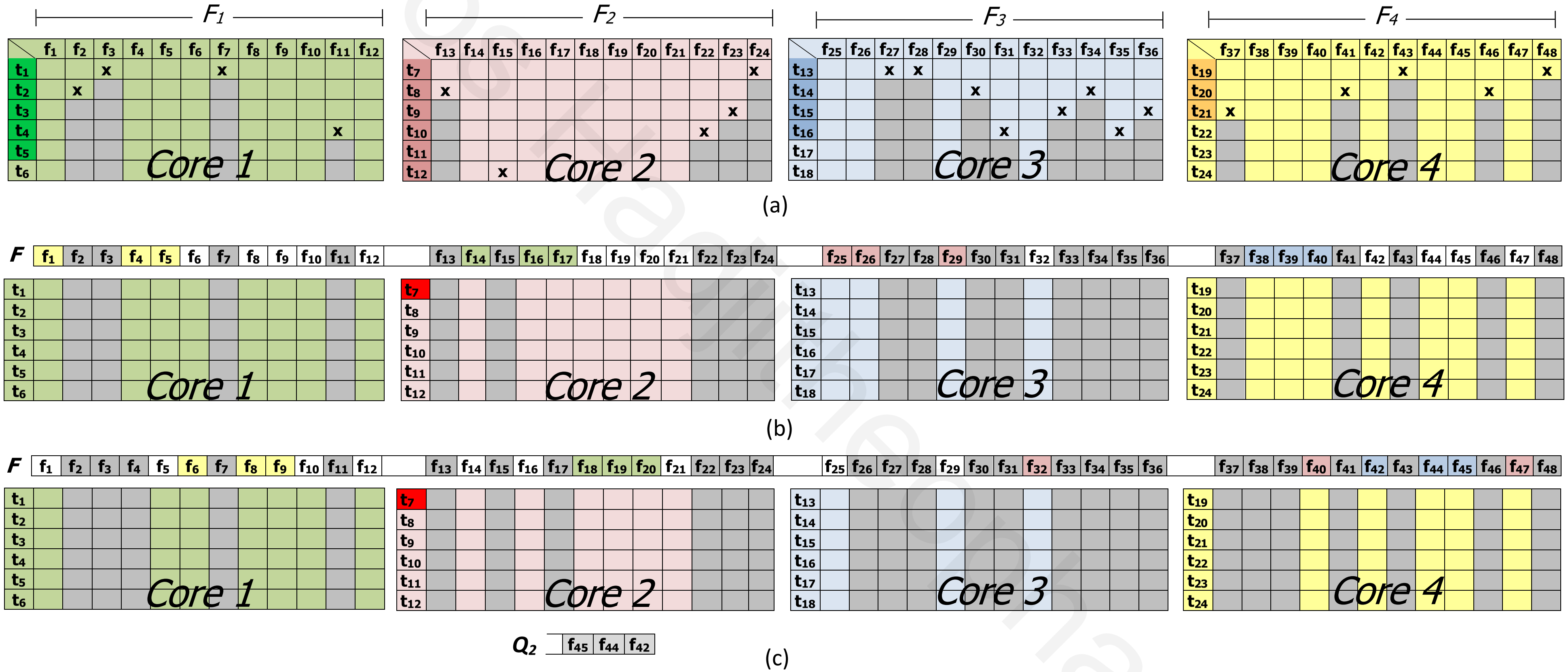


Figure 4.4: Execution example of the proposed methodology. (a) Independent Phase, (b) Dynamic Collaborative Phase - 1st iteration, (c) Dynamic Collaborative Phase - 2nd iteration.

Algorithm 1 shows the basic steps of the proposed approach executed at each core i. The main goal of this Independent Phase is to detect quickly the vast majority of the easy-to-detect faults. For this we introduce the ***fault dropping rate*** *as the lower acceptable bound of fault detections per test*. This rate is monitored and a low value is an indication that the majority of the easy-to-detect faults have been detected, and hence, this phase should be terminated. More details on this can be found in Section 4.2.2.

Initially, all tests allocated to core i are simulated to provide the fault-free responses (lines 01-03). This is done in a bit-parallel manner with w tests simulated at the same time; w is the data word size of the processor. Each one of the w bits is assigned a signal value corresponding to a different test starting from the primary inputs. The simulation traverses the circuit by applying bitwise logic operations at each line indicated by the respective logic gate. Interested readers are refereed to [7] for further details on bit-parallel (fault) simulation. Next, each test in T_i is simulated for all faults assigned to the core (F_i) in groups of w, again in a bit-parallel fashion (lines 05-07). The faults detected from this simulation (F_{det}) are removed from F_i (line 08) and accumulated in the local list of detected faults (F_{di}) (line 09). This iterative approach terminates when any of the termination conditions occurs: (i) all tests have been considered for simulation (line 05), or (ii) the fault dropping rate is lower than a predefined threshold, referred to as DropRate (lines 10-11). Finally, all accumulated faults detected are dropped from further consideration from the shared fault list F (line 13). This updating requires locking of the shared memory to ensure coherency (lines 12 and 14).

Fig.4.4 illustrates the execution of the proposed method using an example, assuming a system with 4 cores (m=4) and with w=3, $|F|$=48 faults, and $|T|$=24 tests. Hence, faults and tests are equally distributed to the available cores, i.e, $|F_i|$=12 faults and $|T_i|$=6 tests as illustrated in Fig.4.4(a). Detected faults are dropped locally per core and only at the end of the Independent Phase are dropped from the global shared fault list F (gray shaded cells of Fig.4.4(b)) to avoid excessive shared memory reads/writes. This write-back scheme is possible since each core has its own search space and, thus, broadcasting detections earlier is meaningless. When the first core finishes processing with its tests or the fault dropping rate for a test is too low (here DropRate=1), the Independent Phase terminates. In this example *Core 2* finishes processing for tests t_7-t_{12} (indicated in green) while *Core 1* has one (t_6), *Core 2* has two (t_{17} and t_{18}) and *Core 4* has three (t_{22}, t_{23}, and t_{24}) tests that are not fully simulated yet. These tests will be considered in the two subsequent phases.

(2) Dynamic Collaborative Phase

During this phase, the same $|T_i|$ tests that are statically allocated to core i during the Independent Phase are considered for simulation. However, faults are distributed per core in a dynamic manner by examining the global shared list of undetected faults F. The main principles used during this phase are (i) each core is allocated w faults from F at a time, and (ii) no two cores simulate the same fault, even for different tests, in order to avoid superfluous work. The latter condition requires global bookkeeping to mark a fault as “under processing” when allocated to a core and not allow for other cores to claim it. This bookkeeping is realized using local queues per core (Q_i). For some test t_j a core goes through F in a circular modulus manner searching for w undetected faults that are not under processing by any other

Algorithm 2 *Dynamic Collaborative Phase*

```
Inputs: Test set partition $T_i$, shared fault list $F$
Outputs: Test and Fault Map $TFM$

01. For each test $t_j \in T_i$
02.   if $t_j$ not examined in previous phase
03.     $TFM(t_j) \leftarrow F$
04.   else
05.     $TFM(t_j) \leftarrow F \setminus F_j$
06. For each test $t_j \in T_i$
07.   For each fault $f_k \in F$
08.     if $f_k$ is under processing
09.       add-queue ($Q_i$, $f_k$)
10.     else
11.       add-list ($F_w$, $f_k$)
12.       mark faults of $F_w$ as under processing in $F$
13.     if ($|F_w| = w$ )
14.       $F_{di}$ = bit-parallel-simulation($t_j$, $F_w$)
15.       lock-shared($F$)
16.       $F = F \setminus F_{di}$
17.       unmark faults in $F_w \setminus F_{di}$ from $F$
18.       unlock-shared($F$)
19.       $F_w = \emptyset$
20.   while ($|Q_i| \geq w$ ) do
21.     $f_k$ = next fault in $Q_i$
22.     if $f_k \in F$ AND not under processing
23.       add-list($F_w$, $f_k$)
24.       remove-queue ($Q_i$, $f_k$)
25.     if ($|F_w| = w$)
26.       repeat steps 13-19
27.   if ($Q_i = 0$)
28.     $T = T - t_j$
29.   else
30.     $TFM(t_j) \leftarrow Q_i$
31.     $Q_i = \emptyset$
32. return
```

core. Undetected faults that are currently under processing by other cores are stored in Q_i in order to be considered later by core i, only if this is necessary. Hence, after a fault in Q_i has been processed by some core j and still remains undetected, then core i will process it. Alternatively, if the fault is detected by some core j then it is dropped from the global fault list F. This procedure continues until all tests in T_i are simulated for all undetected faults in F by repeatedly revisiting Q_i until its size becomes smaller than w. This termination condition has been set to allow core i to continue with its other tests for which many faults remain to be simulated.

Algorithm 2 presents the pseudo-code of the Dynamic Collaborative Phase. The procedure begins by initiating a Test and Fault Map (*TFM*) which serves as input to the following phase. *TFM* maintains a record per test that keeps all the faults that have not been simulated by the test. Each core creates the records for the tests allocated to it (i.e., those in T_i), after the end of the Independent Phase by inserting all faults in F in the test's record *TFM*(t_j) (lines 02-03). If a test has been processed in the Independent Phase, all the simulated faults (i.e., those in F_i) are excluded from *TFM*(t_j) (lines 04-05). Then, each test in T_i is simulated for all faults in F, in groups of w and *TFM* is updated accordingly. Faults are placed in the local list F_w to be simulated and marked as "under processing" (lines 10-12). Faults marked as under processing by other cores are placed in local fault-skip queue Q_i for later consideration (lines 08-09). The outcome of the simulation (stored in F_{di}) (line 14) is used to update the shared fault list F (lines 15-18). This process is repeated until all faults in F have been either simulated or queued. Faults in Q_i are then revisited for simulation (if not already dropped from F by some other core) in lines 20-26. If the fault-skip queue has size smaller than w then its contained faults are not simulated and saved in *TFM* (line 30). If Q_i becomes empty then the test has been simulated for all faults and, thus, can be removed from any further consideration (lines 27-28).

Fig.3(b) and (c) show an execution example for this phase. Let's concentrate on *Core 2* (pink core). T_2 remains the same as in the Independent Phase (Fig.4.4(a)), hence, $T_2 = \{t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$. Starting at the fault immediately following the last fault simulated by *Core 2* in the Independent Phase, i.e., fault f_{25}, *Core 2* looks for w non-detected faults from F which are not under processing. Dropped faults are shown in gray shaded cells in F. These actually no longer exist in F; they are shown here for completeness. Under processing fault cells in F are shown in colors green, pink, blue and yellow, indicating the core that process them, i.e., *Core 1*, *Core 2*, *Core 3* and *Core 4*, respectively. For this example, let w=3. Therefore, during the first iteration of the Dynamic Collaborative Phase, *Core 2* acquires 3 faults and $F_w = \{f_{25}, f_{26}, f_{29}\}$. Each core traverses F in a circular modulus order. For *Core 2* this order is f_{25},f_{26},...,f_{48},f_1,...,f_{12}. For *Core 3*, the order is f_{37},f_{38},...,f_{48},f_1,...,f_{24}.

During the 2nd iteration (shown in Fig.4.4(c)), *Core 2* simulates faults in $F_w = \{f_{32}$, f_{40}, $f_{47}\}$. Faults f_{38} and f_{39} were detected by *Core 3* during its 1st iteration and, hence, are not considered by *Core 2*. Faults f_{42}, f_{44} and f_{45} are under processing by *Core 3* (blue in Fig.4.4(c)) and as a result they are placed in fault-skip queue Q_2. Faults collected in Q_2 are considered after *Core 2* terminates the circular traversal of F. Additional details on the implementation as well as the fault space completeness using this technique is given in Section 4.2.2.

Algorithm 3 *Workload Balancing Phase*

```
Inputs: Shared Fault and Test Map $TFM$,
Shared fault list $F$, Shared test set $T$
Outputs: Fault Coverage

if core $i$ is idle do
  For each not idle core $j$
    if $\exists\ t_k \in T_j \cap TFM$
      remove-list ($T_j$, $t_k$)
      add-list ($T_i$, $t_k$)
      $Fnsim \leftarrow TFM(t_k)$
      add-list ($F_i$, $Fnsim$)
     $F_w$ = Select $w$ faults from $F_i$
     $F_{di}$= bit-parallel-simulation($t_k$, $F_w$)
     lock-shared($F$)
     $F = F \setminus F_{di}$
     unlock-shared($F$)
     $F_w = \emptyset$
     if ($F_i = \emptyset$)
        $T = T - t_k$
return
```

(3) Workload Balancing Phase

In this phase the restriction that each test is allocated to a specific core is relaxed. Targeting full utilization of the available processing power, tests from busy cores are moved to idle cores together with corresponding faults not yet simulated. For this purpose, the *TFM* constructed in a distributive manner by all cores during the previous phase and stored in the shared memory is utilized. Avoidance of superfluous work is not explicitly enforced since in this stage only very few faults remain undetected. The main purpose here is to conclude as fast as possible if these faults are not detected by the given test set or they have (possibly) only one detection that has not been examined yet. Covering all possible combinations will provide the actual fault coverage of the test set under evaluation.

Algorithm 3 summarizes the basic steps of the *Workload Balancing Phase*. The process is activated when the workload assigned to a core in the Dynamic Collaborative Phase has been executed (line 01), e.g. all tests in its test list (T_i) have been retired. The procedure acquires tests from busy cores by searching *TFM* (lines 02-03). The test selected (t_k) is removed from the test list T_j of the busy core (line 04) and added to the test list T_i of the idle core (line 05). Then, all the faults not yet simulated for the acquired test and not dropped by other tests during the previous two phases are allocated to the idle core (lines 06-07). This information is obtained from the globally maintained *TFM* that keeps a record for the faults not yet simulated per test (see Section 4.2.2). The acquired test is simulated for the remaining faults and the shared fault list is updated with the new detections (line 08-12). A test is retired when it has been simulated for all undetected faults (lines 14-15) and the phase terminates when the entire workload is examined (all cores become idle).

4.2.2 Parallelization Optimizations

This subsection provides details on three optimizations incorporated in the methodology presented in Section 4.2 that play important role in achieving high scalability, beyond the generic techniques used to ensure efficient utilization of the shared memory.

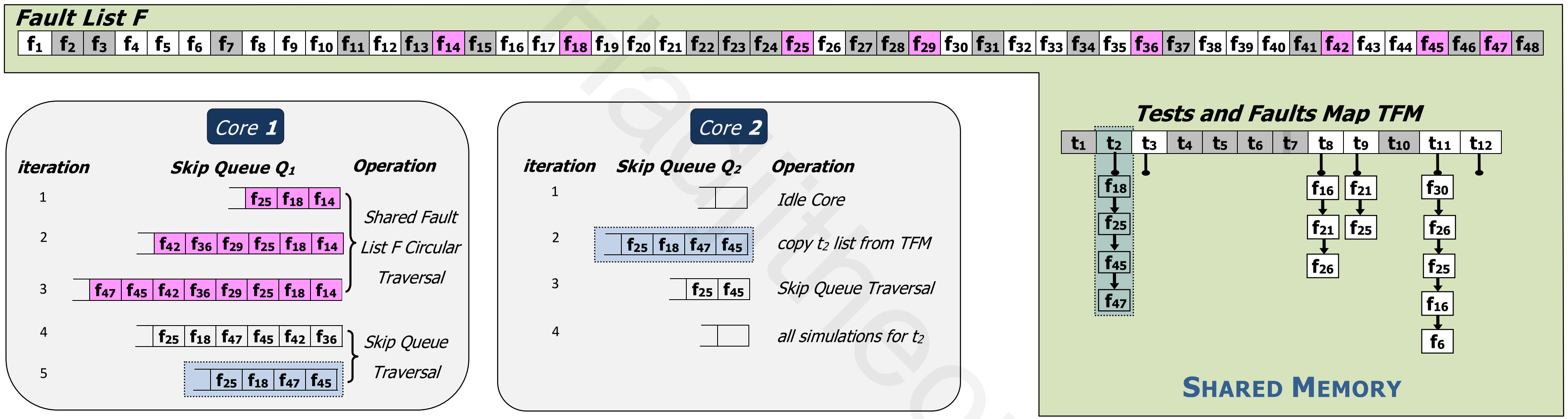


Figure 4.5: Construction of the Tests and Faults Map (*TFM*) example: *Core 1* updates *TFM* with t_2 during the Dynamic Collaborative Phase and *Core 2* utilize *TFM* information during the Workload Balancing Phase

Fault dropping rate monitoring

As discussed in Section 4.2, the main target of the Independent Phase is to cover easy-to-detect faults as early as possible, in order to reduce the overall workload. To achieve this with high utilization of the available processing power, faults and tests are equally distributed to cores in a mutually exclusive manner. As discussed in Section 4.1.3 the difference in the fault dropping rate between cores can lead to idle cores and, hence, to underutilization of the available processing power. The *fault dropping rate* is defined as the number of dropped faults over the number of simulated faults per test, averaged over all tests examined by same core i at a particular point in time. When this ratio drops significantly, this is an indication that the majority of easy-to-detect faults has been detected by some test and that the static distribution of workload followed in the Independent Phase will eventually introduce idle cores. To avoid this, each core monitors the fault dropping rate and when significantly reduced, the Independent Phase is terminated in order to allow for the redistribution of the faults among the cores in a dynamic fashion.

In order to find a suitable value for this rate (threshold) we have examined how the speed-up is affected by the following 4 parameters: $|F|, |T|, |F_i|$ and $|T_i|$. Hence, we run the method using for the threshold weighted values of each one of them varying from $\frac{1}{m}$ up to m, where m is the number of the cores considered. These experiments have shown that the speed-up is affected by the changes of $|F_i|$ in a directly proportional manner and by the changes in $|T_i|$ and $|T|$ in an inversely proportional manner. The changes in the weight of $|F|$ do not affect the obtained speed-up. Next, we combined the values found for the three former parameters and performed a similar exploration. As a result, we have set the fault dropping rate threshold to $|F_i|/(|T_i| \cdot |F| \cdot 2)$, which simplifies to $\frac{1}{2\cdot|T|}$ when $|F_i| = \frac{|F|}{m}$, $|T_i| = \frac{|T|}{m}$. Hence, the threshold for the fault dropping rate is inversely proportional to the test set size.

Fault-skip queues

While in the Independent Phase the mutually exclusive nature of the partitioning guarantees that no superfluous work will be executed, this is not the case for Dynamic Collaborative Phase. Although tests are not shared among cores and, thus, no two cores can simulate exactly the same fault-test combination, superfluous work occurs when two (or more) tests are simulated for the same fault in two (or more) cores in parallel (explained in Section 4.1.3). In order to avoid this problem, faults under processing are marked and not considered by other cores. Any fault found to be under processing during the Dynamic Collaborative Phase is inserted in a fault-skip queue maintained per core (Q_i) in order to be considered after the traversal of the shared fault list F for a test. Faults remaining in the fault-skip queue after this phase terminates are stored along with the corresponding test in *TFM*. Fig.4.5 presents a example of the usage of the fault-skip queues. Let *Core 1* be in the Dynamic Collaborative Phase. *Core 1* simulates t_2 for all the faults not simulated during the Independent Phase, by traversing F. Faults indicated in pink are not simulated by t_2 since they have been marked by some other core(s) as under processing and, hence, placed in Q_1. Specifically, f_{25}, f_{18}, f_{14} are inserted in Q_1 during iteration 1, f_{29}, f_{36}, f_{42} during iteration 2 and f_{45}, f_{47} during iteration 3. In iteration 4, after *Core 1* performed a full traversal of F, t_2 is simulated for faults f_{14}, f_{29} which are removed from Q_1. Faults f_{18}, f_{25} are still under processing and, hence, skipped simulation and inserted back in Q_1. After the end of iteration 5, not enough faults (fewer than w, here 2) can be found to be simulated for t_2. Faults remaining in Q_1 are saved in the shared *TFM* (right part of Fig. 4.5) indicated in blue color. This way *Core 1* can proceed with its other tests for which many faults remain to be simulated. The skipped simulations of t_2 (for faults f_{25}, f_{18}, f_{45}, and f_{47}) will be considered in the Workload Balancing Phase. In Fig. 4.5, *Core 2* enters the Workload Balancing Phase after it becomes idle, acquiring these faults from the record corresponding to test t_2 in *TFM* and simulates them for t_2 without performing any further condition checks.

Shared Tests and Faults Map (*TFM*)

A shared *Tests and Faults Map* (*TFM*) is employed for the appropriate bookkeeping during the simulation evolution to ensure that all test-fault combinations are considered. It consists of lists of faults, one for each test, that keep the faults not yet simulated by the test. For example, in Fig.4.5, t_8 has been simulated for all faults except f_{16}, f_{21}, and f_{26}. *TFM* is initiated after the termination of the Independent Phase by recording all non-retired tests together with their corresponding faults not yet simulated. During the Dynamic Collaborative Phase each list is updated when the corresponding test is considered for simulation. As shown in the previous subsection, this updating is in practice carried out by replacing a test's list in *TFM* with the fault-skip queue formed at the core executing the test's simulations. For the example of t_2, simulated in *Core 1* in Fig.4.5, the fault-skip queue saved in *TFM* is shown in blue. Shaded tests in *TFM* (Fig.4.5) indicate tests for which all the simulations have finished, while tests without a specific list (e.g., t_3) indicate tests which are currently under processing by some core in the Dynamic Collaborative Phase. A core becoming idle because all its tests

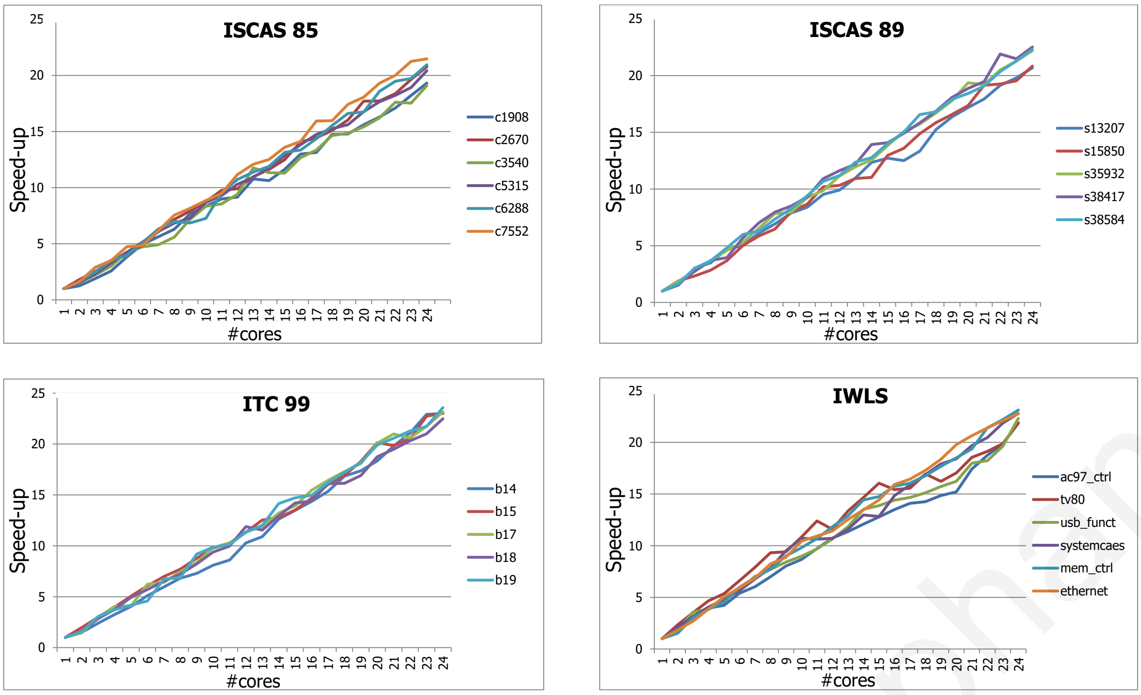


Figure 4.6: Scalability of the proposed fault simulation method using a randomly generated test set.

have been simulated for all faults (*Core 2* in Fig.4.5) enters the Workload Balancing Phase simulating tests that were originally assigned to other cores by considering the corresponding lists from *TFM*. For this example, *Core 2* simulates t_2 (initially assigned to *Core 1*) and in this manner workload is reallocated from *Core 1* to *Core 2*.

4.2.3 Experimental Results

The method was implemented in C++ language with OpenMP parallelization framework and run on a 20-core 2.5GHz Intel Xeon CPU E52670v2 Linux machine with 98GBs of RAM and hyperthreading enabled (2 threads per core, 40 logical cores). The basis for the simulation is an in-house event driven DFS-based fault simulation tool. The larger full-scan version of the circuits in IWLS’05 benchmarks suite under the stuck-at fault model are considered during simulations. Even though stuck-at fault model is used, any linear fault model can be applied to the proposed parallelization method.

Table 4.3 presents the speed-up obtained using 24-logical cores, over a serial fault simulation process (single core run), for a workload of 10000 random tests. 24 logical cores are used instead of the 40 available to allow fair comparison with the work of [1]. For the same reason, the speed-up metric is used as it normalizes the comparison between different approaches that consider similar architectures but with no identical characteristics such as memory size and

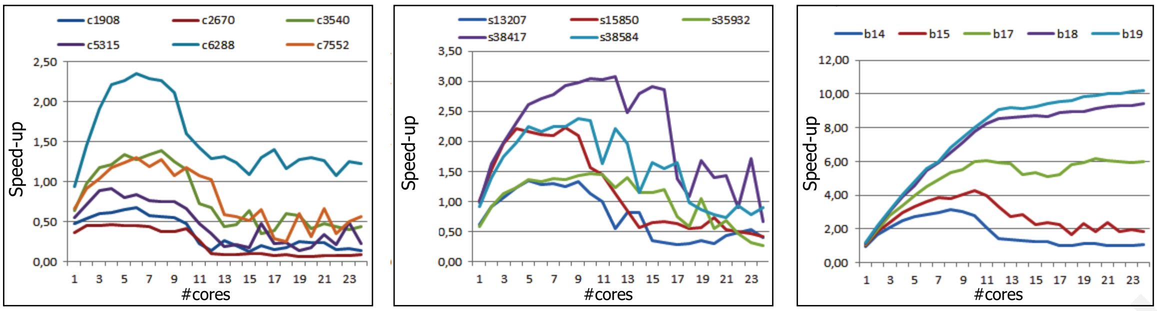


Figure 4.7: Fault simulation scalability as reported in [1] using a randomly generated test set.

Table 4.3: Obtained speed-up and CPU time using 24 cores for 10000 random test patterns.

			Single Dimension Scalability		Scalability (proposed)				
Circuit	**# Nodes**	**# Faults**	**Fault parallelism**	**Test parallelism**	**w/o optim.**	**w/ optim.**	**incr. (%)**	**Scalability [1]**	**CPU time (s)**
c1908	5800	2056	1.14	10.57	16.6	19.32	11.33	1.35	<0.01
c2670	7766	2854	2.14	14.56	16.91	20.78	16.13	2.24	<0.01
c3540	10724	3742	1.01	11.36	16.87	19.1	9.29	3.07	<0.01
c5315	16258	5956	2.36	14.69	17.11	20.42	13.79	2.39	0.01
c6288	12578	6016	3.55	13.3	19.18	20.95	7.38	1.47	0.01
c7552	22686	8080	1.12	13.39	20.4	21.98	6.58	0.67	0.03
s13207.1	13207	10456	1.22	14.59	15.21	20.69	22.83	0.46	0.09
s15850.1	15850	12150	1.61	15.27	17.9	20.84	12.25	1.39	0.06
tv80	24357	21961	1.29	18.03	20.16	21.86	7.08	0.92	1.34
b15	20186	23111	1.93	15.3	22.41	23.04	2.63	2.35	0.11
b14	21680	23512	1.16	14.8	21.71	23.01	5.42	1.3	0.07
systemcaes	30015	26172	2.28	16.69	22.18	22.8	2.58	3.17	1.36
mem_ctrl	37904	27198	2.08	15.59	22.43	23.13	2.92	4.28	9.92
s38417	38417	32151	1.04	18.16	20.34	22.52	9.08	6.18	0.56
s35932	35932	34598	1.23	18.84	20.19	22.2	8.38	9.42	0.67
s38584.1	38584	36759	2.05	17.88	21.38	22.31	3.88	10.2	0.81
ac97_ctrl	39485	38961	2.34	18.11	21.62	21.9	1.17	n/a	3.48
usb_funct	40479	41249	2.75	16.71	19.96	22.32	9.84	n/a	4.76
b17_1	61044	75758	1.97	18.17	22.01	23.16	4.79	n/a	1.72
ethernet	223959	216925	3.04	16.92	21.8	22.76	4.00	n/a	7.21
b18_1	179967	223022	3.39	14.42	21.86	22.49	2.63	n/a	5.68
b19_1	479800	531381	2.65	17.33	23.29	23.58	1.21	n/a	9.53
		Average	**1.97**	**15.67**	**20.1**	**21.87**	**7.38**	**3.18**	

system CPU clock frequency. Column 1 lists the circuit name followed by the number of circuit lines (column 2) and number of stuck-at-faults (column 3) in the collapsed fault list considered for each circuit. Columns 4 and 5 report the speed-up achieved when only one dimension of parallelism is applied; column 4 reports the speed-up achieved when only fault list partitioning is applied (fault parallelism), while column 5 reports the achieved speed-up when only test set partitioning is applied (test parallelism). In fault parallelism the test set is shared among the cores, while faults are evenly distributed to the available cores. This option suffers from considerable overhead due to high idle core times resulted from imbalanced fault dropping among cores and overhead for re-calculation of test fault free responses. In the test parallelism approach the fault list is shared among the cores. While this approach has less overhead it still produces some superfluous work when two (or more) tests are concurrently simulated at different cores for the same fault.

Column 6 reports the speed-up achieved by the proposed method without the optimizations presented in Section 4.2, i.e., the usage of *TFM*, the fault-skip queues and fault dropping rate monitoring are disabled (reported in [28]). Disabling the fault dropping monitoring heuristic extends the *Independent Phase* beyond the point where it effectively identifies easy-to-detect faults. Column 7 reports the speed-up achieved when all optimizations are enabled, which on average offer an additional speed-up of ~2×. This improvement is significant since there is a theoretical upper limit to the possible speed-up, imposed by the number of available processing cores (24 in this case). Hence, the impact of the proposed optimizations is more obvious when the obtained speed-up by the method is significantly lower than 24×. For example, consider circuit s13207.1. The speed-up without the optimizations is 15.21×, much lower than the theoretical possible 24×. By enabling the optimizations, the speed-up is increased to 20.69×, (22.83% improvement). Obviously, proportional improvement is not possible when the speed-up achieved by the method (without the optimizations) is close to 24×, for example in the case of circuit b19_1 where only 1.21% improvement is possible beyond the 23.29× achieved. We report the additional speed-up obtained by the proposed optimizations as a percentage of the maximum theoretical speed-up (24×) for each circuit in column 8. Column 10 lists the overall CPU time of the proposed approach in seconds (including the optimizations heuristics) which is not greater than a few seconds for the larger circuits.

The main observations here are: (i) Test parallelism is more effective in maintaining high speed-up, yet the combination of the two as it is proposed for the *Independent Phase* provides considerably improved speed-ups. (ii) The proposed optimizations are necessary to provide a speed-up boost closer to the optimal (24×) and in no case their inclusion has a negative impact to the speed-up of the methodology (i.e. the overhead is less than the gain). (iii) The obtained speed-up implies that the proposed method will continue to provide scalable speed-ups as the number of cores increases in systems having similar architectures to the one considered here.

Compared to the most recent and related work of [1] reported in column 9 of Table 4.3), the proposed work achieves considerably higher speed-up efficiency on all circuits and especially on larger circuits where the additional workload allows more room for parallelization efficiency. For example, for the larger circuits reported in [1] *systemcaes* and *mem_ctrl* the referred speed-up is 3.17× and 4.28×, whereas in the proposed approach it is 22.8× and

Table 4.4: Obtained speed-up and CPU time using 24-cores for deterministic test sets.

				Workload	Scalability		
Circuit	**# Nodes**	**# Faults**	**# Tests**	**(#test x #faults)**	**Standard order**	**Reverse order**	**Random order**
c1908	5800	2056	112	11924800	6.4	6.58	6.75
c2670	7766	2854	67	22164164	7.42	6.18	7.65
c3540	10724	3742	107	40129208	11.36	10.36	11.06
c5315	16258	5956	67	96832648	11.16	10.72	10.98
c6288	12578	6016	27	75669248	6.32	6.67	5.5
c7552	22686	8080	102	183302880	12.27	11.33	10.05
s13207.1	13207	10456	260	138092392	11.58	11.03	10.75
s15850.1	15850	12150	126	192577500	12.41	11.54	11.85
tv80	24357	21961	555	534904077	18.76	17.79	17.39
b15	20186	23111	467	466518646	20.36	19.7	18.02
b14	21680	23512	757	509740160	20.1	18.35	18.56
systemcaes	30015	26172	152	785552580	18.51	17.89	17.06
mem_ctrl	37904	27198	491	1030912992	20.98	19.59	18.12
s38417	38417	32151	118	1235144967	16.36	15.08	14.58
s35932	35932	34598	21	1243175336	9.13	8.34	8.09
s38584.1	38584	36759	133	1418309256	17.67	17.09	16.51
ac97_ctrl	39485	38961	63	1538375085	14.81	14.33	13.32
usb_funct	40479	41249	122	1669718271	17.78	18.84	18.21
b17_1	61044	75758	1135	4624571352	20.42	20.01	19.09
ethernet	223959	216925	1792	48582306075	20.68	19.4	19.95
b18_1	179967	223022	1087	40136600274	20.13	19.87	18.32
b19_1	479800	531381	3291	254956603800	21.98	21.78	21.63
				Average	**15.30**	**14.66**	**14.25**

23.13×, respectively.

A comprehensive picture of the superiority of the proposed method in terms of scalability is illustrated in Fig. 4.6 and 4.7. Scalability is reported (y-axis) for different number of cores (x-axis). Fig. 4.6 shows the results obtained for 10000 random patterns simulated by the proposed method, while Fig. 4.7 the results for 10000 random patterns as reported in [1]. For all the cases examined, the proposed methodology continues to scale with the same rate for a larger number of cores, where the method of [1] exhibits significant speed-up saturation. This observation implies that the scalability of the proposed method will continue with a similar rate as the number of cores is increased. This is partially supported by the design of the proposed methodology where a simple basic simulation process was used in each core which, in turn, provides high level of freedom during parallelization. This is in a different direction from the relevant work of [1] where the basic process is aggressively optimized limiting the exploitation of parallelization techniques.

Table 4.4 presents the speed-up obtained by the proposed approach (including the opti-

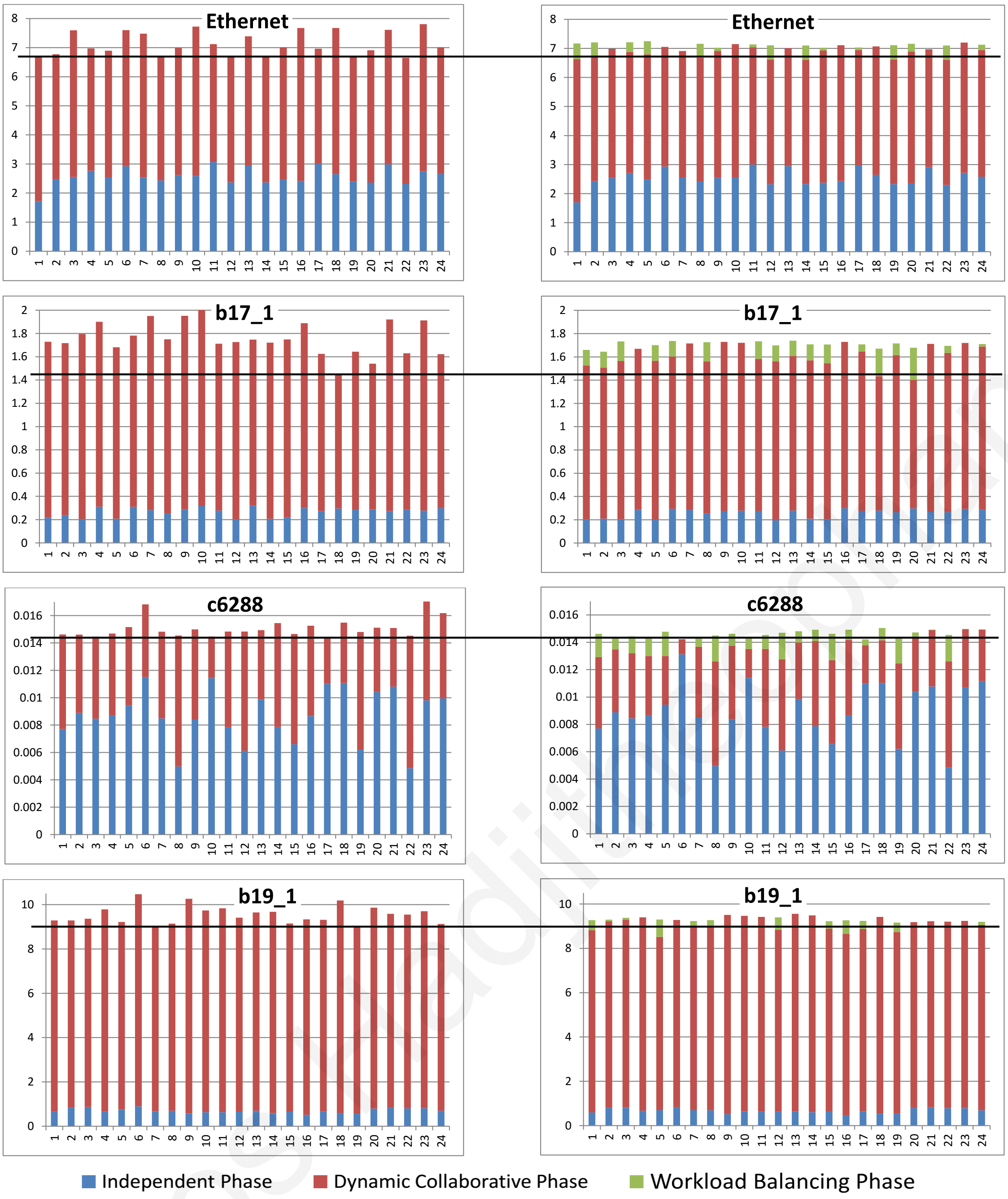


Figure 4.8: Impact of Workload Balancing Phase example. Cores shown in x-axis and CPU-time(s) in y-axis.

mizations), using again 24-logical cores, when a deterministic test set is simulated in different orders. Columns 1-3 list the circuit name, the number of nodes and the number of faults in the circuit, respectively. Column 4 reports the size of the test set considered and column 5 provides an indication of the maximum amount of workload to be carried out by the simulation process, by multiplying the number of faults with the number of tests considered. The latter is by no means an accurate measure, as it only partially takes into account the complexity of the circuit; yet, cases with larger workload (as defined here) provide more solid conclusion for the evaluation of the proposed methodology as they represent more realistic examples. Columns 6-8 show the speed-up achieved following three different orders of the test sets which are then distributed to the cores as described in Section 4.2. In Column 6, the order by which the test generator tool has produced the test set is followed while in column 7 the reverse order is used. In column 8 a random test order is used. By comparing the three techniques we conclude that the test order is of little or no importance to the scalability of the approach. This is in contrast to many existing serial fault simulation approaches where the test simulation order can affect significantly the fault dropping rate and, hence, the overall CPU time. In the proposed parallel approach however, test order does not seem to factor in, mainly due to the proposed fault dropping monitoring and dynamic workload balancing techniques. Observe that higher speed-ups are achieved for larger workloads because these cases take full benefit from the efficient utilization of the processing power.

The final part of our experimentation evaluates the impact of workload balancing in the proposed methodology. As discussed in Sections 4.1.3 and 4.2.2, workload balancing is crucial in achieving high and scalable speed-ups. Fig. 4.8 shows how workload is distributed among the processing cores for four indicative benchmarks. Similar behavior was observed in all the benchmarks considered. The y-axis shows overall CPU time and the x-axis shows the different cores. In the plots on the left side, the Workload Balancing Phase has been disabled, while in the plots on the right side it was enabled. Blue bars show the execution time of the *Independent Phase*, red bars correspond to the *Dynamic Collaborative Phase* and green bars to the *Workload Distribution Phase*.

During the first phase cores are independently working on their private partitions detecting a large percentage of easy-to-detect faults. Very often the Dynamic Collaborative Phase (red bars) is the most time consuming phase since the remaining (not easy-to-detect) faults are systematically targeted in a dynamic manner. Redundant faults play an important role in the CPU time of the entire simulation process since they need to be simulated for all tests. When no Workload Balancing Phase is applied then a large number of cores remain idle while other cores have a large amount of workload left. The horizontal lines in the plots indicate when the first core terminates when no load balancing is applied. For example for circuit b17 core 18 remains idle for 28.5% of the execution time; yet when the Workload Balancing Phase is enabled the maximum idle time is only 0.5% of the execution time. Dynamically redistributing the workload as proposed alleviates this problem and results in smaller overall CPU times for the entire process.

4.3 Chapter Summary

Basic concepts and terminology for fault simulation problem along with the related parallelization attempts have been presented in the chapter. At first, the necessary preliminaries, terminology and basic concepts for fault simulation problem are presented. The important role of fault models, the difference among logic and fault simulation and its important role in test automation process are analyzed.

Despite the high speed-up achieved by some of the works in the literature there is still opportunities for further enhancements regarding the speed-up and for the quality of the results for CMPs architectures. This section summarizes the considerations for parallelization specifically for fault simulation problem and presents in details a new method for on-chip CMPs capable of maintaining its scalability as the number of processing cores utilized increases. The method utilizes a simple, non-optimized single thread simulation process which allows high degrees of freedom to be exploited. The experimental results show that the proposed approach achieves high speed-up rates which, in contrast to comparable state-of the-art methods, increase monotonically with the number of cores demonstrating a highly scalable solution.

Chapter 5

Parallel Test Pattern Generation for Shared-Memory On-chip Multiprocessor Architectures

This chapter outlines basic concepts for test generation problem and proposes a parallel framework for it. Section 5.3 presents in details the parallel test pattern generation methodology under investigation along with extensive experimentation. Results indicate the significance of the proposed parallelization attempt on modern on-chip CMPs architectures.

5.1 Preliminaries - Basic Concepts

Test Generation

Test Generation (TG) is the process of generating vectors called patterns to test a circuit [85]. Typical ATPG algorithms inject a fault into a circuit, *activate* and *justify* a fault, and then, use various mechanisms to *propagate* its effect to an output. The fault is detected when the output signal changes from the expected fault-free value. ATPG algorithms need data structures that are used to describe the search space for test patterns (e.g. binary search trees, binary decision diagrams [93]). ATPG algorithms also need to be complete covering the case where the algorithm must be able to search the entire binary decision tree if necessary in order to generate a test-pattern. [94] analyzed the computational complexity of an ATPG and found that it is an NP-Complete ($O(k^3)$), where k corresponds to the number of circuit nodes). Several combinational ATPG algorithms have been proposed over the years starting with Roth's D-Algorithm (*D-Algorithm*) who defined the basic calculus for many others more sophisticated ones [95]. Later, *PODEM* algorithm introduced the notion of backtrace and used path propagation constraints efficiently with the goal to limit the ATPG algorithms search space [96]. Next, *FAN* algorithm introduced which efficiently constrains the backtrace for further speed-up increase and introduction of headlines for limiting search space [97].

Automatic Test Pattern generation

The task of an ATPG process is to identify/generate an assignment at the inputs of the circuit (i.e. a test pattern), such that the circuit response given a fault f is different from the reference response if the fault f is present. More formally, a test (or test pattern) t for a fault f in a circuit C is the solution to the following equation: $C(t) \neq Cf(t)$, where $C(t)$ represents the logic function of the fault-free circuit and $Cf(t)$ the logic function of the faulty one. An ATPG flow aims at generating a test for each fault in the fault list and combines them to a test set. In addition to the detection of the fault, such test sets often have secondary objectives and properties. For instance, technical requirements and limitations of the test infrastructure of the target circuit have to be considered. Additionally, the number of test patterns in a test set highly influences the test application time (i.e. the time an Automatic Test Equipment (ATE) needs, to apply the test set to an individual circuit), which directly influences test economics. Depending on the outcome of the ATPG, a fault f is classified as follows. If a test t for a fault f exists, the fault is called testable and additionally called detected if t is also part of the final test set. Whereas if the ATPG process could show that no such test exists f is said to be redundant or untestable. ATPG is a well-known NP-hard problem and becomes more demanding as devices under test are becoming larger, more complicated due to the new emerging defects that require new fault models of higher complexity.

5.2 Motivation and Considerations for Parallelization

In on-chip multi-core architectures with shared memory, on-chip communication is not a limitation any more and the cost of inter-core communication is reduced significantly. Furthermore, the tends on current on-chip multi-core architectures is that the level of memory coherency is guaranteed and the number of available cores keeps increasing. These new developments and trends motivate towards the investigation of parallel ATPG approaches capable of achieving speed-up scalability as the number of on-chip cores increases, while overcoming new challenges such as shared memory contention, as well as efficient workload distribution parallel threads.

A common parallelization procedure consists of three steps: (i) decomposition (domain or functional), (ii) parallel execution, and (iii) final result assembly. Step (ii) can result in significant compromise of the quality of the obtained results and, at the same time, not offer the expected speed-up. An efficient parallel algorithm should effectively overcome challenges such as memory contention and imbalanced workload distribution. Any parallel ATPG method has to appropriately designs all three steps to ensure that these challenges are treated efficiently.

5.2.1 Test inflation problem

Proper problem decomposition, workload distribution and final test set recomposition are essential to guarantee the quality of the results while maintaining fault coverage. Since typically each core does not consider the entire search space, parallel approaches tend to choose local optimal solutions resulting in test set increase [2, 40], known as the test inflation problem. An analysis presented in [98], states that an increase (15.9%) in test patterns can cause an 100% increase in test cost per unit at the worst-case scenario without including the additional test time and the financial loss due to delayed time to market. Hence, test inflation is an important problem for a parallel ATPG solution which can limit the practicality of the proposed solutions.

5.3 Utilizing Shared Memory Multi-cores to Speed-up the ATPG process

In [26] it is presented a parallel ATPG methodology for shared-memory systems geared towards high speed-up and test inflation containment. The methodology takes advantage of fast and low cost shared memory communication inherent in the underlying architecture in order to coordinate the main steps of the ATPG to avoid redundant work and dynamically allocate the workload while minimizing memory contention caused by multiple cores (threads) when accessing shared data. A test generation flow is proposed in which hard-to-detect faults are targeted first, followed by a parallel fault simulation-based merging process to maximize fault coverage. This process employs a series of newly proposed parallelization heuristics to explicitly avoid simultaneous consideration of the same faults by two or more cores, in order to minimize extra work and thread idle time. Any remaining undetected faults are targeted during a following phase, in a similar manner. The obtained experimental results demonstrate the effectiveness of the proposed approach in speeding-up the ATPG process and provide comparisons with relevant recent work.

5.3.1 High-Level Parallel ATPG Framework

The proposed ATPG method appropriately designs all three steps presented in 5.2 to ensure that these challenges are treated efficiently. Specifically, two conceptual approaches are adopted: (i) problem partitioning to avoid executing the same work concurrently in different cores and (ii) fine-grained granularity of each step to provide dynamic distribution of work. Various parallel optimization heuristics based on these concepts are discussed in Section 5.3.2. The test generation flow of the proposed ATPG Framework is based on the rationale of these two concepts.

The proposed methodology relies on an initial test-per-fault step, for a limited number of faults, to obtain an initial seed test set over which the algorithm evolves. This, combined with the many degrees of freedom allowed in a test seed by our single fault ATPG process, provides the desired granularity that allows mutually exclusive distribution of work in the different cores. Such distribution benefits the exploration of different parallelization directions, including dynamic partitioning and adaptive decision making for test merging. However, an approach with high granularity may perform large amount of unnecessary work when not taking advantage of fault dropping. Fault dropping plays a critical role in test generation anyway, as it can significantly affect test set size. In parallel test generation, inefficient dropping of faults can also restrict speed-up, regardless from the fact that the main process for identifying faults to be dropped (fault simulation) can be implemented very efficiently in parallel environments [21,24]. A fair trade-off between high granularity and fault dropping consideration is to develop a methodology based on distinct test epochs, one targeting hard-to-detect faults and a following one targeting the remaining undetected faults.

Fig.5.1 presents the high level description of the proposed methodology. Firstly, the circuit netlist is analyzed to obtain a collapsed fault list $\boldsymbol{F}$ for the underlying fault model $\boldsymbol{M}$. Consequently, the fault list is sorted in a depth-first-search (DFS) order (based on their location in the netlist) in an attempt to implicitly group faults with structural similarities in $\boldsymbol{F}$. This fault locality property of the input fault list benefits fault dropping after $\boldsymbol{F}$ is partitioned to the available cores. The next step identifies hard-to-detect faults to be targeted by the first test epoch (Epoch I) of the methodology. We use random test pattern generation, which is a simple, quick and acceptable way to classify faults; however, other more sophisticated methods can be incorporated. Hard-to-detect faults are identified using a multiple detection

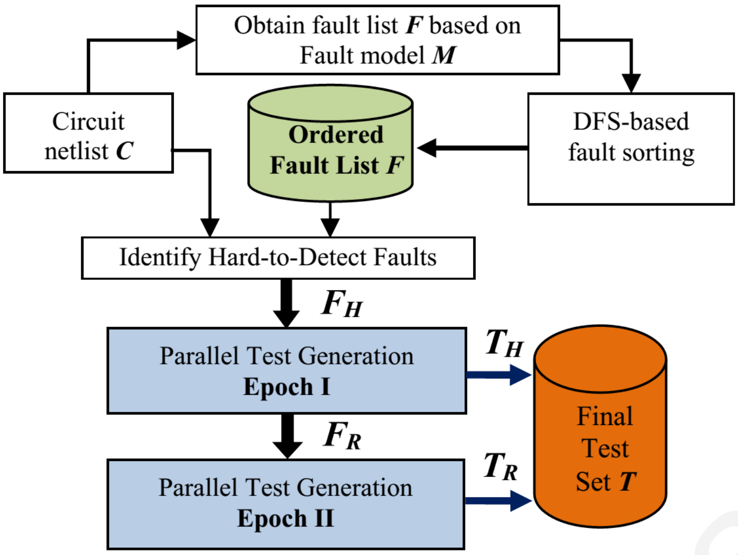

Obtain fault list $\boldsymbol{F}$ based on Fault model $\boldsymbol{M}$
Circuit netlist $\boldsymbol{C}$
Ordered Fault List $\boldsymbol{F}$
DFS-based fault sorting
Identify Hard-to-Detect Faults
$\boldsymbol{F_H}$
Parallel Test Generation Epoch I
$\boldsymbol{T_H}$
$\boldsymbol{F_R}$
Parallel Test Generation Epoch II
$\boldsymbol{T_R}$
Final Test Set $\boldsymbol{T}$

Figure 5.1: High level flow of the main Test Generation (TG) processes.

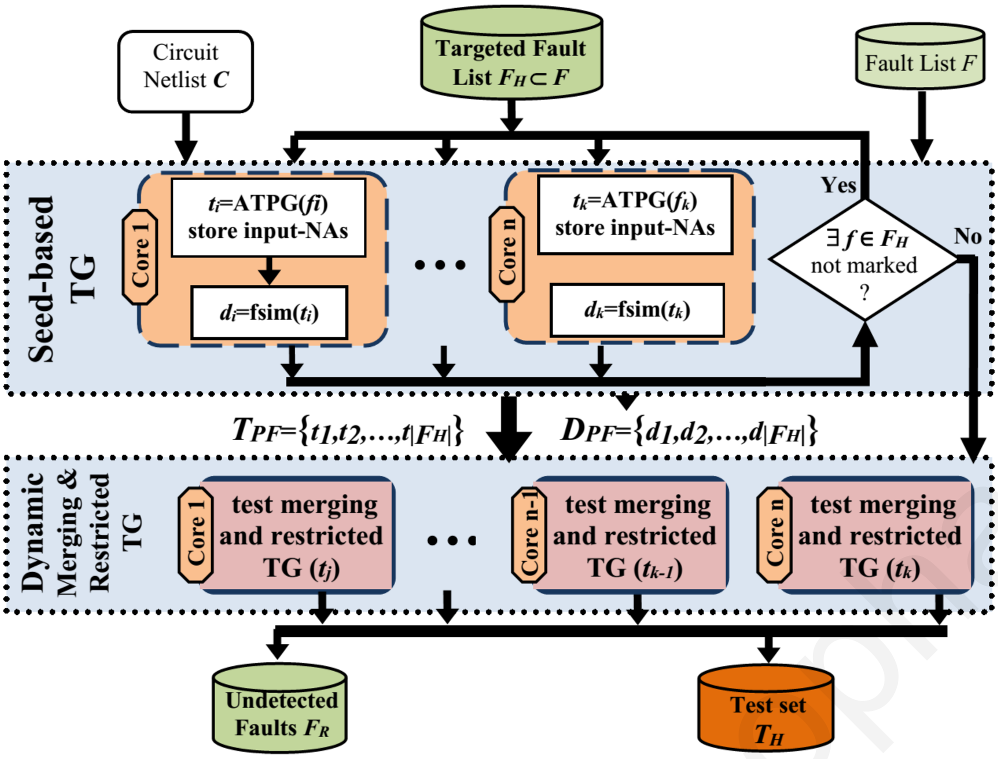
approach where 10% (set by experimental exploration) of the faults in $\boldsymbol{F}$ with the fewer detections are considered as hard and used as the input fault list of test Epoch I ($\boldsymbol{F}_H$). Epoch I performs explicit test generation for each fault in $\boldsymbol{F}_H$ while also considering faults in $\boldsymbol{F}-\boldsymbol{F}_H$ during fault simulation to identify faults detected coincidentally ($\boldsymbol{F}_C$). A merging process to reduce the number of tests obtained follows, and the final output is a set of test patterns $\boldsymbol{T}_H$ detecting all faults in $\boldsymbol{F}_H \cup \boldsymbol{F}_C$. A second test epoch (Epoch II), similar to the first one, is invoked to target the remaining faults, i.e. $\boldsymbol{F}_R = \boldsymbol{F} - (\boldsymbol{F}_H \cup \boldsymbol{F}_C)$ producing a set of tests $\boldsymbol{T}_R$ such that $\boldsymbol{T} = \boldsymbol{T}_H \cup \boldsymbol{T}_R$ detects all faults in $\boldsymbol{F}$.

5.3.2 Parallelization Methodology And Optimizations

Section 5.3.2 describes the major steps undertaken during a test epoch, discussing dynamic fault partitioning and core synchronization, while this Section 5.3.2 describes a number of optimizations proposed to overcome parallelization issues.

Test-Epoch Parallelization

Fig. 5.2 presents a flowchart illustrating the basic steps of the parallel Test Generation (TG) methodology followed during a test epoch, namely *seed-based TG* and *dynamic test merging and restricted TG*. An epoch explicitly targets on a fault-by-fault basis, only a small subset of the fault list $\boldsymbol{F}$ ($\boldsymbol{F}_H$ for Epoch I and $\boldsymbol{F}_R$for Epoch II). Note that $\boldsymbol{F}_C$= $\boldsymbol{F} - (\boldsymbol{F}_H \cup \boldsymbol{F}_R)$ typically constitutes the overwhelming majority of the faults which are easily detectable in an implicit manner (i.e., via fault simulation). The faults in a fault list are sorted based on structural



Circuit Netlist C
Targeted Fault List $F_H \subset F$
Fault List F
Seed-based TG
Core 1
t_i=ATPG(f_i) store input-NAs
d_i=fsim(t_i)
• • •
Core n
t_k=ATPG(f_k) store input-NAs
d_k=fsim(t_k)
Yes
$\exists f \in F_H$ not marked ?
No
$T_{PF}=\{t1,t2,\ldots,t|F_H|\}$
$D_{PF}=\{d1,d2,\ldots,d|F_H|\}$
Dynamic Merging & Restricted TG
Core 1
test merging and restricted TG (t_j)
• • •
Core n-1
test merging and restricted TG (t_{k-1})
Core n
test merging and restricted TG (t_k)
Undetected Faults F_R
Test set T_H

Figure 5.2: Test Epoch flowchart: A test epoch targeting hard-to-detect faults (Epoch I). Same steps are repeated in Epoch II, with input fault list F_R and resulting test set T_R.

similarities of fault locations (netlist), in order to increase the probability of proximate faults to be detected by the same test.

During the first step (seed-based TG in Fig. 5.2), each available core performs test seed generation (TG with maximal don't care bits) for the next undetected fault $\boldsymbol{f}_i$ in the list using a PODEM-based process optimized to identify tests with a large number of unspecified bits. The order of the selection of the next fault(s) is not important here, as the partitioning is designed to work in an independent manner and produce standalone results. The system shared memory holds the updated fault list (indicating faults not yet targeted) and, therefore, duplication of work is avoided as each core works on a distinct fault. For each test seed $\boldsymbol{t}_i$ generated during this step parallel fault simulation is performed and all faults detected (including those in $\boldsymbol{F}$ -$\boldsymbol{F}_H$) are stored in a list $\boldsymbol{d}_i$. Faults in $\boldsymbol{d}_i$ are not immediately dropped as this information is used during the following step. Also, the input *necessary assignments* (NA) of $\boldsymbol{t}_i$ are stored, along with $\boldsymbol{d}_i$, to be exploited in the next step. This first step terminates when all faults in $\boldsymbol{F}_H$ have been targeted, constituting a synchronization barrier in the process. $\boldsymbol{T}_{PF}$ contains the test seeds and $\boldsymbol{D}_{PF}$ contains the corresponding fault simulation results which are both kept in the shared memory.

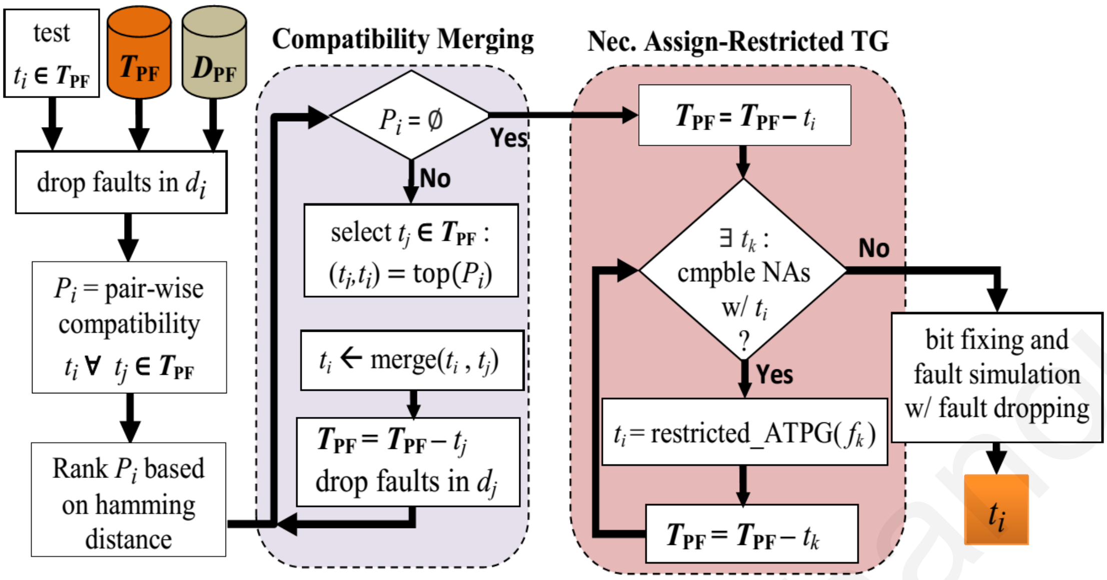

test
$t_i \in T_{PF}$
T_{PF}
D_{PF}
drop faults in d_i
P_i = pair-wise compatibility $t_i \,\forall\, t_j \in T_{PF}$
Rank P_i based on hamming distance
Compatibility Merging
$P_i = \emptyset$
Yes
No
select $t_j \in T_{PF}$: (t_j, t_i) = top(P_i)
$t_i \leftarrow$ merge(t_i, t_j)
$T_{PF} = T_{PF} - t_j$
drop faults in d_j
Nec. Assign-Restricted TG
$T_{PF} = T_{PF} - t_i$
$\exists\, t_k$: cmpble NAs w/ t_i ?
No
Yes
t_i= restricted_ATPG(f_k)
$T_{PF} = T_{PF} - t_k$
bit fixing and fault simulation w/ fault dropping
t_i

Figure 5.3: Merging process flowchart: dynamic test merging and restricted TG processes per core.

Upon completion of the first step, the next step is invoked (dynamic test merging & restricted TG in Fig. 5.2) in order to merge compatible tests and reduce the size of $\boldsymbol{T}_{PF}$. Each core selects its primary test target $\boldsymbol{t}_i$ from $\boldsymbol{T}_{PF}$ to be the test seed with the larger detection list d_i(test with the highest number of coincidental detections) and immediately marks it so that other cores cannot select it. Tests are selected in an iterative manner until no further merging is possible. A detailed description of this selection is given in Section 5.3.2 under *detection-based test selection*. This merging step is dynamic *due to the efficient communication of the merged tests through the shared memory*. Thus, in each iteration, the number of candidate tests for merging is reduced at a fast rate.

Fig. 5.3 shows the merging process undertaken by each core while the shared memory accommodates information about faults detected and tests discarded. The faults detected by the primary test $\boldsymbol{t}_i$ (kept in $\boldsymbol{d}_i$) are immediate dropped from further consideration. Then, pair-wise compatibilities of the primary test $\boldsymbol{t}_i$ with each remaining test $\boldsymbol{t}_j$ of $\boldsymbol{T}_{PF}$ are calculated and ranked based on increasing Hamming distance. The test pair ($\boldsymbol{t}_i$, $\boldsymbol{t}_j$) with the smallest Hamming distance is thereafter selected to be merged. Upon merging, $\boldsymbol{t}_i$ is updated and $\boldsymbol{t}_j$ is discarded from $\boldsymbol{T}_{PF.}$Also, all faults in the corresponding list $\boldsymbol{d}_j$ are dropped from the shared fault list. When no further merging is possible, restricted TG for $\boldsymbol{t}_i$ is performed based on the necessary assignments (NAs) on primary inputs collected during the first step. NAs are used as hard constraints for test generation, yet only for faults corresponding to tests not already marked and having identical NAs as $\boldsymbol{t}_i$**.** This iterative step terminates when no more tests with identical NAs exist that could lead to further test discarding. As a final step, the remaining unspecified bits of $\boldsymbol{t}_i$ are assigned specified values and the test is fault simulated to identify any further coincidental detection of faults. All the tests obtained by the process of Fig. 5.3 are appended at the output of the corresponding test epoch, i.e., $\boldsymbol{T}_H$ for Epoch I and $\boldsymbol{T}_R$ for Epoch II.

Optimizations

Detection-Based Primary Test Selection

In the merging step of Dynamic Test merging and Restricted TG process 5.3, test selection is very important for the efficient evolution of merging since it sets the constraints and outcomes of consequent merging iterations, and fault simulation. Practice in ATPG suggests that early fault dropping plays a more important role than having fewer constraints (more unspecified bits) in the test seed. For this reason, the *primary test* $\boldsymbol{t}_i$ during dynamic merging is selected based on its number of detected faults in $\boldsymbol{d}_i$. Recall that the fault simulation process performed at the end of the first step of the test epoch (Fig.5.1) does not drop faults; instead, it is used for providing a more precise metric for this selection during the second step. Tests to be merged (with the primary test) are then selected based on their Hamming distance to the primary test. The Hamming distance based merging produces merged tests with a smaller number of specified bits and, hence, fewer constraints in the following iterations of the merging process. In the (often common) case where more than one tests have the same Hamming distance to the primary test, their fault detection metric is used to decide which test will be merged. This optimization greatly assists in *dynamic workload balancing* and *minimization of unnecessary work* since high amount of early fault dropping reduces the faults for which explicit test generation is needed.

Balanced Workload Distribution

Distribution of workload to the available cores can significantly impact the speed-up of a parallel methodology. Test generation and fault simulation processes have unpredictable execution times due to the nature of the problems and fault dropping. Core idle time is minimized by dynamically selecting: (i) the next fault to be targeted in seed-based test generation in each epoch (Fig.5.1), (ii) the next test to be used as primary in test merging (Fig.5.1), and (iii) the tests to be merged with the primary test seed (Fig. 5.3). Since, data is stored in shared memory (fault list and test seeds), and thus, is easily accessible by all cores, *provides a punctual way of determining how the workload will be selected at each step and by each optimization mechanism of the approach.*

Scalable Parallel Fault Simulation

Fault simulation is used in many cases in the proposed methodology and, thus, its performance significantly affects the overall performance. Specifically, fault simulation is used two times is each Epoch:

(i) To find the number of detected faults per test seed without fault dropping. This information is given as input to the merging procedure in order to avoid repeated simulations after each merging.

(ii) At the end of the merging step in order to detect as many coincidental faults as possible and, hence, minimize the test set size.

In case (i) the fault simulation is performed after test seeds have been generated for all faults. Since generation in the various cores is executed independently (only for the faults assigned), the cores finishes this step in different times, resulting in idle cores. These cores can be utilized for fault simulation in a parallel fashion. The challenge here is that the number of idle cores is changing (increasing) as more cores finish and, hence, the simulation should utilize them as well. To take full advantage of this situation (no core remain idle) we have fully incorporated for this case the highly scalable parallel fault simulation of [28]. This fault simulator has been shown to provide linear speedup as the number of cores increases and can be dynamically adjusted to the number of available cores. In case (ii) the fault simulation should proceed within one core since, after the merging step, the final test should be simulated to identify co-incidental fault detections (see Fig. 5.1).Recall, that in this step the test seeds are dynamically acquired by cores from the shared fault list and merged with other seed until no more merging is possible. Hence, fault simulation cannot run in parallel to get maximum benefit. Nevertheless, the fault simulations exploits bit-parallel simulation principles presented in [28] where many faults (equal to the machine word size w) are simulated with a single circuit traversal.This results in a considerable speedup of this step by a factor very close to w.

Test Set Private Consideration

The search for the best candidate tests to be merged (either the primary or the ones to follow) involves high interaction of each core with the shared memory. Specifically, selecting the primary test, as well as computing the pair-wise compatibilities with the remaining tests in $\boldsymbol{T}_{PF}$,inherently involves memory contention since all cores are searching $\boldsymbol{T}_{PF}$.This issue is addressed by dynamically partitioning $\boldsymbol{T}_{PF}$ in $\boldsymbol{m}$ private subsets ($\boldsymbol{m}$ being the number of available cores), one for each core. Each core can only select tests from its own private subset of $\boldsymbol{T}_{PF}$ (and the corresponding $\boldsymbol{D}_{PF}$) which can be safely moved to its own private cache. This *implicitly minimizes concurrent memory access requests from different cores* that can result in inefficient memory utilization due to memory contention. Moreover, it implicitly *minimizes duplication of work* as each core considers a distinct part in $\boldsymbol{T}_{PF}$. When a core finishes with the merging process within its private part of $\boldsymbol{T}_{PF}$, it is allowed to work on the entire set in order to *ensure workload balancing by avoiding idle periods in cores*. At this point, concurrent memory accesses can occur, however, their impact is minimal as the bulk of the merging process has already occurred during the private consideration, and, hence, the size of $\boldsymbol{T}_{PF}$ is by this point significantly reduced.

Test Provisional Marking

During compatibility merging, the list $\boldsymbol{P}_i$ which holds pair-wise compatibilities between tests, requires updating after each merging. This updating is highly demanding in processing resources as it is of cubic complexity in the worst case. To avoid this issue the proposed methodology calculates and ranks compatibilities only once for each test $\boldsymbol{t}_i$. If a test $\boldsymbol{t}_j$ is selected to be merged with $\boldsymbol{t}_i$, it is provisionally marked in $\boldsymbol{T}_{PF}$ so that it is not merged in another core, *explicitly avoiding imposing unnecessary constraints in another thread that performs merging*. If compatibility between $\boldsymbol{t}_i$ and a test $\boldsymbol{t}_j$ in $\boldsymbol{P}_i$ is invalidated by a previous merging, merging between $\boldsymbol{t}_i$ and $\boldsymbol{t}_j$ is not completed and the provisional marking is cleared. Otherwise, provisional marking indicates permanent discarding of $\boldsymbol{t}_j$ from $\boldsymbol{T}_{PF}$.

5.3.3 Experimental Results

The proposed method implemented using C++ language and run on a 20 cores Intel Xeon CPU E5-2670v2 with 98GBs of RAM, running Linux. OpenMP parallel programming framework was used for parallelization. We present results for the larger full-scan versions of the circuits in the IWLS'05 benchmarks suite. The method can be applied to any linear fault model; here we present results for the stuck-at fault model.

Table 5.1: Speed-up and Test set Increase Results for the proposed method using 8, 12, 16, 20, 30 and 40 cores.

				Serial		8 cores		12 cores		16 cores		20 cores		30 cores		40 cores	
Circuit	**# PIs**	**# Nodes**	**# Faults**	**Aborted**	**|T|**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**	**|T| incr. (%)**	**Speed-up (x)**
c1355	41	1355	1410	0	84	0	5.61	1.19	7.43	1.19	9.21	1.19	10.38	1.79	14.91	0.0	20.30
c1908	33	1908	2056	0	111	7.21	4.18	2.70	5.85	6.31	5.02	2.70	7.00	2.99	9.67	1.80	12.09
c2670	233	2670	2954	0	66	1.52	9.65	-3.03	13.17	-1.52	15.27	1.52	18.40	2.27	22.95	3.03	28.12
c3540	50	3540	3742	0	103	1.94	8.24	9.71	8.87	2.91	10.56	2.91	11.74	2.89	14.78	2.91	22.54
c5315	178	5315	6016	0	66	6.06	8.85	9.09	11.35	12.12	14.79	15.15	15.02	18.55	18.29	21.21	22.54
c6288	32	6288	7744	0	26	7.69	6.58	15.39	7.87	11.54	9.32	11.54	11.57	11.54	18.26	11.54	24.90
c7552	207	7552	8080	0	101	4.95	6.48	2.97	9.52	5.94	12.25	5.94	15.14	11.54	18.26	9.90	28.64
s9234.1	247	9234	6781	0	141	7.09	4.91	14.18	7.48	12.06	9.22	15.60	11.67	17.44	15.18	18.44	18.56
s13207	700	13207	10456	0	255	4.31	5.92	6.67	9.01	7.06	10.72	7.45	13.54	8.43	17.78	9.02	21.09
s15850.1	611	15850	1215	0	125	8.00	4.97	10.40	8.29	12.8	9.77	14.40	11.44	17.09	18.22	18.40	23.70
s38417	1664	38417	32320	0	117	1.71	7.61	4.27	10.38	7.69	14.09	9.40	16.50	12.69	20.61	15.38	24.06
s38584.1	1464	38584	38358	0	131	3.05	6.93	6.11	10.07	7.63	13.99	9.16	16.97	12.01	21.35	13.74	26.32
s35932	1763	35932	39094	0	20	5.00	6.22	15.00	9.37	20.00	11.00	25.00	11.69	25	12.78	25.00	14.42
b14	277	21680	23716	0	751	1.33	7.34	1.20	11.20	0.80	14.71	1.07	17.50	0.91	24.14	0.67	30.71
b15	485	20186	23498	192	461	0	7.86	0.87	11.40	0.87	14.93	1.30	19.20	2.23	23.89	2.17	28.64
b17	1449	61044	75498	0	826	0.36	7.32	0.85	11.33	1.57	15.04	2.78	18.54	4.72	24.04	4.96	30.63
b18	3307	179967	223352	8	1030	0.78	7.52	1.46	10.57	2.04	13.92	2.33	15.34	3.99	22.53	4.17	29.22
b19	6666	479800	534144	991	3245	0.80	7.16	6.01	10.05	5.76	12.18	6.29	14.03	6.7	19.69	6.72	24.69
b20	522	31258	34528	0	519	2.70	7.78	3.28	11.75	2.31	15.04	5.97	17.91	7.64	25.39	8.67	31.99
b21	522	31157	34331	0	561	2.67	7.66	1.78	11.45	3.03	14.76	1.43	19.48	1.94	25.2	1.78	31.23
b22	735	39385	48812	0	561	-1.07	7.58	1.426	11.59	0.36	14.34	3.21	17.99	3.22	25.92	4.99	34.56
ac97_ctrl	2283	39485	39226	0	62	3.23	5.98	6.45	7.67	11.29	10.26	12.90	11.98	17.39	16.64	19.35	21.03
ucb_funct	1874	40479	42214	0	113	7.96	6.18	13.27	9.18	14.16	11.12	15.04	14.46	18.06	18.98	18.58	22.98
tv80	373	24357	24810	0	554	0.61	8.08	0.61	11.71	0.20	15.19	1.01	17.06	0.82	22.31	1.44	26.77
systemcaes	930	30015	29256	0	143	0.70	6.77	6.29	9.06	5.59	12.43	8.39	13.99	10.17	18.41	10.49	22.73
mem_ctrl	1198	37904	39882	0	485	1.86	6.91	3.09	10.52	0.82	13.91	2.89	17.25	4.18	24.87	6.80	32.40
ethernet	10640	223959	221628	5	1421	1.27	7.05	1.97	9.70	2.18	12.50	3.03	14.33	4.33	22.01	4.64	28.91
					Average	**3.03**	**6.94**	**5.30**	**9.85**	**5.80**	**12.43**	**7.02**	**14.82**	**8.43**	**19.96**	**9.10**	**25.16**
			Average Memory Increase (x)			**2.11**		**2.59**		**3.02**		**3.27**		**3.59**		**3.77**	

Table 5.1 presents the obtained speedup and test set sizes (as increase %) of the proposed parallel ATPG method compared to a serial version of the algorithm. The speed-up measure allows for the evaluation of the scalability of the approach under different execution set-ups, as well as for a fair comparison with other works considering the same architecture but with different characteristics such as CPU clock and total memory. Results from experimental set-ups with 8, 12, 16, 20, 30 and 40 cores are reported. After the circuit name and the number of inputs (Col. 1-2), the size of the circuit and number of faults in the collapsed fault list (Col. 3-4) are presented. Col. 5-6 report the number of aborted faults (indicating the achieved fault coverage) and the test set size obtained by a serial version of the proposed methodology, respectively. The number of aborted faults in the multi-core execution set-ups is always smaller than the one reported in Col. 5 (hence, fault coverage is at least as high) and is not reported here due to space limitations. Col. 7, 9, 11, 13, 15 and 17 list the test set size increase as a percentage of the one obtained by the serial execution and Col. 8, 10, 12, 14, 16 and 18 report the speed-up achieved, when 8, 12, 16, 20, 30 and 40 cores are employed.

The obtained results demonstrate almost linear speed-up increase while at the same time the test set size increase is very limited for most of the circuits. In the worst case test set increase is no more than ∼15% whereas in the average case it is only 7.02%, for 20 cores. Circuit s35932 is an exception (with 25% increase) due to the very small test set size of the serial version with 20 tests, which becomes 25 tests for 20 cores. The proposed method also exhibits small memory increase, an objective often targeted by parallel methods. The last row of Table I summarizes the average memory increase factor among all circuits. For the 8-cores run, the required memory is only 2.11× more than the serial version, while the 16 cores runs increase the memory by 3.02× on the average. These numbers indicate that the memory increase does not grow proportionally with the number of cores; rather, it grows with a decreasing rate as the number of cores increase. This is attributed to the dynamic manner in which the proposed algorithm performs fault dropping, alleviating the following steps from unnecessary calculations.

The proposed methodology is compared with the most relevant and recent parallel approaches considering the shared-memory multi-core architecture model, such as [2–5]. Where available, results are compared directly for the common benchmarks. Moreover, results on additional benchmarks for each technique are listed and average trends for each methodology are analyzed. For the proposed methodology, the larger (in terms of # Nodes) circuits (b18, b19 and ethernet) are listed, on top of the common ones. The number of execution cores in each case was determined by the results reported in these works. Col. 2-7 of Table 5.2

Table 5.2: Speed-up, Test Set Size and Memory Increase Comparison with the works in [4], [5] and [3].

	Comparison with [4] - 12 cores						Comparison with [5] and [3] - 16 cores								
	Speed-up (x)		Test set increase (%)		MemoryIncrease (x)		Speed-up(x)			Test set increase(%)			MemoryIncrease (x)		
Circuit	**[10]**	**prop.**	**[10]**	**prop.**	**[10]**	**prop.**	**[11]**	**[13]**	**prop.**	**[11]**	**[13]**	**prop.**	**[11]**	**[13]**	**prop.**
D1	8.20	-	-1.50	-	-	-	-	-	-	-	-	-	-	-	-
D2	7.70	-	19.00	-	-	-	-	-	-	-	-	-	-	-	-
D3	7.50	-	16.00	-	-	-	-	-	-	-	-	-	-	-	-
D4	7.30	-	1.40	-	-	-	-	-	-	-	-	-	-	-	-
D5	-	-	-	-	-	-	7.38	9.99	-	10.64	0.41	-	4.33	3.02	-
D6	-	-	-	-	-	-	9.37	10.00	-	2.54	12.17	-	4.89	3.22	-
D7	-	-	-	-	-	-	8.88	9.28	-	1.37	2.14	-	3.94	2.78	-
s38417	7.50	9.99	40.00	4.27	-	2.18	-	-	14.09	-	-	7.69	-	-	2.98
s38584.1	7.30	9.89	12.00	6.11	-	2.03	-	-	13.99	-	-	7.63	-	-	2.35
s35932	7.40	8.64	10.00	15.00	-	2.75	-	-	9.34	-	-	20.00	-	-	3.15
b15	7.80	11.40	18.00	0.87	-	2.92	-	-	14.93	-	-	0.87	-	-	3.01
b17	8.30	11.33	5.00	0.85	-	3.09	-	-	15.04	-	-	1.57	-	-	3.27
b18	-	10.57	-	1.46	-	3.80	-	-	13.92	-	-	2.04	-	-	4.09
b19	-	10.05	-	6.01	-	3.11	-	-	12.18	-	-	5.76	-	-	3.78
ethernet	-	9.70	-	1.97	-	3.59	-	-	12.50	-	-	2.18	-	-	4.19
Average	**7.67**	**10.20**	**13.32**	**4.58**	**-**	**2.93**	**8.54**	**9.76**	**13.25**	**4.85**	**4.91**	**5.97**	**4.39**	**3.01**	**3.35**

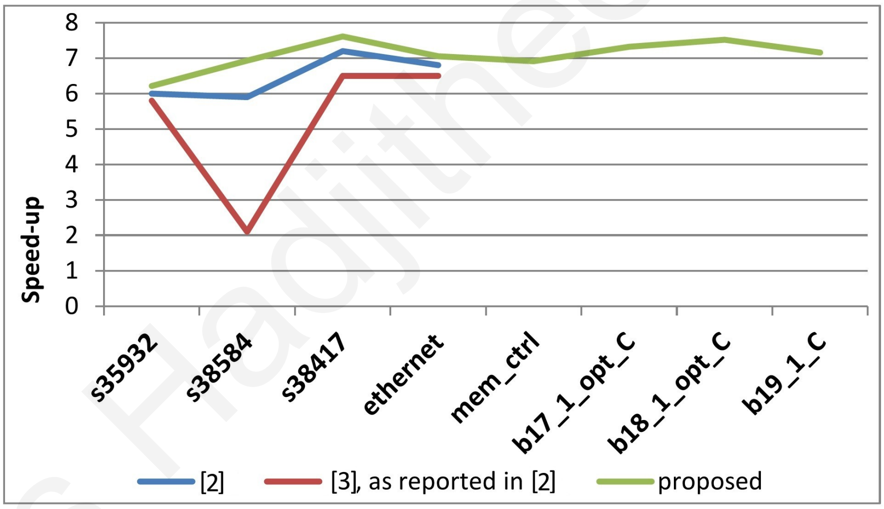


Figure 5.4: Test generation method: speed-up comparison with [2] and [3] for an 8-core set-up.

provides a comparison with [4] for the set-up of 12 cores. Similarly, Col. 8-16 of Table Table 5.2 compare the proposed methodology with the approaches of [5] and [3], for the 16-cores setup reported in these works. A "–" indicates not available results for the corresponding work. Average trends are reported in the last row of Table 5.2.

An 8-core system is considered in [2] and compared with an implementation of [3]. Fig. 5.4 compares the speed-up of the proposed methodology with that of [2] and [3], as reported in [2]. The proposed methodology achieves on average ~7x speed-up, outperforming both existing methods for the 4 common circuits. This is achieved mainly due to the optimizations (discussed in Section 5.3.2), which minimize redundant work by immediate updating of the fault status. Comparison regarding the test set size reduction is not possible with [2] as absolute values for test set sizes as well as corresponding fault coverage are not reported.

5.4 Chapter Summary

This chapter presents the basic concepts for ATPG problem along various optimizations for parallelization. A parallel test pattern methodology for shared memory multi-core environments in detail described along with a number of newly proposed heuristics that attempt to avoid assigning the same portion of the workload to multiple cores, while the distribution of work in the available resources targets to minimize core idle time. Experimental results demonstrate high speed-up rates that keep increasing as the number of the available cores increases. At the same time, test set size increase is limited and comparable to other state-of-the-art parallel approaches.

Chapter 6

Parallel *n*-Detect Test Pattern Generation on Shared-memory Multi-core Architectures

This chapter presents an extension of the method highlighted in Chapter 5 exploring a parallel multiple-detect (*n*-detect) test pattern generation problem. It has been shown that emerging defects and various faults can be detected by a set of test vectors that obtain high stuck-at fault coverage, particularly multiple detect (*n*-detect stuck-at fault) test vectors. This chapter investigates a parallel *n*-detect methodology that utilizes on-chip CMPs for extending an efficient parallel test generation for a more computational intensive problem. Results maintain all the good properties of single detect test generation (scalability and test inflation) in a more beneficial extent.

6.1 *n*-detect Parallelization Methodology

The method (illustrated with a flow chart in Fig.6.1) follows a similar pathway like the process presented in Section 5.3. Fig.6.1 presents the high level description of the *n*-detect methodology. The framework with the two Epochs is used in order to detect and drop the majority of the (easy to detect) faults and focus the available resources on the hard-to-detect faults. The basic steps of the the test Epoch are illustrated in Fig.6.1 named *seed-based TG* and *dynamic test merging*. An epoch explicitly targets on a fault-by-fault basis, only a small subset of the fault list $\boldsymbol{F}$ ($\boldsymbol{F}_H$ for Epoch I and $\boldsymbol{F}_R$for Epoch II). Note that $\boldsymbol{F}_C$= $\boldsymbol{F}$ – ($\boldsymbol{F}_H \cup \boldsymbol{F}_R$) typically constitutes the overwhelming majority of the faults which are easily detectable in

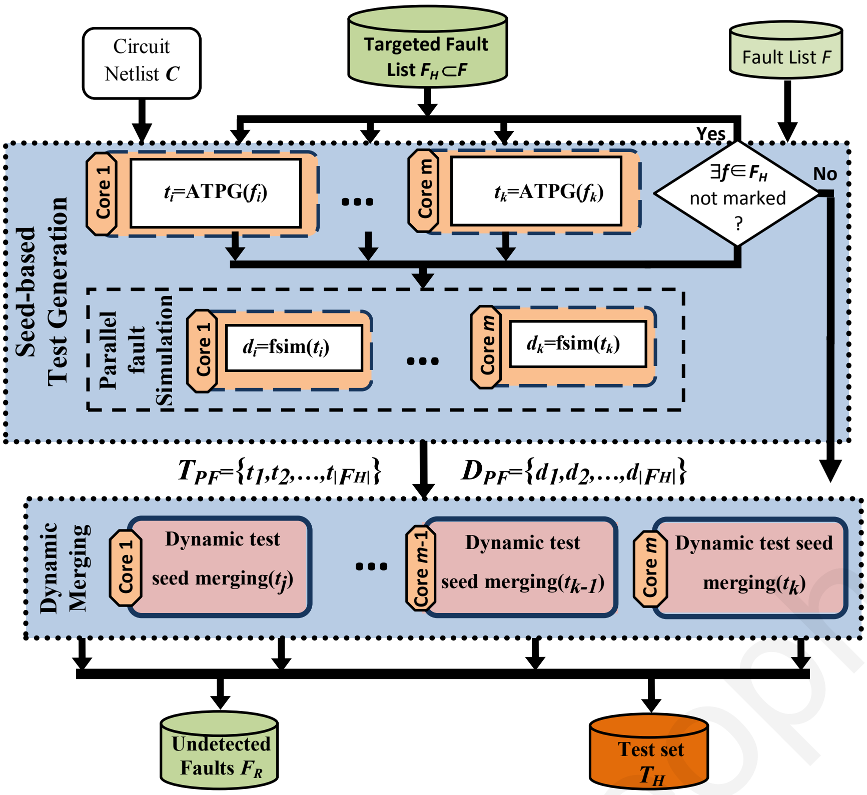

Circuit Netlist C
Targeted Fault List $F_H \subset F$
Fault List F
Yes
No
$\exists f \in F_H$ not marked ?
Seed-based Test Generation
Core 1
t_i=ATPG(f_i)
Core m
t_k=ATPG(f_k)
Parallel fault Simulation
Core 1
d_i=fsim(t_i)
Core m
d_k=fsim(t_k)
$T_{PF}=\{t_1,t_2,\ldots,t_{|F_H|}\}$
$D_{PF}=\{d_1,d_2,\ldots,d_{|F_H|}\}$
Dynamic Merging
Core 1
Dynamic test seed merging(t_j)
Core m-1
Dynamic test seed merging(t_{k-1})
Core m
Dynamic test seed merging(t_k)
Undetected Faults F_R
Test set T_H

Figure 6.1: High level flowchart: A test epoch targeting hard-to-detect faults (Epoch I). Same steps are repeated in Epoch II, with input fault list F_R and resulting test set T_R.

an implicit manner (i.e., via fault simulation). The faults in a fault list are sorted based on structural similarities of fault locations (netlist), in order to increase the probability of proximate faults to be detected by the same test. More details for the framework are presented in Section 5.3.1.

Algorithm 4 outlines the merging process undertaken by each core while the shared memory accommodates information about faults detected and test seeds discarded. The input to this merging process is the test seed generated in the previous step (T_{PF}) and their corresponding faults detected (D_{PF}) as well as the fault list F_H. This process is similar for the 2 Epochs of the methodology hence, without loss of generality, here we describe the method for Epoch I. Each core considers seeds in a specific range of the test seed set T_{PF} in order avoid utilizing the same seed for merging in more than one cores. Core k (out of the m available) considers only $\frac{|T_{PF}|}{m}$ seeds for primary selection in the range $k \times \frac{|T_{PF}|}{m}, \ldots, (k+1) \times \frac{|T_{PF}|}{m} - 1$, denoted here as $T_{PF}(k : k+1)$. Once a seed t_i in this range is selected (line 02) as a primary seed to be merged all the detected faults are immediately dropped from the globally maintained fault list F_H (line 03) and the seed is removed from the given seed test T_{PF} (line 04). Then, the hamming distance between each of the remaining seeds in the considered range (i.e., $T_{PF}(k : k+1)$) and t_i is calculated and saved in list P_i (lines 06-07). Observe that, in subsequent iterations, this range changes only for the secondary seeds (line 05) so that all seeds in T_{PF} are considered to be merged with the primary t_i. Then, the seed with the minimum hamming distance is selected (line 08) and merged (line 11) with t_i. The merged seed is removed from T_{PF} (line 12) and its detected faults are dropped from further consideration (line 13). When no more seeds in the range can be merged with t_i, the algorithm continues to the next range of $\frac{|T_{PF}|}{m}$ seeds (lines 09-10). Lines 14 to 24 are invoked when all the secondary seeds are considered and, hence no more merging is possible, in order to identified detections of faults not in F_H (i.e., in $F - F_H$). Lines 14 to 24 are skipped in Epoch II since all the remaining faults are placed in the given sublist F_R. First, a bit still with unspecified value in t_i is randomly selected (line 14), fixed to 0 (line 15) and simulated for faults not in F_H (line 16). In lines 17 and 18 bit fixing and fault simulation is repeated for value 1. Based on which bit fixing detects more faults, t_i is updated accordingly (lines 19-20 for 0, lines 22-23 for 1) and a list of faults F_C accumulates all the coincidentally detected faults (line 21 for 0 and 24 for 1). All these faults

Algorithm 4 *Dynamic Merging for Core k*

```
Inputs: Test seeds $T_{PF}$, faults detected per seed $D_{PF}$, shared fault list $F$,
fault sublist $F_H$
Output: Test set $T^k{}_{PF}$
01. $T^k{}_{PF}$ = $\emptyset$
02. For each seed $t_i \in T_{PF}$(k:k+1) : $d_i$ = max{$D_{PF}$(k:k+1)}
03.   $F_H$ = $F_H \backslash d_i$
04.   $T_{PF}$ = $T_{PF}$ - $t_i$
05.   For h = k,k+1,…,m-1,0,1,…,k-1
06.     For each $t_j \neq t_i \in T_{PF}$(h:h+1)
07.         $P_i \leftarrow$ hamming($t_i$,$t_j$)
08.     For each $t_j \neq t_i \in T_{PF}$(h:h+1) : ($t_i$, $t_j$)=min{$P_i$}
09.       If (min{$P_i$} = $\infty$)
10.         break;
11.       $t_i \leftarrow$ merge($t_i$, $t_j$)
12.       $T_{PF}$ = $T_{PF}$- $t_j$
13.       $F_H$ = $F_H \backslash d_j$
14.   For each unspecified bit $b_z$of $t_i$
15.     $t_i^0$ = $t_i$ : $b_z$ = 0
16.     $d_i{}^0$ = fault_sim($t_i^0$,$F$-$F_H$)
17.     $t_i^1$ = $t_i$ : $b_z$ = 1
18.     $d_i{}^1$ = fault_sim($t_i^1$,$F$-$F_H$)
19.     If ($d_i{}^0 \neq \emptyset$ AND |$d_i{}^0$| $\geq$ |$d_i{}^1$|)
20.       $t_i$ = $t_i{}^0$
21.       $F_C$ = $F_C \cup d_i{}^0$
22.     If ($d_i{}^1 \neq \emptyset$ AND |$d_i{}^0$| < |$d_i{}^1$|)
23.       $t_i$ = $t_i{}^1$
24.       $F_C$ = $F_C \cup d_i{}^1$
25.   $T^k{}_{PF}$ = $T^k{}_{PF}$+$t_i$
26. $F$ = $F \backslash F_C$
```

will be dropped from the global fault list F at the end of this process (line 26). Finally, the obtained test t_i is inserted in a test set for core k (line 25) that contains tests to be placed in the output test set of the Epoch i.e., $T_H = \bigcup_{k=0:m-1} T^k_{PF}$.

Dynamic merging based on pair-wise hamming distance

t_i	T_{PF}	d_i
t_1	XX1X001XXX	6
t_2	1X0XXX1X0X	2
t_3	**XXXX0X1X0X**	**11**
t_4	00XX0X1XX0	10
t_5	1XX11X010X	3
t_6	0X1X01XXXX	9
t_7	11XXXXX10X	4

➡

1st iteration		2nd iteration		
P_i	T_{PF}	P_i	T_{PF}	T^K_{PF}
	-		-	-
Hamming Distance	1X0XXX1X0X	Hamming Distance	~~1X0XXX1X0X~~	~~1X0XXX1X0X~~
t_1 t_2 t_3 t_4 t_5 t_6 t_7 3 3 - 4 ∞ 5 5	merge(t_3, t_1) = XX1X001X0X	t_1 t_2 **t_3** t_4 t_5 t_6 t_7 - ∞ - 6 ∞ ∞ 7	merge(**t_3**, t_4) = 001X001X00	**001X001X00**
	00XX0X1XX0		-	-
	~~1XX11X010X~~		~~1XX11X010X~~	~~1XX11X010X~~
	0X1X01XXXX		~~0X1X01XXXX~~	~~0X1X01XXXX~~
	11XXXXX10X		~~11XXXXX10X~~	~~11XXXXX10X~~

Bit-wise hamming distance calculation and merging rules

i^{th} bit of		hamming	i^{th} bit of
t_i	t_j	distance	merge(t_i,t_j)
0	0	+0	0
0	1	+∞	conflict
0	X	+1	0
1	0	+∞	conflict
1	1	+0	1

i^{th} bit of		hamming	i^{th} bit of
t_i	t_j	distance	merge(t_i,t_j)
1	X	+1	1
X	0	+1	0
X	1	+1	1
X	X	+0	X

Figure 6.2: n-detect dynamic merging execution example.

Fig. 6.2 presents the dynamic test merging procedure with an example. First t_3 is selected since it detects the most faults (11) among the other seeds in T_{PF} (left top table in Fig. 6.2). Next, the hamming distances between t_3 and all other seeds in T_{PF} are calculated as the sum of the bit-wise distances per bit pair indicated in Column 3 of the bottom tables. For example, the hamming distance between t_3 and t_5 is $1+0+0+1+\infty+0+\infty+1+0+0=\infty$ indicated that no merging between them is possible. The hamming distances between t_3 and all other seeds in T_{PF} are saved in P_i (shown under P_i column of 1^{st} iteration in top table). These values indicate that the best seed to be merged with t_3 is either t_1 or t_2 with the former selected. Merging of t_3 and t_1 is shown in the T_{PF} column under 1^{st} iteration (changed bits are shown in red) and is realized following the rules listed in Column 4 of bottom tables. During the 2^{nd} iteration the hamming distances are recalculated in P_i. Observe that t_2 and t_6 are now incompatible with t_3 after its merging with t_1. Seed t_4 is selected to be merged with the current seed. The resulting seed has no compatibility with the remaining seeds and hence, no further merging is possible. During Epoch I, bit fixing together with fault simulation follows the process shown here to detect coincidental faults. When all unspecified bits have been fixed and fault simulated, the test is advanced to the output test set T^k_{PF} and all the corresponding faults detected are dropped from the global fault list.

The main challenge of this extension is to ensure the n-detect property. The challenge applies both to seed generation and seed merging processes. Test seed generation should guarantee the generation of n different seeds per fault in the given fault list (F_H in Epoch I and F_R in Epoch II). Furthermore, in order to ensure high quality of the resulting test set, test seeds should have significant difference since this was shown to detect more defects [99]. In addition, the merging process should be constrained in order to guarantee that the seeds generated for the same fault are not merged in the same final test, and, hence, reduce the number of detections for that fault. In Subsection 6.1.1 we propose a method that generates n different tests for the same fault with significance difference can be generated with little impact on the speedup. Subsection 6.1.2 describes a technique for partitioning test seeds to be merged that ensures the n-detect property and maintains speedup increase as the number of cores increases. All steps of the 2-epochs methodology highlighted in Section 5.3.2 not explicitly described here remain exactly the same.

6.1.1 Multiple Test Seed Generation

The proposed seed generation procedure, extends the PODEM-based tests generation algorithm mentioned in III.A to produce multiple (n) test seeds per considered fault. The extension rolls back in the decision tree of the algorithm altering taken decisions for the activation of the fault and its propagation to an observable point. After their generation the multiple seeds for each fault are verified to be or enforce to become incompatible. In summary the procedure for each considered fault consists of three steps:

i. Generate one test seed using a PODEM-based process (as in single detect).

ii. Roll back on the decision tree, altering decisions to produce more (different) seeds.

iii. Discard any seed compatible with other seeds.

iv. If necessary fix unspecified bits to obtain n different test seeds.

Hence, following the test generation of the first test seed for a fault (same as in the methodology presented in Fig. 6.1) in step (i), the process looks back at the various decisions made during this initial seed generation (step (ii)). A decision here refers to alternative circuit path segments that the algorithm selects in order to generate the seeds. For example, in the circuit of Fig.6.3 test generation takes place for fault f_x sa0. In order to enforce value 1 to f_x (activate fault)we need at least a 1 at any of the OR gate inputs (i.e, a, b, c). When generating the first seed, the algorithm assigns value 1 at input c. Then, it proceeds to justify this value

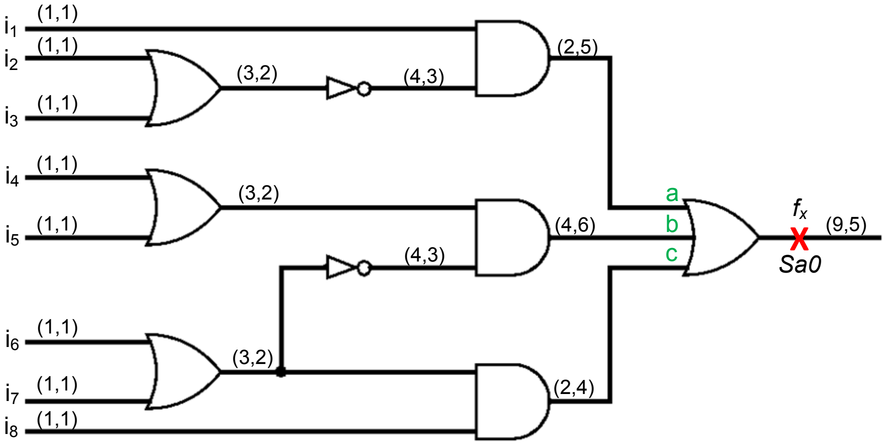


Figure 6.3: Decision changes for multiple test seed generation circuit example.

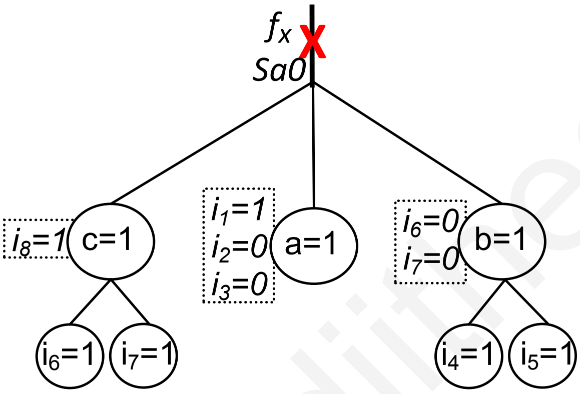


Figure 6.4: Fault activation decision tree for f_x Sa0 in Fig.6.3.

with a backward traversal on the circuit. For the second seed, the algorithm takes the alternative decision of assigning 1 at line *a* removing the initial constraint of assigning 1 to *c*. The justification of this new decision can produce a different test seed than the first one. Similar decisions take place for the propagation of the fault effect to a primary output. The closer the decision to be altered is to the fault site, the higher the difference will be between the seeds. In step (ii) a decision tree is progressively constructed to record all the taken decisions and their alternatives. Fig. 6.4 shows the decision tree for the activation of the sa0 fault at line f_x. The different paths of the tree indicate the different possible combinations of decisions that exist.

In step (iii) each generated seed is *checked for compatibility* with previously generated seeds (compatible seeds have no conflicting bits). For each pair of compatible seeds only one is kept, that with the fewer number of specified bits. Discarding compatible seeds ensures the *n*-detect property of the final test set throughout the following merging process of the

	steps (i), (ii) and (iii)			step (iv)					
seed #	decisions taken	implications	t_i	# Sp bits	t_i'	# Sp bits	t_i'	# Sp bits	t_i'
1	c=1, i_6=1	i_8=1	XXXXX1X1	2	XXXXX1**0**1	3	XXX**1**X1**0**1	4	XXX**1**X1**0**1
~~2~~	a=1	i_1=1, i_2=0, i_3=0	~~100XXXXX~~	-	-	-	-	-	
2	b=1, i_4=1	i_6=0, i_7=0	XXX1X00X	3	XXX1X00X	3	XXX1X00X	3	XXX1X00**0**
~~3~~	c=1, i_7=1	i_8=1	~~XXXXXX11~~	-	-	-	-	-	
~~3~~	b=1, i_5=1	i_6=0, i_7=0	~~XXXX100X~~	-	-	-	-	-	
3	none	none	-	-	XXXXX1**1**1	3	XXX**1**X1**1**1	4	XXX**1**X1**1**1
4	none	none	-	-	-	-	XXX0X1**0**1	4	XXX0X1**0**1
5	none	none	-	-	-	-	-	-	XXX1X00**1**

Figure 6.5: *n*-detect test seed generation example. *n=5*, number of PIs=*8*

methodology. Specifically, it prevents the generation of the same final test two or more times for the same fault as this will result in the reduction of the number of detections for that fault below n. Starting with n incompatible seeds for each fault ensures that merging of seeds will produce at least n different tests per fault. Keeping the seed with the largest number of unspecified bits provides more room for merging.

Step (iv) is invoked only when all the decision combinations for a fault (both for the fault activation and propagation) have been exhausted and only fewer than n seeds have been generated. In this step, the seeds with the fewer specified bits are modified by *specifying bits to conflicting values*, to derive two or more incompatible seeds. The output of this process is n different sets of seeds $T_{PF_1}, T_{PF_2}, ...T_{PF_n}$ each containing one seed generated per fault together with the corresponding number of faults detected by each seed saved in $D_{PF_1}, D_{PF_2}, ...D_{PF_n}$ (Fig. 6.6).

We explain this modified seed generation with a comprehensive example summarized in Fig. 6.5. Consider again the circuit of Fig.6.3 and assume seed generation for fault f_x stuck-at-0. The values inside brackets denote the controllability values for 0 and 1, respectively [85]. The generation algorithm selects the line with the smaller controllability metric for logic value 1 i.e., line c to get value 1. In order to justify $c = 1$, the algorithm performs another decision i.e., $i_6 = 1$ and directly implies that $i_8 = 1$ (direct implications for each decision is shown dashed outlined next to its node). The seed XXXXX1X1 is generated by taking the leftmost path of the tree in Fig. 6.4 (step (i)). This step is shown in row 1 of Fig. 6.5. Next the decision closest to the fault is altered and the input with the next smaller controllability is selected, i.e., $a = 1$. This decision's direct implications ($i_1 = 1, i_2 = i_3 = 0$) generates seed 100XXXXX (step (ii)) which, however, is discarded since it is compatible with the initial one (step (iii)) shown in row 2 of Fig. 6.5. In the same way the next decision is taken and a new seed XXX1X001 (seed #2) is generated which is not discarded as it contains a conflicting bit 6^{th} with seed #1 (row 3 in Fig. 6.5). The process goes on until n different tests are generated or until all decision combinations have been examined. For the specific example and for $n = 5$, steps (i) - (iii) have produced only 2 different seeds and, hence, step (iv) is necessary. After selecting the seed with the fewer specified bits, seed #1 is replaced by two other seeds one setting its 7^{th} bit (chosen randomly) to value 1 and one setting the same bit to value 0 (column 5 in Fig. 6.5). This process is repeated two more times to generate two more tests (columns 7 and 9 in Fig. 6.5) i.e., until 5 different seeds are generated. Step (iv) guarantees to produce distinguished seeds,since from previous steps no compatible seeds are allowed to reach step (iv) and the bit fixing process ensures conflicting bits in the obtained seeds. Each of this bit will be placed in a different seed set T_{PF_i}.

Although the proposed multiple-seed generation process has a number of additional constraints and discards a significant number of seeds, it is necessary to maintain the n-detect property. Moreover, it drastically simplifies the following merging process, since it will consider much fewer constraints among seeds. The procedure is also efficient since the n different seeds are generated in an incremental test generation process which is much faster that n independent seed generations. This is the reason why the multiple seeds generation for the same fault is not chosen to be executed in parallel in the proposed method.

6.1.2 Clustered Dynamic Seed Merging

When n seeds are generated per modeled fault, the methodology proceeds to the dynamic merging (as in Fig. 6.1). The main challenge in this step in order to ensure the n-detect property is to prevent merging for generating two identical tests explicitly targeting the same fault. This undesired situation occurs when either:

- different seeds for the same fault are merged independently with other faults' seeds, incidentally resulting at the same test, or
- different seeds for the same fault are merged together.

However, both these cases are implicitly prevented because seed generation ensures that the seeds generated for the same fault have at least on conflicting bit which each other. Thus, the only remaining issue is to ensure that the merging will not affect the speed-up scalability of the methodology. Leaving the merging process identical to the one presented in Algorithm 4, results in significant reduction in the obtained speed-up by 10% 25%. These observations have been made by employing corresponding experimentation which are not presented here due to space limitations.

To mitigate the speed-up reduction, the parallel framework has been extended to a cluster-based approach where the available cores are partitioned into n clusters. Cluster i explicitly targets dynamic merging for a subset of seeds (i.e., T_{PF_i}) obtained from the seed generation of Subsection 6.1.1 (Fig. 6.6). Inside each cluster the dynamic merging procedure is identical to that described with Algorithm 4; yet the number of calculations is significantly smaller than the non-clustered merging as there are fewer seeds to pairwise compare with. While each cluster operates on its own T_{PF_i}, fault dropping is performed via the global fault list located in the shared memory. Hence, any fault detection due to seed merging or identified during the subsequent fault simulation (lines 13, 21 & 24 of Algorithm 4) updates the global fault list, reducing the number of desired detections by 1 per obtained test. When this number becomes 0 the corresponding fault is completely dropped from further consideration. This n-detect aware global fault dropping makes sure that merging will not continue to consider seeds corresponding to dropped faults.

The clustered-based dynamic merging produces n independent test sets with 100% coverage of the given faults. These n sets are combined into a unified test set by eliminating duplicate tests to provide the final n-detect test set as shown in Fig. 6.6. This elimination does not affect the n-detect property of the obtained set since by construction the methodology prevents the generation of the same test two or more times explicitly targeting the same fault. In other words, if two tests are identical (each coming from a separate cluster) they have been generated by merging seeds corresponding to different faults. This is implied by

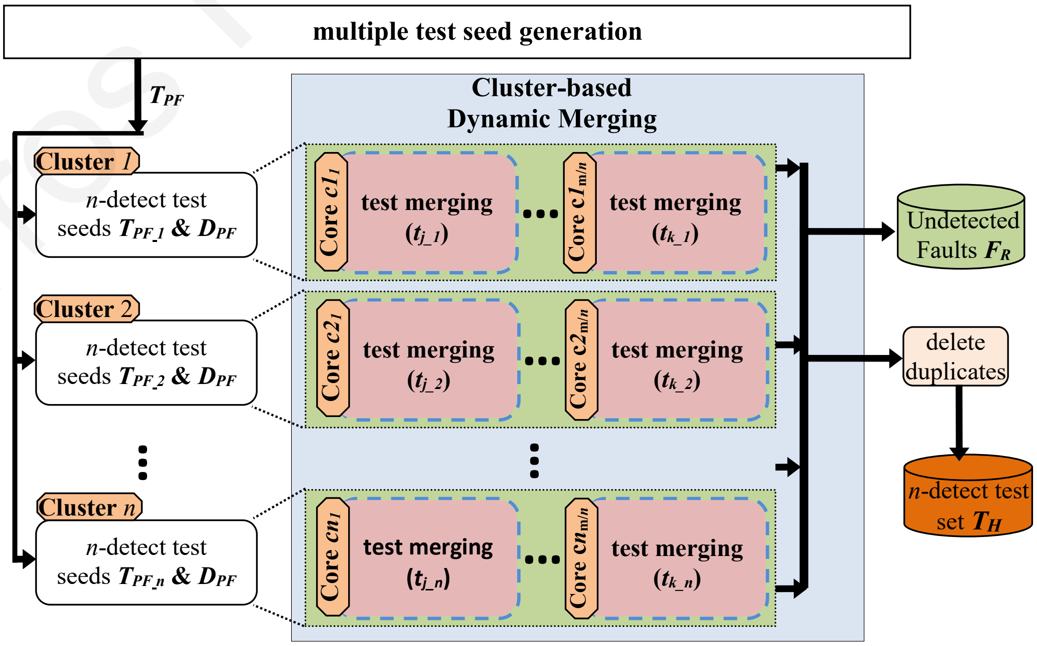

multiple test seed generation
T_{PF}
Cluster-based
Dynamic Merging
Cluster 1
n-detect test seeds T_{PF_1} & D_{PF}
Cluster 2
n-detect test seeds T_{PF_2} & D_{PF}
Cluster n
n-detect test seeds T_{PF_n} & D_{PF}
Core $c1_1$
test merging (t_{j_1})
Core $c1_{m/n}$
test merging (t_{k_1})
Core $c2_1$
test merging (t_{j_2})
Core $c2_{m/n}$
test merging (t_{k_2})
Core cn_1
test merging (t_{j_n})
Core $cn_{m/n}$
test merging (t_{k_n})
Undetected Faults F_R
delete duplicates
n-detect test set T_H

Figure 6.6: Cluster-based dynamic merging example.

the constraint imposed during seed generation that all seeds for the same fault must contain at least one conflicting bit and, hence, cannot be part of the same final test.

6.1.3 Parallelization Optimizations

Scalable Parallel Fault Simulation. Fault simulation is used in many cases in the proposed methodology and, thus, its performance significantly affects the overall performance. Specifically, fault simulation is used two times is each Epoch:

(i) To find the number of detected faults per test seed without fault dropping. This information is given as input to the merging procedure in order to avoid repeated simulations after each merging.

(ii) At the end of the merging step in order to detect as many coincidental faults as possible and, hence, minimize the test set size.
In case (i) the fault simulation is performed after test seeds have been generated for all faults. Since generation in the various cores is executed independently (only for the faults assigned), the cores finishes this step in different times, resulting in idle cores. These cores can be utilized for fault simulation in a parallel fashion. The challenge here is that the number of idle cores is changing (increasing) as more cores finish and, hence, the simulation should utilize them as well. To take full advantage of this situation (no core remain idle) we have fully incorporated for this case the highly scalable parallel fault simulation of [28]. This fault simulator has been shown to provide linear speedup as the number of cores increases and can be dynamically adjusted to the number of available cores. In case (ii) the fault simulation should proceed within one core since, after the merging step, the final test should be simulated to identify co-incidental fault detections (see Fig. 6.1).Recall, that in this step the test seeds are dynamically acquired by cores from the shared fault list and merged with other seed until no more merging is possible. Hence, fault simulation cannot run in parallel to get maximum benefit. Nevertheless, the fault simulations exploits bit-parallel simulation principles presented in [28] where many faults (equal to the machine word size w) are simulated with a single circuit traversal.This results in a considerable speedup of this step by a factor very close to w.

Shared Memory Access Avoidance. Access to the shared-memory must be efficient and well targeted in order to avoid memory contentions. The proposed method access the share resources thoughtfully using the following ways: (i) During *Seed-based Test Generation* and *fault simulation* phases shared-memory access is avoided since cores work independent in a

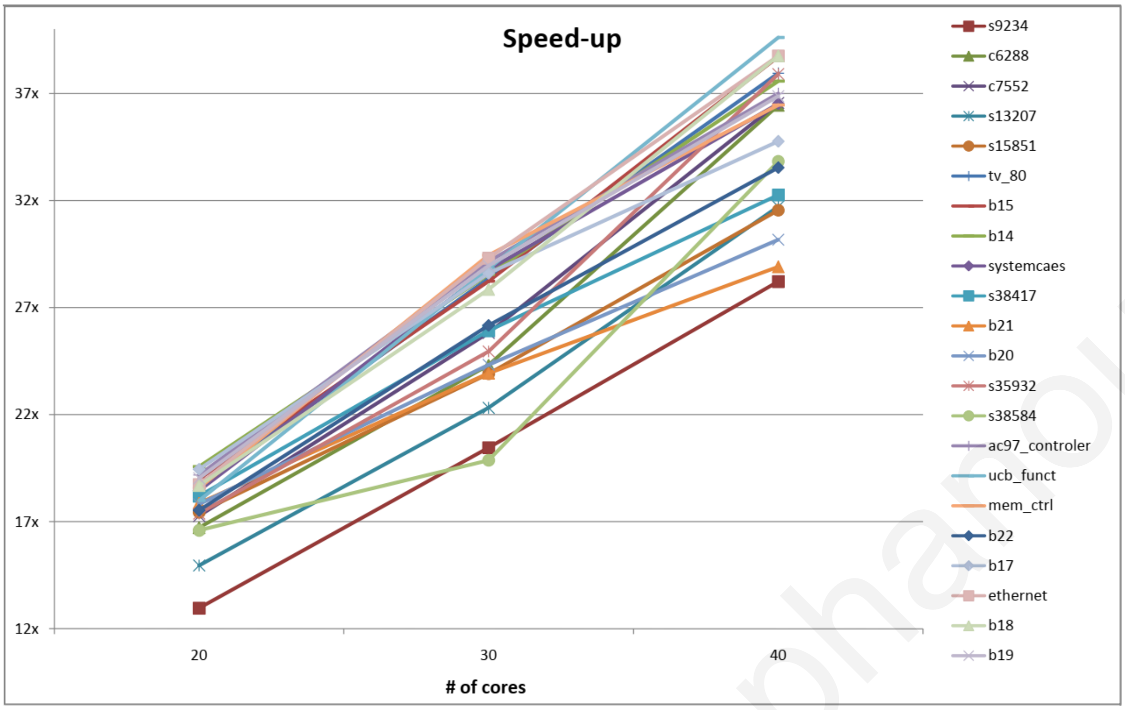


Figure 6.7: Scalability of the proposed parallel n-detect test generation with respect to a serial execution, for n=5.

test seed basis. However, during *Dynamic merging* phase share memory access can affect the efficiency and the quality of the test generation method. Appropriate bookkeeping with shared fault list F ensures that no test is simulated twice for the same fault and that faults will be dropped only after they are detected n-times. (ii) Due the the dynamic nature of proposed method the best candidate test seeds (small number of sp. bits or high number of detected faults) would be attractive by many cores. During merging phase cores are initially working independently on their own private space for seeds assigned to them ($\boldsymbol{T}_{PF}$) and updating of the shared memory is only done at the end before they visit the remaining test seeds in a circular array manner (Test Set Private Consideration). (iii) During *Dynamic merging* phase pair-wise compatibilities between tests seeds are calculated once per test seed for $\boldsymbol{T}_{PF}$ (stored at $\boldsymbol{P}_i$). Access to shared memory is not necessary since all $\boldsymbol{T}_{PF}$ belong to the private space of the cores and only after they finished processing on $\boldsymbol{P}_i$ and only few non-dropped tests remaining shared memory is utilized.

6.1.4 Experimental Results

A comprehensive picture of the linear performance of the proposed n-detect test generation extension for n=5 is presented in Fig. 6.7 and Fig. 6.8. The scalability of the technique is illustrated in Fig. 6.7 compared to a serial execution of the algorithm. The achieved speed-up is reported in y-axis for different number of cores utilized (shown in x-axis). For all circuits examined, the proposed methodology scales linearly as the number of cores used for its computation increases. Moreover, observe that for larger circuits the obtained speed-up is higher due to the increased workload that allows smaller percentage of core idle times. As with the single-detect case the results imply that the scalability of the proposed method will continue scale well as the number of cores is increased. In fact, while the speed-up trends are similar to that of the single-detect in absolute values, the speed-up is higher in n-detect and in some cases close to the theoretical maximum (number of utilized cores). For example, two of the largest circuits considered b18 and ethernet achieve ∼19x speed-up for 20 cores, ∼28x speed-up for 30 cores and ∼39x speed-up when 40 cores are utilized. The corresponding numbers for single-detect are ∼15x, ∼22x and ∼29x. This is mainly attributed to the cluster-based approach that dramatically reduces the number of necessary condition checks during merging. The checks are restricted only inside the cluster where the seed to be merged was assigned. This possibility cannot be exploited in a serial approach in a straightforward manner, without significant effect either at the performance or the final test set size.

Similarly to the results presented in Subsection 5.3.3 the parallelization procedure affects the final test set size. Moreover, for the n-detect approach, due to the clustered-based merging, the solution is obtained from a reduced search space. Hence, it is expected, in many cases, to provide a sub-optimal solution that returns test detecting fewer faults. When fewer faults are detected per test, it is inevitable that the final test set will be larger (test set inflation). Fig. 6.8 shows the test set size increase as the number of cores used is increased for the examined benchmark circuits. The increase is reported as a percentage of the test set size obtained by a serial execution of the same algorithm. Unexpectedly, the test inflation for the n-detect method is not significant. In particular, for the 5-detect test generation the test inflation is 5% for 20 cores, 7.5% for 30 cores, and 9.5% for 40 cores whereas the corresponding numbers for the single-detect case are 7%, 8.5% and 9%.

6.2 Chapter Summary

This Chapter presents an extension of the parallel test generation method presented in Chapter 5 to generate multiple-detect (n-detect) test sets. n-detect test sets are proven to provide higher defect coverage. The target for the method is to avoid assigning the same workload to multiple cores, while the distribution of work in the available resources aims to minimize

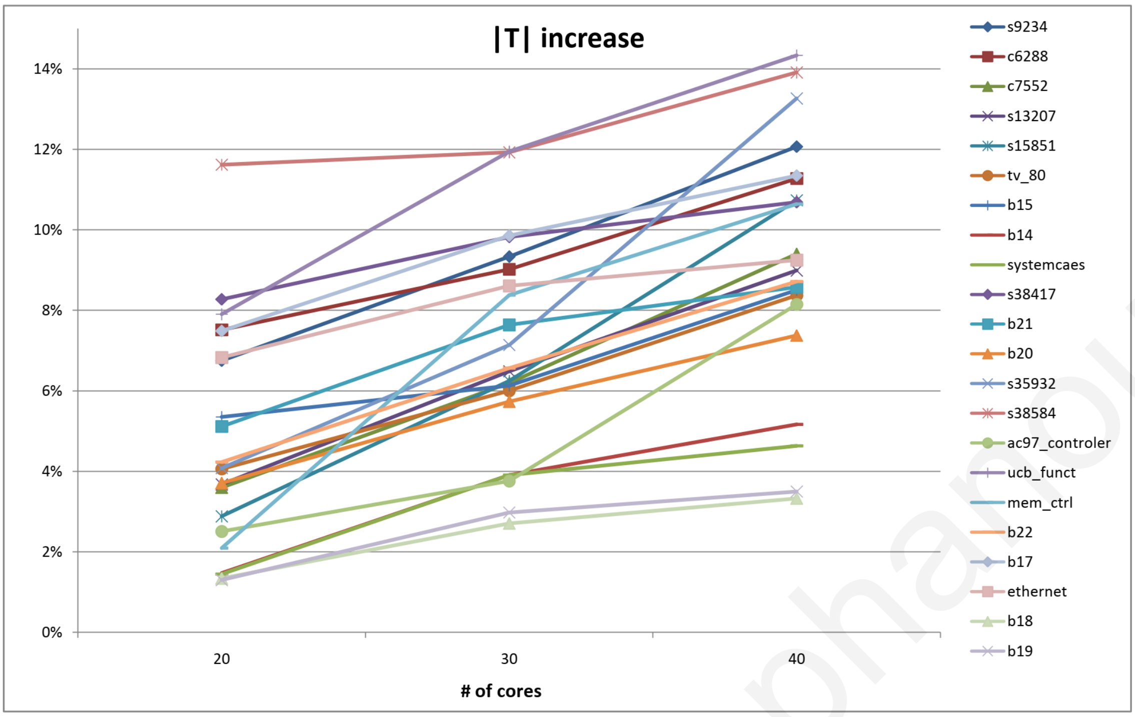


Figure 6.8: 5-detect test set size (|T|) increase % compared to serial execution of the proposed method.

the core idle time. A new technique for the efficient generation of test seeds followed by a clustered-based dynamic merging procedure have been presented. Experimental results demonstrate high speed-up rates that keep increasing as the number of the available cores increases. Test set size increase is limited and comparable to other state-of-the-art parallel approaches.

Chapter 7

Exercise Vectors Generation for Reliability Enhancement in Multiprocessors

This chapter explores reliability techniques using ATPG concepts to deterministically generate exercise vectors which can be utilized to prolong CMPs lifetime. The Chapter begins with an outline of the basic transistor-level models for physical failure mechanisms, Hot-Carrier Injection (HCI)- and Negative Bias Temperature Instability (NBTI). Particularly Section 7.2 investigates the NoC-based Chip Multi-Processors where a single failure on the inter-processor NoC could be catastrophic for the system. The goal is to investigate a wearout-decelerating scheme in which routers can have their wearout-sensitive components exercised (using a deterministically generated vectors). As an enhancement of the NoC work, a novel, non-invasive micro-architectural Proactive Reliability Improvement though EXercise Technique, called PRITEXT, is proposed with the goal to improve the lifetime of a CMOS design under NBTI stress using exercise vectors (7.3). PRITEXT leverages path delay test principles to drive near-ideal vectors while simultaneously providing a deterministic algorithm to generate exercise vectors under circumstances where such tests do not exist. The efficiency of the technique is evaluated with a superscalar processor on realistic benchmarks. The work summarized in this chapter is in collaboration with the Texas A&M University and Cyprus University of Technology (CUT). Particularly, Texas A&M and CUT were responsible for modeling of the failure mechanisms and the identification of the wareout-sensitive components and evaluation while, the work presented in the thesis was mainly focused on the definition and the implementation of the exercise vector generation algorithms.

7.1 Preliminaries - Basic concepts

Moore's Law scaling is continuing to yield to even higher transistor density with each succeeding process generation, leading to today's multi-core CMPs with tens or even hundreds of interconnected cores or tiles. Unfortunately, deep sub-micron CMOS process technology is marred by increasing susceptibility to wearout. Prolonged operational stress gives rise to accelerated wearout and failure, due to several physical failure mechanisms, including HCI and NBTI. Failure mechanisms correlate with various usage-based stresses which can eventually generate permanent faults. While the wearout of an individual core in many-core CMPs may not necessarily be catastrophic for the system, a single fault in the inter-processor Network-on-Chip (NoC) [100] fabric could render the entire chip useless, as it could lead to protocol-level deadlocks, or even partition away vital components such as the memory controller or other critical I/O. In this thesis, we present the critical path models for HCI- and NBTI-induced wear develop and applied onto the interconnect micro-architecture.

The two dominant CMOS transistor physical failure mechanisms are HCI and NBTI [101]. Under both failure mechanisms charge becomes trapped in or near the gate oxide resulting in a slow increase of the transistor threshold voltage (Vth). This in turn causes the delay in transistor state switching to expand. In traditional synchronous circuit CMOS designs, the clock frequency of a given design is determined by the circuit path which exhibits the longest latency between its end latches, within a given system design. This critical path comprises a chain of connected gates between latches. As HCI- and NBTI-induced aging progresses, it gradually extends the delay of each gate found in this chain, slowing down the entire critical path. In modern CMOS designs, due to this age-induced slow-down, and other causes, such as process variation [102], designs are given timing guard-bands so as to guarantee their intended functionality for a certain duration of time [103]. Once the aggregate increase in delay along a timing-critical path exceeds this guard-band, due to the aggregation of increasing delays occurring in individual gates along this path, the functionality of the system is no longer assured. The moment at which this timing violation first occurs determines the system's useful life span. Of course, HCI and NBTI impact all transistors in the design (not only those in the critical path), however, those on the critical path are more likely to exceed the guard-band causing a critical failure.

Fig. 7.1 illustrates a CMP exposed to wearout failures in various of its components. As literature indicates, individual core wearout and failure need not to be catastrophic to the functionality of many-core CMPs due to the inherent core redundancy that a CMP implies.

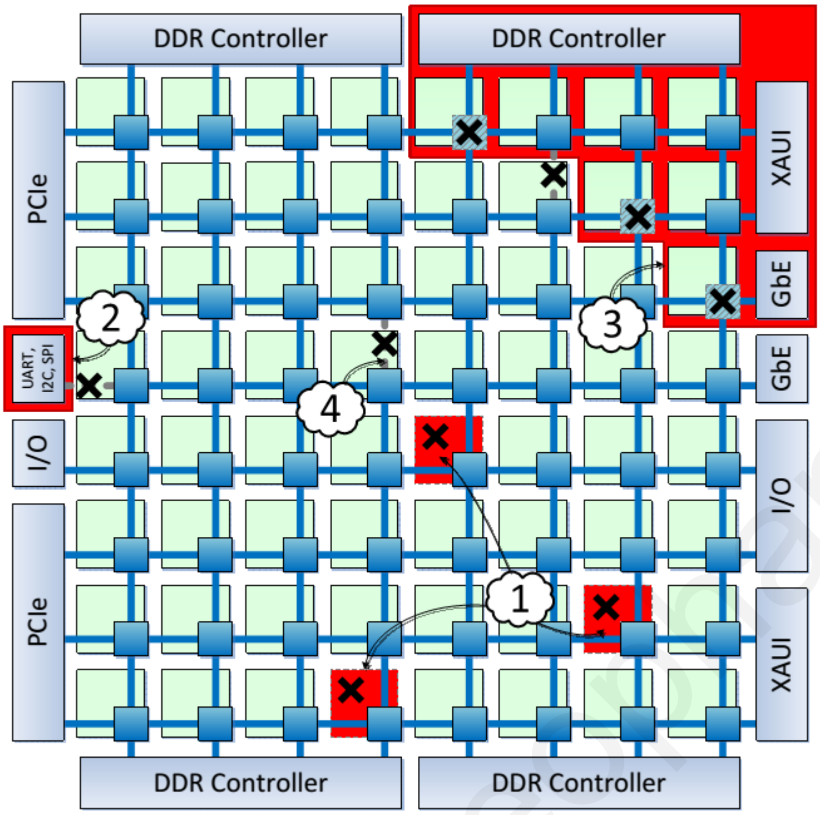


Figure 7.1: A 64-core CMP interconnected with an 8×8 2D mesh NoC. Components marked with a black × illustrate wearout failure. The failure scenarios are as follows: (1) failure of cores; (2) peripheral device disconnected from the system due to link failure; (3) network segmentation resulting in a disconnected sub-network; (4) individual link failure

With increasing numbers of cores, a proportionally smaller portion of the overall system's required throughput is dependent upon each individual core. The component failure scenario (1) of Fig. 7.1 shows this case. Failure caused by wearout of some cores need not result in full-system failure. Instead the system could suffer some performance loss while preserving correct functionality, assuming core-level error detection and appropriate system support is available [68, 104–108]. For the NoC interconnecting the cores, however, the assumption of redundancy based wear resilience breaks down, (c.f., component failure scenarios (2), (3) and (4) of Fig. 7.1). Scenario (2) illustrates the case where a wearout-induced link failure precludes access to a key I/O peripheral, while in scenario (3) link and router wearout has partitioned away a large fraction of the network, making those cores and I/O components inaccessible from the rest of the system. In both cases, wearout is catastrophic, in that the system will likely be rendered unusable due to these failures, unlike the core wearout in scenario (1) discussed earlier. Even scenario (4), in which a single link is broken due to wear-induced failure, might lead to a communication protocol-induced deadlock(s), or subnetwork isolation, if the network is not provisioned to address wear induced failures.

Various fault-tolerant routing algorithms have been proposed and fault insensitive router and link designs in an attempt to manage faults as they occur [79, 109, 110], however, network isolation and key resource partitioning cannot be fully resolved using only such reactive techniques. Ideally, one would prefer to develop proactive mechanisms to extend the healthy status of the system without failure, rather than react to the faults once they occur. Such proactive mechanisms could be coupled to the reactive mechanisms, in the hope that the latter would be required less frequently as faults in the system would occur less frequently.

7.2 Use It Or Lose It: Proactive, Deterministic Longevity in Future Chip Multiprocessors

For an NoC-based CMP the aging process is highly depended on the incoming rate along the critical path. Usually, the gate delay increases and the timing constraints are violated along the critical path first. A low incoming rate causes a biased duty cycle in the wires along the critical paths. These biases accelerate NBTI, thus the router requires an increased incoming rate to improve its longevity. However, increasing the incoming rate artificially yields other problems such as increased power consumption and acceleration of the HCI effect. The duty cycle must therefore be improved without increasing the activity factor significantly. We note that, although duty cycle and activity factor are related, it is possible to reduce duty cycle of a node substantially without increasing the activity factor substantially, by infrequently changing the value of that node. Hence we will investigate for a method to exercise the critical path, which improves the duty cycle while minimally disturbing activity factor, improving NBTI without substantially impacting HCI.

The requirements for the mechanism under investigation are:

(1) to improve the duty cycle by allowing the circuits to operate at a greater portion of their time in the ”*1*” state, without affecting the actual data values being transferred,

(2) to not change the state of the router,

(3) to not worsen the critical path timing, and

(4) to not significantly increase the activity factor.

The overall goal is to identify an algorithm that can generate vectors that can exercise the critical path. Then, the generated exercise vectors can be used for balancing the duty cycle of the nets on the critical paths. Vectors are deterministically generated in a process that resembles to an ATPG process. However, the problem examined in this thesis resembles to an easier, restricted version of the ATPG problem. The process of exercising the value '*1*' at some critical net *f* corresponds to *activating* the stuck-at-0 fault at *f*. No *propagation* is necessary in this case, hence, it suffices to *justify* the activation value in order to generate the necessary exercise vector. More details for can be found on [78].

7.2.1 Lifetime-Extending Router Microarchitecture

Aging process is strongly related with incoming rate-dependent along the critical path. The gate delay increases and the timing constraints are violated along the critical path first. A low incoming rate causes a biased duty cycle in the wires along the critical paths, because those paths deal with allocation corner-cases which are rare unless the load is very high. These biases accelerate NBTI, thus the router requires an increased incoming rate to improve its longevity. However, increasing the incoming rate artificially yields other problems such as increased power consumption and acceleration of the HCI effect. The duty cycle must therefore be improved without increasing the activity factor significantly. We note that, although duty cycle and activity factor are related, it is possible to reduce duty cycle of a node substantially without increasing the activity factor substantially, by *infrequently* changing the value of that node. Hence we propose a method to exercise the critical path, which improves the duty cycle while minimally disturbing activity factor, improving NBTI without substantially impacting HCI.

The netlist which represents the combinatorial logic in a pipeline stage can be represented as a Directed Acyclic Graph (DAG) with a set of primary inputs and a set of primary outputs. All vertices of the graph comprise the gate instances, while the graph edges represent the connections between the gates. A timing arc on this DAG can be defined as a path from any of the primary inputs to any of the primary outputs. By starting at the end-point of a timing arc and building the logic cone backwards till a set of primary inputs are reached (basically a graph traversal using Breadth First Search or Depth First Search), all the logic gates which affect that particular path can be extracted. We have constructed such a connectivity graph for our netlist, obtained after synthesis of our baseline router. The critical path logic is extracted by constructing the logic cone for each of the timing paths which have slack of less than 10%.

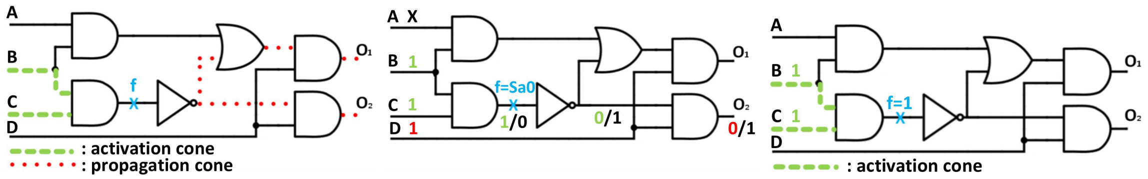


Figure 7.2: (a) Activation and propagation cones for fault location *f*; input signals *B, C* (*A,B,C,D*) determine activation (propagation), (b) test generation for *f* stuck-at-0; *B=1* and *C=1* activate the fault and *D=1* propagates its effect to O_2; possible test vectors *ABCD=X111 ={0111, 1111}*, (c) let *f* be a critical net; exercising *f=1* requires activation of *f* stuck-at-0 with *B=1* and *C=1*.

7.2.2 ATPG Preliminaries for Vector Generation

Exercise data is injected during the exercise mode of the router for the purpose of balancing the duty cycle of the nets on the critical paths. In order to optimize this process we consider deterministic generation of the data to be injected. This particular problem resembles the ATPG process, a well-known NP-complete problem [111] used for manufacturing tests for integrated circuits [112]. The ATPG process involves the generation of a set of vectors, called tests, which are applied to each manufactured circuit in order to detect possible defects. ATPG is typically performed at the gate-level, using predefined fault models such as the established stuck-at-fault model in which each signal may be stuck to either the logic *"1"* or the logic *"0"* value.

The basic ATPG procedure followed in generating a test vector for stuck-at fault tests comprises two phases: the fault activation phase and the fault propagation phase. During fault activation the fault location (signal) is activated by injecting the opposite of the fault value. The part of the netlist driving the fault location is referred to as the activation cone. The fault propagation phase involves the propagation of the fault effect to some observable output signal. The part of the circuit driven by the fault location is referred to as the propagation cone and it contains all the possible propagation paths from the fault location to the output signals. Fig. 7.2 illustrate the activation and propagation cones for the fault location *f* in the given netlist.

During ATPG, a signal justification procedure is performed during each of the two phases. Justification during fault activation determines values on the input signals to allow for the activation of the fault, whereas justification during fault propagation determines the values of remaining input signals to allow for fault propagation via some propagation path. Fig. 7.2 illustrates one such scenario which sets *B=1* and *C=1* during the activation phase for the fault *f* stuck-at-0, and *D=1* in order to propagate the fault to the output signal O_2. It is noted that

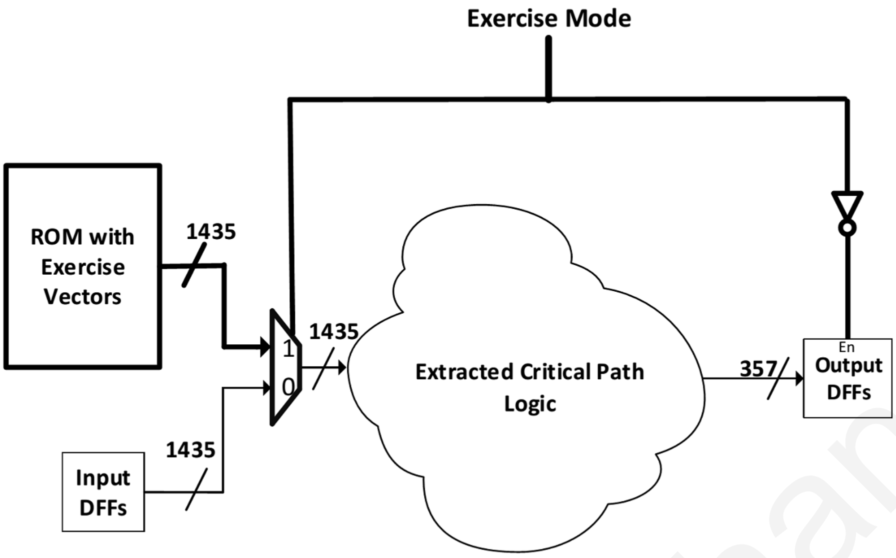


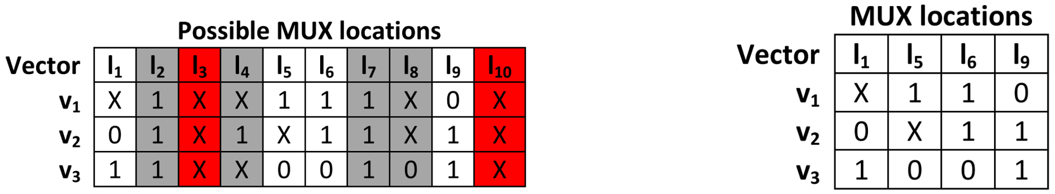
Figure 7.3: Critical Path Logic with proposed exercise logic. Additional exercise logic is darkened.

signal A is not set and assumes the don't care value (X) which implies that it can be set to any of the two logic values. In this example, if a stuck-at-0 fault exists at f, the value at the output O_2 is '*1*', otherwise it is '*0*' (the composite value $\mathrm{v}_{ff}/\mathrm{v}_f$ stands for fault-free value v_{ff} and faulty value v_f at f). We note that typically the fault propagation phase in ATPG is harder than the activation phase as it involves the selection of propagation paths and constrained justification based on the results of the activation phase. Nevertheless, both processes are NP-complete due to the justification process which is, in the worst case, exponential to the number of input signals.

The problem examined in this work resembles an easier, restricted version of the ATPG problem discussed above. The process of exercising the value '*1*' at some critical net f corresponds to activating the stuck-at-0 fault at f. No propagation is necessary in this case, hence, it suffices to justify the activation value in order to generate the necessary exercise vector. For example, it suffices to set *B=1* and *C=1* in Fig. 7.2c) in order to exercise signal f (which could belong to the critical netlist). The generated vector in this case is *ABCD= {X11X}*.

7.2.3 Optimization of Hardware Overhead via Compaction of Exercise Data

Moreover, the generated exercise vectors need to be stored in order to be utilized during NoC idle periods. A considerable part of the hardware overhead of the exercise logic given in Fig. 7.3 consists of the ROM which stores the exercise vectors as well as the various



Possible MUX locations

Vector	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
v_1	X	1	X	X	1	1	1	X	0	X
v_2	0	1	X	1	X	1	1	X	1	X
v_3	1	1	X	X	0	0	1	0	1	X

(a)

MUX locations

Vector	l_1	l_5	l_6	l_9
v_1	X	1	1	0
v_2	0	X	1	1
v_3	1	0	0	1

(b)

Figure 7.4: ROM size: (a) Possible ROM size 3x10 with 10 possible MUX locations, (b) necessary ROM size 3x4 with 4 + 4 MUX locations

multiplexers (MUXes) inserted to allow for the ROM vectors to be exercised. Both the size of the ROM and the number of new MUXes is data-dependent on both dimensions of the exercised data matrix. To better understand this issue consider the example in Fig. 7.4 which shows 3 exercise vectors. The row dimension of the matrix depends on the number of exercise vectors, 3 for this example. Hence, the generation procedure should attempt to minimize the number of exercised vectors by generating vectors that exercise a large number of critical nets. The ATPG tool will be based on an efficient test vector compaction algorithm in order to satisfy the goals [112].

The column dimension contains the exercise data feeding each new MUX (up to 10 in this example). A straight forward implementation requires a ROM of size 3×10 and 10 new MUXes for this example. However, we observe that each MUX's data can fall in one of three categories. In the first category all data have the don't care value (columns 3 and 10 in Fig. 7.4). These columns can be removed from the ROM. Furthermore, no MUX is necessary for these signals. In the second category we have columns that can assumes either the constant '0' or constant '1' value (columns 2, 4, 7, 8). These columns can also be removed from the ROM but still require a corresponding MUX set to the constant value. In the third category both a MUX and a ROM column are needed as the value of the MUX data varies among different vectors (columns 1, 5, 6, 9 in Fig. 7.4a). We define all MUXes in the first category as MUX_X, those in the second category as $MUX_0 + MUX_1$ and, finally, those in the last category as MUXROM. Using the above analysis the final ROM size in this example is (3×4). The number of necessary MUXes is the number of signals driven by a ROM column plus the number of columns with constant values computed by $MUX_{ROM} + MUX_1 + MUX_0$, which is 4+3+1=8 ($MUX_{ROM}$=$l_1, l_5, l_6, l_9$, MUX_1=l_2, l_4, l_7, MUX_0=l_8, MUX_X=l_3, l_{10}). Clearly the existence of don't care bits (X) in the vector set enables ROM compaction towards the column dimension as well as reduction of the necessary new MUXes. Hence, the vector generation procedure should aim towards a compacted vector set to exercise the critical nets which, (a) has a small number of vectors and, (b) has a large number of don't care bits in each vectors.

Such an approach is described in the next paragraphs.

Algorithm 5 Deterministic vector generation algorithm.

Procedure Exercise Vector Generation ( )
Inputs: Baseline router netlist ***R***, critical nets list ***N***, duty cycle per critical net ***D***
Outputs: Set of exercise vectors ***V***, list of exercised critical nets $\boldsymbol{N_e}$

```
Sort the elements of the critical nets list $\boldsymbol{N}$ based on $\boldsymbol{D}$
$\boldsymbol{N_e}$ = NULL;                    // list of exercised critical nets
$N_{red}$ = NULL;                   // list of redundant critical nets
$j=1$;                              // exercise vector index
while ($\boldsymbol{N} \neq \emptyset$)
    $v_j = X$;                      // initialize $v_j$ with all unassigned values (don't cares)
    $\forall$ critical net $n_i \in \boldsymbol{N}$          // for each net not exercised yet
        $v_j$'= justify ($\boldsymbol{R}$, $n_i$, $v_j$);   // justify additional values of $v_j$ in order to exercise $n_i$
        if ($v_j$' != NULL )
            add $n_i$ in $\boldsymbol{N_e}$ and delete $n_i$ from $\boldsymbol{N}$
            simulate $v_j$' on $\boldsymbol{R}$
            $\forall$ $n_k \in \boldsymbol{N}$              // for each net not exercised yet
                if ($n_k$ == 1)
                    add $n_k$ in $\boldsymbol{N_e}$ and delete $n_k$ from $\boldsymbol{N}$
            $v_j = v_j$';                // update current vector
        else
            add $n_i$ to $N_{red}$ and delete $n_i$ from $\boldsymbol{N}$
    add $v_j$ in $\boldsymbol{V}$
    $j$++;
return $\boldsymbol{V}$, $\boldsymbol{N_e}$;
```

The target for the vector generation algorithm (briefly outlined in the following paragraph and 5) is to generate a small number of vectors, each with a large number of unspecified bits, which exercise all nets on the critical path logic. The input to the algorithm is the critical path logic of the router and the list of critical nets N with corresponding duty cycles D. Priority is given to nets with high duty cycle, even though all nets are considered. The output of the algorithm will be a set of vectors V and a list of critical nets exercised. Starting with a vector with all unassigned values (v_j = X) the proposed algorithm iteratively attempts to exercise as many critical nets as possible by justifying values on the current vector v_j. After each successful justification the vector (v_j) is simulated to check for the existence of additional critical net activations that can also be exercised by v_j, which are then deleted from N. When no more nets can be exercised, the generated vector v_j is added to the final vector set V and the procedure is repeated again with a completely new vector (with all unassigned inputs) until all the critical nets are exercised (N is empty) or are classified as redundant. Redundant nets are the nets that cannot be exercised under any input assignment and identification of those nets can indicate a possible problem in the synthesis of the router. We did not have

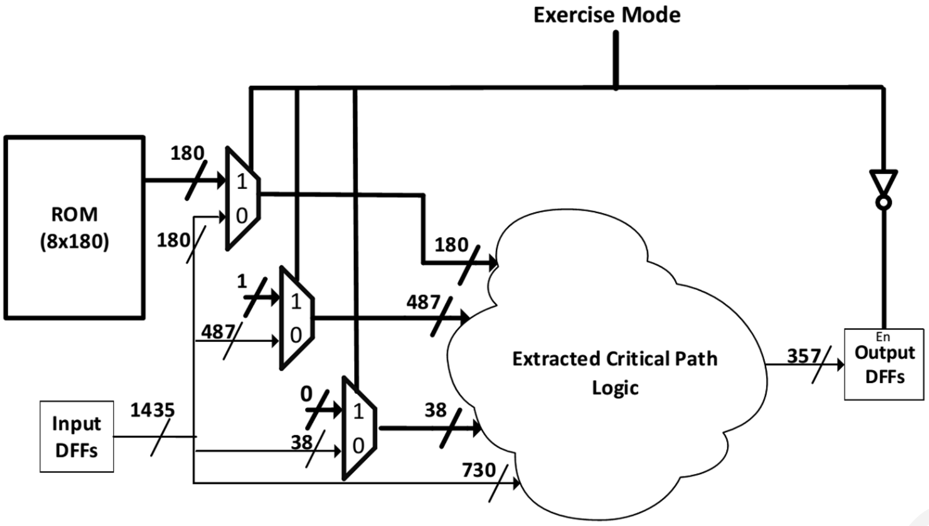


Figure 7.5: Critical path logic with proposed exercised logic (shown in bold), after vector generation.

any redundant nets in the extracted critical path logic circuit, but the proposed algorithm also covers this case for completeness purposes.

The algorithm (5) has two goals. Firstly, to generates a small number of vectors. This can be achieved by forcing each vector to exercise as many critical nets as possible by explicitly targeting them and, furthermore simulating the vector values for any other critical nets that may be exercised without explicitly being targeted during each iteration (lines 7-17). The second goal is to have a large number of unspecified bits in the generated vectors in order to optimizing the hardware overhead via compaction of exercise data. This might be achieved by using a variant of a powerful ATPG justification procedure [86]. The necessary justification procedure (line 08) will be iteratively executed and only the necessary vector bits will be specified during each iteration. In this manner, the generated vector should contains a large number of don't care bits.

7.2.4 Vectors Generation Results and Underlying Exercise Logic

Fig. 7.5 shows the additional exercise mode logic added to the extracted critical path logic of the baseline router. The extracted critical path logic circuit consists of 1,435 inputs, 357 outputs connected to flip-flops inside VCs, and 14,653 internal nodes. From the extracted circuit, the critical path logic consists of 732 critical nets which need to be exercised. Using the deterministic vector generation algorithm (Fig. 5), *eight* vectors are generated which exercise all of the 732 critical nets at least one time (some of them are exercised more than once). After the generation of the vectors, we follow a similar procedure with the one discussed in Section 7.2.3 in order to optimize the hardware overhead (ROM size and number of

MUXes). From 1,435 inputs which correspond to possible MUX locations, 730 have don't care values (MUX_X) and can be removed from the ROM, while 38 can be set to constant value *'0'* (MUX_0) and 487 can be set to constant value *'1'* (MUX_1). Therefore, the necessary ROM size is (8×180) ($= 1,435 - 730 - 38 - 487$) with 705 ($= 180 + 38 + 487$) MUXes (525 of the MUXes are having a constant value on their input pin) shown in Fig. 7.5.

7.2.5 Evaluation

Experimental Setup

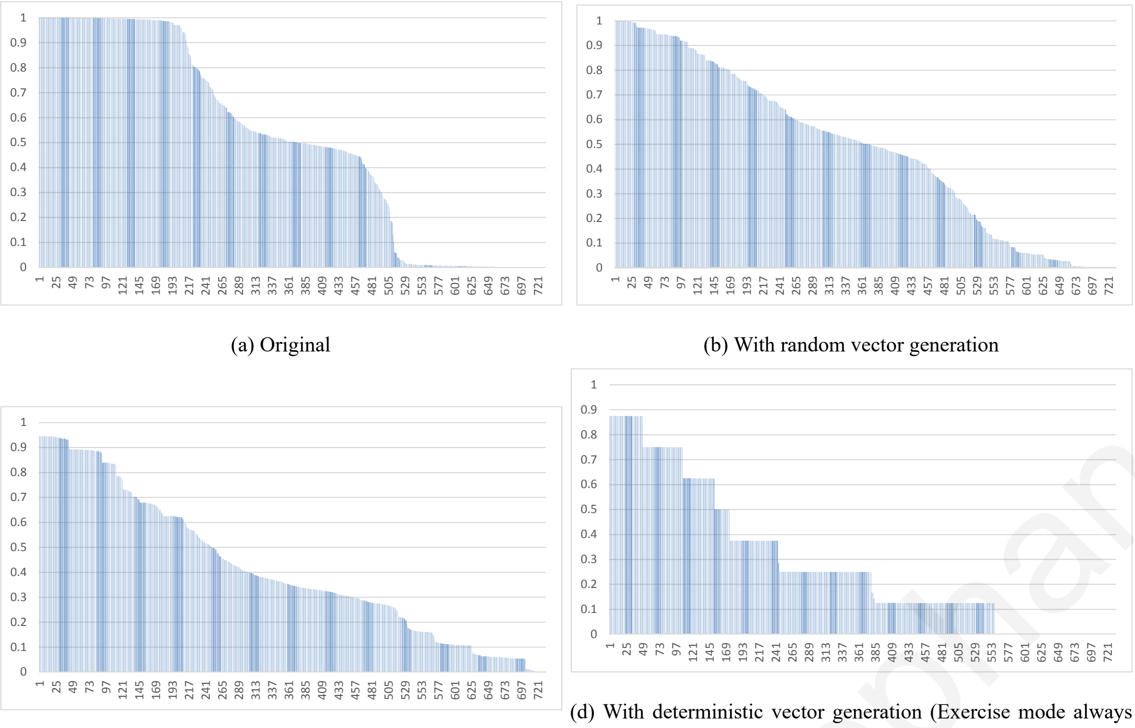
The baseline router, adapted from RTL code made publicly available by Becker [113], contains three pipeline stages. It is synthesized using Synopsys Design Compiler mapped to a 45 nm technology library at 1 GHz. The critical paths were extracted using Synopsys Design Vision. All paths with ≤ 10% slack were retained and analyzed. The wire activity along the paths are extracted with Synopsys VCS and analyzed offline. The power consumption is also evaluated using PrimeTime.

The router is evaluated under both synthetic and realistic workloads. The realistic workloads are captured as traces from gem5 [114] emulating a 64-core system executing multi-threaded programs from the PARSEC v2.1 suite [115]. We compute incoming rate of each router in 8×8 mesh network's individually under X-Y DOR routing. The per-router min, max and average incoming rates for each application were calculated. These rates are then applied to the synthesized router to extract the activity of its wires. For both synthetic and realistic workloads, we execute the post-synthesis models of both the baseline and proposed routers, for 100,000 cycles, to measure the wire activity.

7.2.6 Experimental Results

Random versus deterministic vector generation

Aging due to NBTI depends on the duty cycles of nodes along the critical paths. We studied the impact of randomly generated vector sets to exercise the critical path nodes. Here we used a set 16 of 1,435-bit random vectors to drive the exercise logic. Sixteen vectors were used as more random vectors did not appear to provide any further reduction in duty cycle. Fig. 7.6 shows the duty cycles of the nodes on critical paths under different scenarios. Here, all simulations are performed under synthetic traffic of 0.02 flits/cycle. As Fig. 7.6a) shows, the duty cycles for baseline router are biased towards either "1" or "0." The nodes with duty



(a) Original

(b) With random vector generation

(c) With deterministic vector generation

(d) With deterministic vector generation (Exercise mode always on)

Figure 7.6: Duty cycles of critical path nodes with 2% incoming flit rate, sorted from highest to lowest.

cycle close to 1 significantly affect the aging due to NBTI. Fig. 7.6b) shows that using random vectors to exercise the critical paths produces improvement, but there are still a number of nodes with duty cycle of ~ 1. We note that here, we must have an exercise vector which is of the same bit width as the number of inputs to the critical path logic (1,435 bits), hence requiring 1,435 random bits per vector in the ROM.

As Fig. 7.6d) shows, the duty cycles improve greatly when the vectors used during exercise mode are generated using the deterministic method described in Section 7.2.3. After optimization, just 8 vectors, each 180-bits wide are enough to exercise all the nodes at least once. The ROM size of 8×180 will also be much smaller when compared to that of $16\times1,435$ for randomly generated vectors. When the exercise mode is always on, the maximum duty cycle that a node can have is 0.875 (7/8) which confirms that all the nodes are exercised at least by one of the generated vectors. In Fig. 7.6c, when synthetic traffic of 0.02 flits/cycle is added to the generated vectors, none of the nodes have a duty cycle of 1, though the results are smoothed somewhat from Fig. 7.6d).

Lifetime under PARSEC workloads

Fig. 7.7 depicts the normalized lifetime of the network using the proposed technique under PARSEC workloads. The lifetime of the network is estimated by computing the acceleration

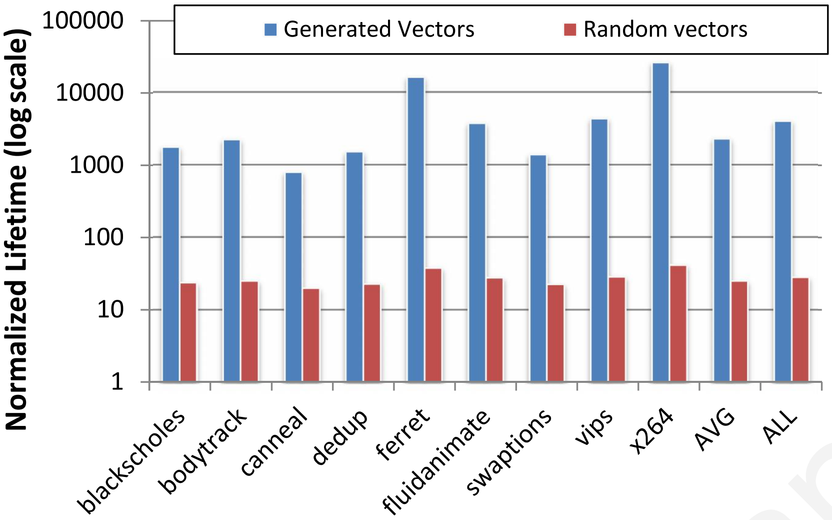


Figure 7.7: Normalized lifetime of the network using the proposed technique under realistic workload.

factor of the router with the minimum incoming rate in the network, as it is the most susceptible to aging effects. The reference system is the baseline router receiving the same incoming rate.

Deterministic vector generation achieves an average of ~2300× reduction in wear rate (bars marked "AVG") as compared to that of random vector generation which only gives ~28× improvement. As expected, the proposed technique performs better when incoming rate is low. An analysis presented in [78] showed that "ferret" and "x264" are the applications with the two lowest incoming rates in the PARSEC suite. Even when the average incoming rate is as high as 0.05 flits per cycle (`canneal`), deterministic vector generation still achieves the normalized lifetime of 800× due to the extreme spread in per-router incoming rates from minimal to maximum seen in that application. The random vector generation does give a little improvement in lifetime but it is no where close to what we can achieve with deterministic vector generation. The bars designated as "ALL" denote a case in which the system executes each of the applications sequentially one at a time. In this case, the improvement becomes ~ 4000×. We found that the execution times of "ferret" and "x264" are the longest among the applications, and hence the incoming rate for "ALL" is dominated by those applications.

7.3 PRITEXT: A Novel Minimal Exercise Vector Generation Technique for Reliability Improvement

This section presents a Proactive Reliability Improvement though EXercise Technique, called PRITEXT, a novel technique which generates a minimal set of deterministic exercise vectors based on test generation techniques which inherently near optimizes the bit patterns across each of the generated vectors; the end target being to exercise the critical paths of a device when dormant so as to achieve near-ideal NBTI stress reduction. Modern processors can be inactive for some periods mainly waiting for events such as input/output or off-chip memory access etc. Those periods can be leverage by PRITEXT to enable the exercise mode on the idle processor for NBTI reduction. The method explores the design space of the generated vectors and results indicate that PRITEXT leads to significant lifetime improvement. In addition, in an attempt to reduce hardware overheads even further, in this section it is presented a heuristic to further reduce the number of exercise vectors with minimum loss in lifetime improvement.

7.3.1 Related Background on NBTI and IVC

NBTI affects the gate of a PMOS transistor (NMOS transistors exhibit a similar, but lesser, effect, i.e., PBTI) when reverse biased, i.e., pulled to logic "0" (V_{gs} = -V_{dd}), by a corresponding internal net, and under continuous stress [116]. This activity leads to generation of interface traps due to disassociation of Si-H bonds in the Si/SiO_2 interface, leading to an increase in the transistor's threshold voltage (V_{th}) and a simultaneous reduction in the drive current due to charge carrier mobility degradation (stress phase). A continuous increase in V_{th} leads to accelerated transistor aging leading to a steady decrease in its switching speed. Hence NBTI does not cause actual PMOS transistor failure, but decelerates its switching.

NBTI has an interesting recovery phenomenon when the PMOS transistor's gate is not reverse biased (V_{gs} = 0); most of the Hydrogen atoms diffuse back and bond with Silicon leading to V_{th} [116] readjustment (recovery phase). However, V_{th} is only partially recovered, depending on the ratio of the stress mode versus the recovery mode, where the net increase in operating threshold voltage due to dynamic NBTI stress is sensitive to the fraction of the time the transistor is under negative bias; this is defined as the duty cycle β. Hence, to further reduce NBTI effects, as self-recovery is obviously not sufficient, the transistor duty cycle should be ideally balanced at 50%.

Input Vector Control

Any internal net of a combinational circuit, which drives the gate of a PMOS transistor, can be deliberately switched to logic "1" by applying a relevant input vector. This logic state-controlling technique, termed Input Vector Control (IVC) [74], is used to activate (i.e., pull-up) the internal nets so as to eliminate the negative bias of the PMOS transistors during standby mode or during device inactivity. Hence, by balancing the ratio of stress time to the total operational time (equivalent to $\beta = 50\%$), NBTI degradation can be greatly alleviated. Balancing the duty cycle (β) of an internal net using IVC forms the basis of the technique in building PRITEXT microarchitecture. We note that a single input vector cannot guarantee the activation of all the internal nets on a single timing path. Hence, the aim with PRITEXT is to derive a set of deterministic input vectors so as to achieve balanced duty cycles across all the internal critical nets. Pulling up critical internal nets (i.e., recovery phase) during standby time using IVC forms the so called "exercise technique" [78], while the set of derived input vectors are dubbed "exercise vectors".

Application of the Generated Exercise Vectors

The focus of the present Chapter is mainly on modeling of the failure mechanisms and identification of the wareour-sensitive components. Moreover, the focus is to define and implement efficient vector generation algorithms based on test automation techniques targeting CMPs architectures. Application of the generated exercise vectors in not the main focus however, the two problems under consideration (NoC and a superscalar processor) allow enough room for vector application. After extensive experimentation with PARSEC and x264 benchmarks under realistic workloads generated with gem5 simulator (more details can be found at [78]) it is identified that the average incoming rate for an NoC is low and causes NBTI-induced aging.

Idle times for processors are also frequent due to the fact that for a high percentage of time they are waiting for events such as off-chip memory access or input/output. This provides opportunities to enable inject the generated vectors for reliability improvement. Based on statistics and a related work for a single core processor system ([71]) it is found out that processors are active and idle in a ratio 1:1 providing enough room for the application of the exercise vectors.

7.3.2 PRITEXT Microarchitecture and Vector Set Overhead Reduction

PRITEXT: Proactive Reliability Improvement though EXercise Technique

In an attempt to exercise all the critical nets such that their β can be balanced at an ideal 50%, it is vital to capture the activation cone of all the critical nets which begins at the fan-in of the critical path as shown conceptually in Fig. 7.8. With the relevant circuit netlist in hand,

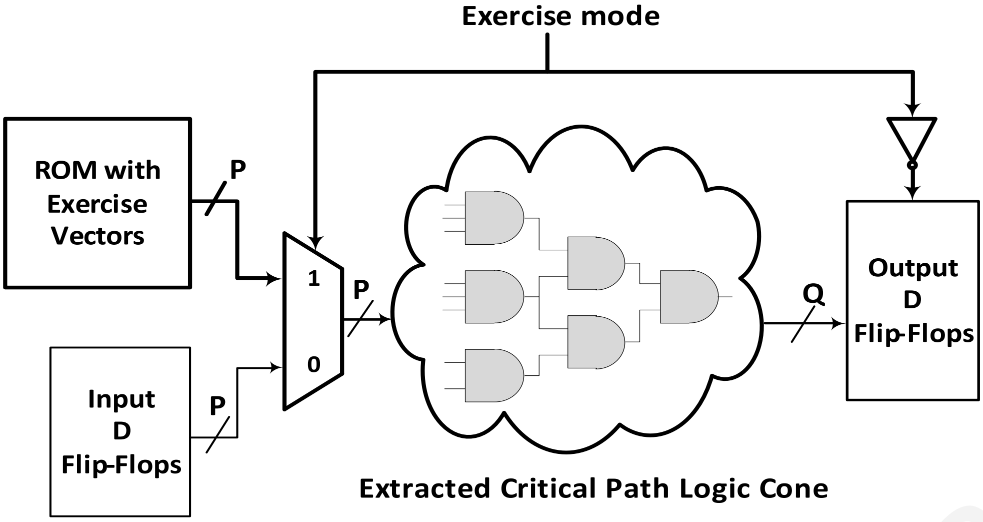


Figure 7.8: Critical path combinational logic cone extraction with proposed PRITEXT exercise logic. Additional exercise logic is shown in a darker shade.

deterministic vectors that exercise all these critical nets are fist generated and then applied. This two-step process defines PRITEXT technique, with the exercise vector generation algorithm described in Section 7.3.2. As seen in Fig. 7.8, PRITEXT is backed by additional hardware which augments the base critical path logic. A Read-Only Memory (ROM) block and multiple lightweight multiplexers (MUXes) store and apply the generated exercise vectors to the critical circuit path respectively. Note that each row in the ROM corresponds to a unique vector. The exercise vectors are applied non-invasively to the extracted critical path logic cone when the system's exercise mode signal is enabled in standby phase, hence not affecting the architectural state of the system; this is ensured by disabling the output flip-flops during exercise mode. The exercise vectors are applied rotatively with a pre-defined periodicity in terms of 100s of cycles (typically 1024) so as to account for the entire critical path tree.

Overhead Reduction of Exercise Vector Set

Exercise vectors generated for NBTI rejuvenation, where their aggregation comprises a vector set, may overload an underlying architecture with considerable CMOS area and power consumption overheads if not optimized. The hardware overhead of a vector set is proportional to the product of the length (i.e., number of vectors) times the width (i.e., size of individual vector) of the vector set.

Reducing the number of exercise vectors (length) offers a number of benefits. First, ROM real-estate occupancy on-chip is reduced (Fig. 7.8), while non-zero dynamic power dissipated as a consequence of critical path nets switching is also kept to minimum when being applied during exercise. Last, the periodicity of exercise application can be enhanced, directly leading to improved circuit rejuvenation during system standby. The latter is especially critical as no single exercise vector can activate all critical nets within a logic cone at once; instead a series of vectors are applied periodically to cover all exercise cases so that the critical circuit cone is rejuvenated holistically. Intuitively, keeping the number of vectors to a minimum value helps in providing a higher fraction of exercise (standby) period for each internal net, thereby achieving a balance of duty cycles across all the critical nets.

On top of length optimization, reducing the size (bit-width) of each individual vector (termed vector compaction) is crucial as it directly leads to a reduced width of vector set; this has a positive net effect requiring a smaller ROM to house it, and less MUXes to propagate it to the critical logic cone. In lieu of these benefits, various heuristics are developed for maximizing the NBTI rejuvenation impact.

2-Dimensional Vector Set Optimization Heuristics

Length optimization: Each gate lying on a critical path does not influence the total switching delay degradation equally due to presence of the $\frac{\beta_i}{(1-\beta_i)}$ factor. This factor can be considered as the contribution of the internal gate (net) towards the total switching delay degradation along the timing path. This inspired us to develop a heuristic such that the vectors which exercise these low criticality nets can be removed from the vector set, to significantly reduce hardware overhead while incurring minor loss in lifetime improvement.

Width optimization: In the general case, a new MUX is needed for each input bit of the exercise vector to enable its injection into the combinational logic cone. However, some inputs always have "don't care" values on all generated vectors. This can occur when powerful vector generation methods are employed to derive vectors with minimally specified bits. As a result, the corresponding ROM columns can be removed and there is no need for a MUX since the values of those inputs do not affect the state of the circuit. Moreover, other inputs may always assert either the "0" or "1" logic value in all the exercise vectors. In such cases, the corresponding ROM columns can also be removed, but a MUX per such input is still required and is set to the corresponding constant value (either "0" or "1").

Deterministic vector generation derived from testing techniques

As discussed in the previous section, vectors are injected during the exercise mode of the circuit in order to balance the duty cycle of the critical nets, which are nets on critical paths with high duty cycle (=$\frac{\beta_i}{(1-\beta_i)}$). The injected vectors are generated offline using a deterministic vector generation algorithm, optimized for the specific problem at hand. In the manufacturing test domain, vector generation is an NP-complete process and its goal is to generate test vectors capable of detecting defects in manufactured circuits. Fault models are used to model the possible defects.

Vector generation based on stuck-at fault modeling

Under the well-known stuck-at fault model, the ATPG process comprises the fault activation and the fault propagation phases. Fault activation requires the injection of the opposite to the fault value at the fault location by justifying the inputs of the circuit. Fault propagation extends the input justification process in order to propagate the fault effect to an observable output.

Critical net activation can be seen as a restricted version of the ATPG problem (since only fault activation is required but not propagation), where critical nets comprise the faults that need to be activated at the stuck-at-0 value. Stuck-at-0 activation includes the justification of the logic value "*1*" at the net by setting the necessary values at the inputs of the circuit which comprise the generated vector. The goal is to generate as few vectors as possible, collectively capable to exercise all critical nets. In the best case, *a single vector may exist capable of activating all stuck-at-0 faults on the targeted critical nets*. Often, as this is a very strict condition to satisfy, several vectors are needed where each one activates multiple critical nets.

Vector generation based on path delay fault modeling

Consider a set of critical nets, all laying on some circuit path, for which no single vector can activate all nets based on the stuck-at fault model discussed above. Then, the problem of finding the minimum number of vectors to exercise all these critical nets can be reduced to the problem of *finding a robust test for a path delay fault on the particular path*. In general, a delay fault is assumed to cause a defect in the manufactured circuit when the cumulative delay of a combinational logic path exceeds the clock period. The path delay fault model is the most accurate among delay fault models [117], as it can model both lumped and accumulated delays along paths. Under the path delay fault model every fault is represented as a sequence of falling (1→0) or rising (0→1) transitions along a path. The transition initiated at an input is propagated through the path to an output. A path delay test consists of a pair of vectors

(v_1, v_2), where v_1 initializes the path and v_2 launches and propagates the transition through the path to an observable output.

Path delay tests can by categorized as robust or non-robust. A robust test guarantees the detection of the path delay fault in consideration irrespective of the delays on other off-path nets, as it does not allow any hazards to propagate along the path. In this manner, any possible masking due to different timing arrival of transitions is avoided. Hence, a robust test (v_1, v_2) guarantees to exercise all critical nets on a path at the logic “1” value for 50% of the time needed to apply vectors v_1 and v_2. It is often the case that a robust test does not exist. This is mainly attributed to the complex structure of circuits with multiple re-convergent physical paths. In such cases, a relaxed test can be generated (called non-robust test) which allows the propagation of static or dynamic hazards on some of the nets. In the domain of manufacturing test this translates to the possibility of additional circuit delays masking the transition and, hence, invalidating the test. In the context of the work in this thesis, any critical nets asserting transitions with hazards cannot always be considered as exercised. The solution in this case is to isolate such nets and re-target them in the vector generation process (via different paths) so as to exercise them with additional vectors.

	Non-Robust (v_1, v_2)	Robust Test (v_1, v_2)	
Gate Type	Rising or Falling Transition on On-Path Net	Rising Transition on On-Path Net	Falling Transition on On-Path Net
AND / NAND	$(x, 1)$	$(x, 1)$	$(1, 1)$
OR / NOR	$(x, 0)$	$(0, 0)$	$(x, 0)$

Table 7.1: Necessary sensitization conditions on off-input nets for robust and non-robust test generation.

Table 7.1 summarizes the sensitization conditions on a path’s off-inputs for robust and non-robust path delay fault tests. Off-inputs are nets which are inputs to gates on the path but are not nets of the path themselves. For the relaxed non-robust tests, it suffices for v_2 to settle the off-inputs to a steady non-controlling value (“*0*” for OR/NOR gates and “*1*” for AND/NAND gates) and allow any value for v_1 (x). For a robust test, if the on-path net settles to a controlling value (“*1*” for OR/NOR gates and “*0*” for AND/NAND gates) then the off-input nets must assert a stable non-controlling value during both v_1 and v_2; otherwise, the same conditions as with a non-robust test hold. The reader is referred to [117] for a complete discussion on path delay fault test generation.

Fig. 7.9 illustrates a short example of how path delay fault tests can be utilized to exercise critical nets. The first objective is to find a robust test for a path which covers all the critical nets (for simplicity, at this point let us assume that one such physical path exists; this condition

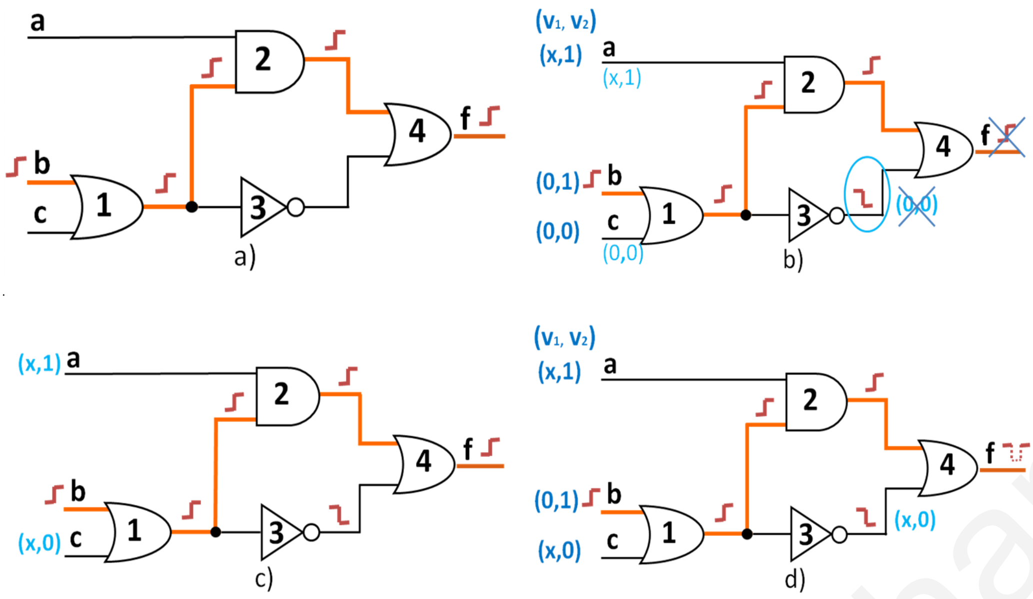


Figure 7.9: Test generation example for robust and non-robust tests.

will be relaxed later in subsection 7.3.3). The targeted path is highlighted in red color and starts from input b, passes through gates *1*, *2*, *4*, and ends at output *f*. We assume, without any loss of generality, that all nets in the path are critical. As a robust test (v_1, v_2) guarantees the propagation of a transition from an input net to an output net through the targeted path, the activation of all critical on-path nets during either the first vector v_1 or the second vector v_2 is also guaranteed.

Test generation begins by asserting a transition at the origin of the path (let this be a rising transition here) and propagates it to the path output based on the types of the gates on the path (see Fig. 7.9.a). This phase is followed by an attempt to justify all off-path nets for all gates on the path to the necessary values, as shown in Fig. 7.9.b. For a robust test, input c is set to a stable 0 value $(0, 0)$, and input a is set to $(x, 1)$, according to the sensitization rules for robust tests given in Table 7.1. However, this does not allow for the off-path input of gate *4* to be set to $(0, 0)$, thus, a robust test is not possible for the particular path and transition type at the origin. A similar issue is created if a falling transition is selected at the path origin. The solution in this case, shown in Fig. 7.9.c, is a non-robust test where the necessary condition for input c and a is $(x, 0)$ and $(x, 1)$, respectively. Moreover, the propagated falling transition at the output of gate 3 can be justified by $(x, 0)$ which is a valid off-input sensitization condition for gate 4. The generated pair of vectors, as shown in Fig. 7.9.d, is $(v_1, v_2) = (x0x, 110)$. In this case, as net f will either assume a static-1 hazard value or a stable 1 value, it will be exercised. A reverse situation (static-0 hazard or stable 0 value) would require that net f is re-targeted by a different vector to ensure that it will be exercised as desired.

7.3.3 Exercise Vector Generation Technique

Activation of critical nets on a single physical path

Assume a circuit netlist C and a set of critical nets N_c. The critical nets are derived based on the NBTI stress model discussed in section 7.3.1 (additional details are also provided in section 7.3.4 where the complete evaluation framework is discussed). Algorithm 6 outlines the major steps in the proposed exercise vector generation methodology. The primary goal is to derive a small number of vectors (V) capable of activating all nets in N_c. A secondary goal is the optimization of each generated vector in terms of unspecified bits as this allows for further compaction of the vector set to be exercised in both dimensions of the vector set (number and size of vectors), by applying the heuristics discussed in section 7.3. This latter goal is achieved by the powerful test generation routines utilized for this work and their particular details are beyond the scope of this thesis.

Algorithm 6 begins by targeting the activation of all critical nets with a single vector based on the relaxed stuck-at model (step 01). If one such vector exists then V will contain a single vector; otherwise, more than 1 vectors are required to exercise all nets in N_c and the approach considers the path delay fault model by first finding a physical path p which contains all critical nets in N_c (step 06). It is assumed at this point that such a path p always exists, however, this condition can be removed as discussed in the next subsection. Consequently, a robust test is targeted for p (step 07). If such a test exists, then the approach guarantees to return a set of 2 vectors in $V = (v_1, v_2)$ able to activate all nets in N_c. As discussed in the previous subsection, for cases where a robust pair of vectors (test) does not exists the conditions of test generation are relaxed to derive a non-robust pair of vectors (step 12). It is assumed that a non-robust test can always be found for path p. Otherwise, p cannot be singly sensitized which means that it cannot affect the timing of the circuit unless delays on other nets, not on p, exist in the circuit. However, as all critical nets are on p this situation cannot occur.

Under the non-robust criterion, some nets of p may assert values with hazards (either static or dynamic) and, therefore, may not be able to guarantee the activation of a critical net. In steps 14-16, the list of critical nets is updated to only contain such problematic nets and the entire approach is repeated but only for these nets. The algorithm terminates when no more critical nets exist in N_c.

Algorithm 6 Proposed exercise vectors generation technique

Inputs: Circuit netlist ***C***, list of critical nets N_c
Outputs: Set of exercise vectors ***V***

```
01. v = generate_stuck_at_activation_vector(C, N_c)
02. if (v ≠ ∅ )
03.    add v to V
04.    return
05. else
06.    p = identify_physical_path(C, N_c)
07.    (v_1, v_2) = generate_robust_test(C, p)
08.    if ((v_1, v_2) ≠ ∅ )
09.       add (v_1, v_2) to V
10.       return
11.    else
12.       (v_1, v_2) = generate_non-robust_test(C, p)
13.    add (v_1, v_2) to V
14.    ∀ net n ∈ N_c
15.       if value of n under (v_1, v_2) ≠ hazard
16.          N_c = N_c - n
17.    if (N_c == ∅ )
18.       return
19.    else
20.       goto step 01
```

Extending to the general case

The proposed exercise vector generation technique can be extended to the general case where critical nets are dispersed over multiple paths in the circuit and do not lay on a single physical path. A heuristic procedure can be used to find the minimum number of paths that can cover all critical nets and then utilize the proposed vector generation procedure to generate vectors for each path. Such a heuristic can be implemented using known graph-theoretic approaches, where the circuit logic is represented as a Directed Acyclic Graph (DAG), such as a modified version of the problem of determining a minimum number of edge-disjoint paths to cover edges in a DAG. In depth-details are abstracted at this point due to space limitations.

7.3.4 Experimental Evaluation and Results

Evaluation Framework

We model the combinational part of any design as a DAG which makes the technique transparent to the microarchitecure of a device. As a proof of concept for evaluating the efficacy of PRITEXT, we have used the synthesizable Verilog RTL model of a superscalar processor core available from FabScalar [118] open source toolset. For the purposes of this work, we consider a core with a frontend width (Fetch, Decode, Rename, Dispatch, Issue, Rename Register, Execute, Writeback) of 4, Load/Store Queue of size 32 and 128-entry Re-Order Buffer. More details regarding the evaluation framework used are out of the scope of this thesis and can be found in [119].

NBTI Critical Timing Paths

NBTI degradation is not uniform across all the paths in a device because of different timing delays and duty cycles along the gates. In short, workload stresses each net and path differently leading to highly skewed duty cycles across some of the timing paths. Hence only the paths which do not have enough slack to overcome the degraded switching delay are highly prone to timing failures due to NBTI stress. All the paths having less than 10% slack are considered to be NBTI critical. We synthesized the processor core for a clock frequency of 500 MHz using Synopsys Design Compiler with a 45 nm TSMC standard cell library.

Workload Characterization of Critical Paths

Since NBTI is highly sensitive to duty cycle of transistors that lie along a timing path, we performed gate level simulations of the synthesized netlist using six workloads (`bzip2`, `gap`, `gzip`, `mcf`, `vortex` and `parser`) from the SPEC CPU2000 benchmark suite to obtain their duty cycles. Using the waveform dumps, we calculated the duty cycle of all the internal nets of critical timing paths. More details can be found in [119].

Modern processors are often quiescent for a major fraction during their operation while waiting for events such as input/output access, or off-chip DRAM access to complete. This provides opportunities to enable the exercise mode to combat NBTI effects whenever the processor is idle. Based on the real-time statistics collected for various processors in several server clusters, we have considered a conservative ratio of 1:1 for active to standby processor state durations in the experimental evaluation. A higher standby time triggers improved opportunities to balance duty cycles along critical logic path since each exercise vector has higher fraction of operational time.

Deterministic Algorithm for Vector Generation

The extracted combinational logic cone of the critical nets in the Load/Store unit of the experimental platform [118] consists of 1,426 primary inputs, 15,194 internal nodes, and 63 unique critical nets which all lay on a single path. Using Algorithm 6 we have obtained 9 exercise vectors (designated by $V1$-$V9$ with each vector having 1,426 bits) which together guarantee to cumulatively exercise all of the 63 critical nets at least once during application of the exercise phase. Fig. 7.10 illustrates the coverage achieved by each of the 9 vectors in terms of new critical nets in the Load/Store unit path. Vector $V1$ exercises 35 unique critical nets while vector $V2$ exercises 12 additional nets, and 34 critical nets in total. Vectors $V1$, $V2$, $V3$ and $V4$ collectively exercises 50 (35+9+2+4) out of the total 63 critical nets, covering approximately 80% of them. This observance, together with the heuristic described in section 7.3.2 motivated us to compute lifetime improvement under two distinct vector sets: a complete set comprising all 9 vectors (dubbed "Set9"), and a second set consisting only of the first 4 vectors (dubbed "Set4"). Both of these vector sets are considerably smaller than the set derived from the algorithm in [78] which gives a set of 14 (10) exercise vectors for 100% (80%) coverage of the critical nets for this design.

In order to minimize the hardware overheads in PRITEXT technique, we performed vector compaction to minimize the newly introduced MUXes and on-chip ROM that stores the generated vectors. The storage capacity of this ROM equals to the product of the vector count and the width of each exercise vector, which in turn corresponds to the 1,426 primary inputs of the extracted logic. Out of 1,426 possible MUX locations corresponding to the inputs under vector set Set4, 1,224 of those contain "don't care" logical values spreading across the 4 generated vectors. This high number of unspecified bit values is attributed to the ability of the proposed vector generation technique to produce vectors that exercise a large number of critical nets per vector, while bit-setting only necessary inputs at a time. Moreover, 81 of the inputs always remain at logic value '0,' while 83 inputs always remain at logic value '1' in each vector. Hence, the final ROM width is only 38 (1426-1224-81-83) bits, leading to a total ROM size of 4×38 bits (i.e., 4 vectors affect 38 inputs). Only 202 (38+81+83) MUXes are necessary, with 164 (81+83) of them presenting a constant value as input during exercise mode. A similar analysis for Set9 leads to a ROM size of 9×53 bits, totaling 357 MUXes (53+171+133 bits). Hence, the width optimization heuristic achieves ROM size and MUX overhead reduction of 97.3% (96.3%) and 86.8% (74.9%) for Set4 (Set9), respectively, compared to the initial unoptimized vector set with a ROM size of 9×1426 bits and 1426 MUXes. The overall hardware overhead due to PRITEXT technique for Set9 was well within 0.5% of the total area of the reference processor.

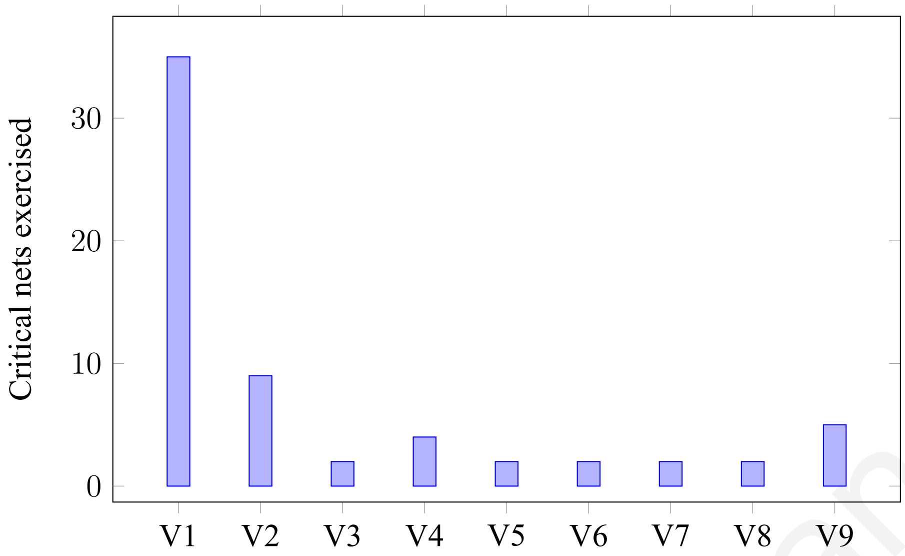


Figure 7.10: Coverage of unique critical nets per vector.

Balanced Duty Cycles

Fig. 7.11 illustrates the duty cycle distribution of critical nets when the PRITEXT technique is applied to the superscalar processor under consideration. The fraction of critical nets having skewed duty cycles (i.e., within bin 0.9) has dropped from 38% to 13% on average, as compared to the reference system with no lifetime extending support, highlighting PRITEXT's efficacy.

Lifetime Improvement

Fig. 7.12 gives lifetime improvements using PRITEXT and the two generated vector sets. Set4 achieves nearly the same improvement as in Set9, yet it demands smaller hardware overheads. This trade-off between the size of the vector set (leading to repercussions such as hardware overhead, dynamic power consumption) and lifetime improvement can be leveraged by designers to trade-off design constraints. The lifetime improvement technique explored in this work achieves an average of 4.99× lifetime improvement over a reference system using deterministic vectors, while a maximum improvement of 13.91× is observed for the highly memory intensive `mcf` workload.

7.4 Chapter Summary

This Chapter explores proactive reliability techniques designed to decelerate the effects of aging in the critical components like core processors or NoC of CMPs. Section 7.2 investigates critical path models for NBTI-induced wear due to the stresses cause by realist workloads and

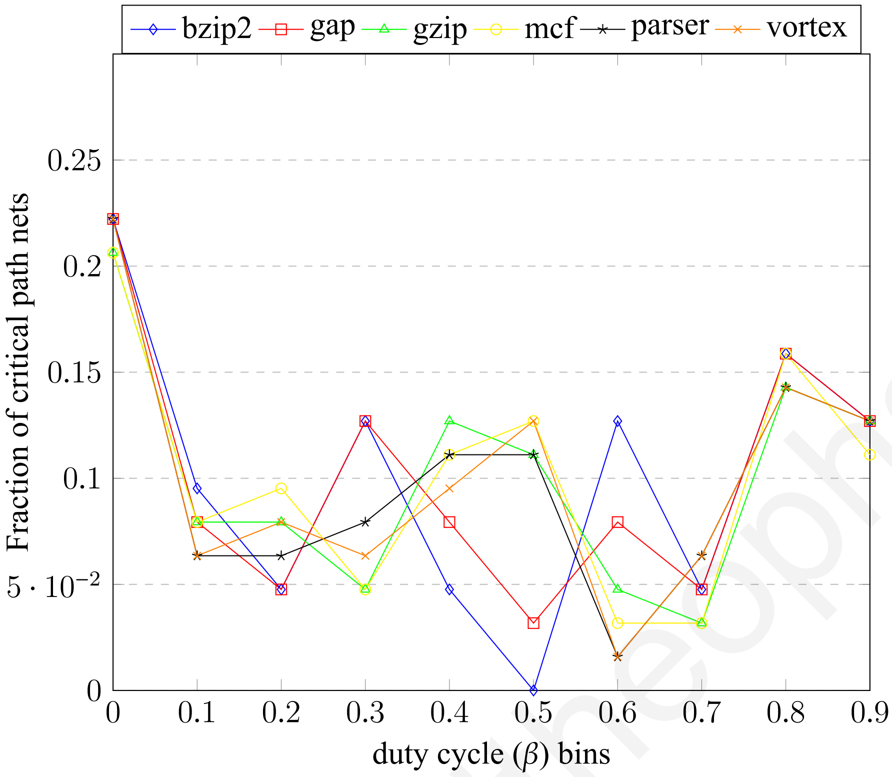


Figure 7.11: Duty cycle distribution under PRITEXT.

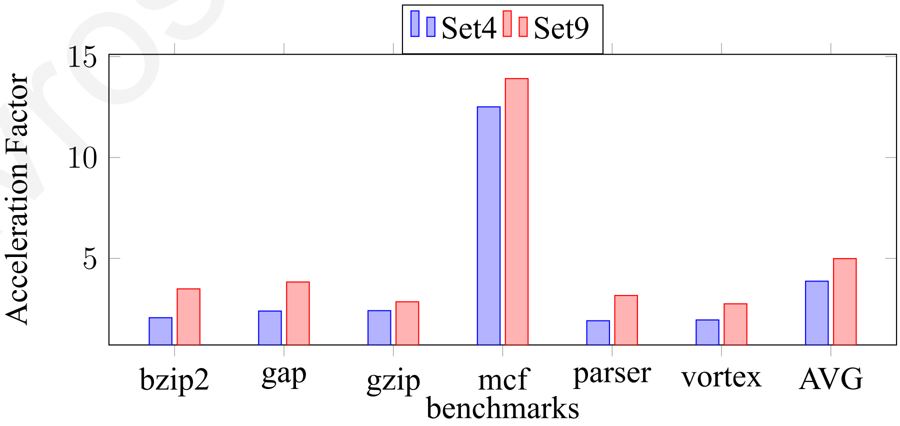


Figure 7.12: Lifetime Improvement with PRITEXT

applied them into an NoC interconnect microarchitecture. Section 7.3 leverages path delay test principles to derive near-ideal vectors while simultaneously providing a deterministic algorithm to generate exercise vectors under circumstances where such tests do not exist. The efficiency of this technique is evaluated on a reference superscalar processor with propitious lifetime improvement results and negligible hardware overheads.

Chapter 8

Conclusions and Future Work

The present thesis explores the new potentials from multi-core era and the new opportunities created for fundamental test automation processes. Due to their computationally intensive nature they can significantly benefit from those developments. However, those opportunities cannot be effortlessly achieved because the performance gain by the use of multi-cores is strongly depended on the software algorithms used and the corresponding implementations. Due to their complexity and unpredicted dynamic execution test automation algorithms cannot rely on automatic parallelization tools because those tools can lead to local optimal solutions compromising the quality of results.

A detailed analysis for the impact of partitioning in parallel fault simulation and test generation processes is presented. Fault and test partitioning affect both the scalability and test inflation of the parallel solutions, thus, simple static partitioning is not recommended for a scalable and well balance parallel solution. Moreover, the thesis explores parallel fault simulation and parallel test generation for shared memory on-chip multiprocessors homogeneous architectures. Both parallel, techniques are able to maintain their scalability as the number of processing cores utilized increases and target to avoid workload duplication and test inflation problems.

In addition, the parallel test pattern generation methodology is appropriately extended to generate high quality multiple-detect (n-detect) test sets. Such test sets, known as n-detect test sets (each fault targeted n times), have been experimentally shown to improve the defect coverage at the expense of an increased test set size. Appropriate experimentation using the investigated parallel n-detect test generation method shows that test sets retain all good properties of a parallel test generation methodology in a more beneficial extent.

Lastly, the thesis investigates reliability techniques where deterministically generated exercise vectors are utilized to prolong CMPs lifetime. It is shown that exercise vectors generated in design phase, stored and utilized by CMP during idle times can significantly prolong its lifetime. A novel vector generation technique based on ATPG compaction concepts is explored to generate compact vector sets that significantly improve the lifetime of an inter-core NoC router and a superscalar processor.

8.1 Future Work

There are several directions to be investigated for future work. First, the investigation findings of the parallel fault simulation and test generation methods can be applied into different non-enumerative fault models such as path delay and bridging. The scalability, the efficient workload balancing and utilization of shared resources are only some of the optimization directions of the explored test automation processes which can be utilized targeting different fault models. Another directions can be the better identification of a-priori compatibilities between faults (know as pseudo-compatibility). Grouping identified compatible faults together can support higher fault dropping and a better guidance of the parallel solutions. Other directions involve investigation of various high performance computing infrastructures other than on-chip homogeneous CMPs like architectures that support dynamic frequency scaling, MPSoCs, super computers farms and hybrid cloud solutions. Will be very interesting to investigate non-homogeneous architectures and adjust the proposed methodologies accordingly.

A popular research topic is the acceleration of functional simulation for processor based designs. Functional simulation is performed after the test bench and design code is created and is an iterative process which may require multiple simulations to achieve the desired end functionality of the design. Since verification of designs is taking an increasing proportion of the design and test cycle of a processor (or embedded processor) functional simulation typically is a time-consuming task. Would be very interesting to investigate the performance of the parallelization concepts proposed in the thesis at functional simulation domain.

Simulation of system-level description languages is another area where the methodologies proposed in this thesis can be utilized to speed-up the execution time. Modern system platforms often consist of multiple processing elements including general purpose CPUs, digital signal processors, dedicated hardware accelerators and many more. The complexity and challenges to design and validate those systems are increasing rapidly. Parallel fault simulation concepts can be utilized for better design some of the existing parallel solutions, which are mostly based on thread-level parallelism. Similar parallelization concepts are applied also in this area since shared memory is the main communication and synchronization mean.

Besides the research directions already explored there are various others research areas where concepts and findings of this thesis can be utilized such as design for testability, programmable build-in self-test, embedded systems, algorithms for autonomous vehicles, hardware security and trusted integrated circuit designs etc.

References

[1] M. Gorev, R. Ubar, and S. Devadze, “Fault simulation with parallel exact critical path tracing in multiple core environment,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 1180–1185.

[2] J. C. Ku, R. H. Huang, L. Y. Lin, and C. H. Wen, “Suppressing test inflation in shared-memory parallel automatic test pattern generation,” in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 664–669.

[3] X. Cai, P. Wohl, and D. Martin, “Fault sharing in a copy-on-write based atpg system,” in *Test Conference (ITC), 2014 IEEE International*. IEEE, 2014, pp. 1–8.

[4] K.-W. Yeh, J.-L. Huang, H.-J. Chao, and L.-T. Wang, “A circular pipeline processing based deterministic parallel test pattern generator,” in *Test Conference (ITC), 2013 IEEE International*. IEEE, 2013, pp. 1–8.

[5] X. Cai, P. Wohl, J. A. Waicukauski, and P. Notiyath, “Highly efficient parallel atpg based on shared memory,” in *Test Conference (ITC), 2010 IEEE International*. IEEE, 2010, pp. 1–7.

[6] K. Olukotun, L. Hammond, and J. Laudon, “Chip multiprocessor architecture: techniques to improve throughput and latency,” *Synthesis Lectures on Computer Architecture*, vol. 2, no. 1, pp. 1–145, 2007.

[7] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI test principles and architectures: design for testability*. Academic Press, 2006.

[8] P. Banerjee, *Parallel algorithms for VLSI computer-aided design*. Prentice-Hall, Inc., 1994.

[9] P. Agrawal, V. D. Agrawal, K.-T. Cheng, and R. Tutundjian, “Fault simulation in a pipelined multiprocessor system,” in *Test Conference, 1989. Proceedings. Meeting the Tests of Time., International*. IEEE, 1989, pp. 727–734.

[10] N. Ishiura, M. Ito, and S. Yajima, “Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 9, no. 8, pp. 868–875, 1990.

[11] V. Narayanan and V. Pitchumani, “Fault simulation on massively parallel simd machines algorithms, implementations and results,” *Journal of Electronic Testing*, vol. 3, no. 1, pp. 79–92, 1992.

[12] M. B. Amin and B. Vinnakota, “Data parallel fault simulation,” *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol. 7, no. 2, pp. 183–190, 1999.

[13] T. Nagumo, M. Nagai, T. Nishida, M. Miyoshi, and S. Miyamoto, "Vfsim: Vectorized fault simulator using a reduction technique excluding temporarily unobservable faults," in *Proceedings of the 31st annual Design Automation Conference*. ACM, 1994, pp. 510–515.

[14] H. K. Lee and D. S. Ha, "Hope: An efficient parallel fault simulator for synchronous sequential circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048–1058, 1996.

[15] Y. Ali, Y. Yamato, T. Yoneda, K. Hatayama, and M. Inoue, "Parallel path delay fault simulation for multi/many-core processors with simd units," in *Test Symposium (ATS), 2014 IEEE 23rd Asian*. IEEE, 2014, pp. 292–297.

[16] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault simulation in a distributed environment," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1988, pp. 686–691.

[17] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on proofs," in *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*. IEEE, 1995, pp. 616–621.

[18] E. Schneider, M. A. Kochte, S. Holst, X. Wen, and H.-J. Wunderlich, "Gpu-accelerated simulation of small delay faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 5, pp. 829–841, 2017.

[19] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with gp-gpus," in *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009, pp. 557–562.

[20] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin, "Efficient fault simulation on many-core processors," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 380–385.

[21] M. Li and M. S. Hsiao, "3-d parallel fault simulation with gpgpu," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 10, pp. 1545–1555, 2011.

[22] K. Gulati and S. P. Khatri, "Fault table computation on gpus," *Journal of Electronic Testing*, vol. 26, no. 2, pp. 195–209, 2010.

[23] H. Li, D. Xu, Y. Han, K.-T. Cheng, and X. Li, "ngfsim: A gpu-based fault simulator for 1-to-n detection and its applications," in *Test Conference (ITC), 2010 IEEE International*. IEEE, 2010, pp. 1–10.

[24] E. Schneider, S. Holst, M. A. Kochte, X. Wen, and H.-J. Wunderlich, "Gpu-accelerated small delay fault simulation," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1174–1179.

[25] A. Czutro, I. Polian, M. Lewis, P. Engelke, S. M. Reddy, and B. Becker, "Thread-parallel integrated test pattern generator utilizing satisfiability analysis," *International Journal of Parallel Programming*, vol. 38, no. 3, pp. 185–202, 2010.

[26] S. Hadjitheophanous, S. N. Neophytou, and M. K. Michael, "Utilizing shared memory multi-cores to speed-up the atpg process," in *Test Symposium (ETS), 2016 21th IEEE European*. IEEE, 2016, pp. 1–6.

[27] M. Abramovici, P. R. Menon, and D. T. Miller, “Critical path tracing-an alternative to fault simulation,” in *Proceedings of the 20th Design Automation Conference*. IEEE Press, 1983, pp. 214–220.

[28] S. Hadjitheophanous, S. N. Neophytou, and M. K. Michael, “Scalable parallel fault simulation for shared-memory multiprocessor systems,” in *VLSI Test Symposium (VTS), 2016 IEEE 34th*. IEEE, 2016, pp. 1–6.

[29] S. Hadjitheophanous, S. Neophytou, and M. K. Michael, “Exploiting shared-memory to steer scalability of fault simulation using multicore systemst,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (accepted for publication)*, 2018.

[30] K. J. Antreich and M. H. Schulz, “Accelerated fault simulation and fault grading in combinational circuits,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 6, no. 5, pp. 704–712, 1987.

[31] D. Harel, R. Sheng, and J. Udell, “Efficient single fault propagation in combinational circuits,” *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[32] F. Maamari and J. Rajski, “A method of fault simulation based on stem regions,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 2, pp. 212–220, 1990.

[33] L. Wu and D. Walker, “A fast algorithm for critical path tracing in vlsi digital circuits,” in *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*. IEEE, 2005, pp. 178–186.

[34] R. Ubar, S. Devadze, J. Raik, and A. Jutman, “Ultra fast parallel fault analysis on structurally synthesized bdds,” in *Test Symposium, 2007. ETS’07. 12th IEEE European*. IEEE, 2007, pp. 131–136.

[35] R. Ubar, S. Devadze, J. Raik, J. Jutman, and Jutman, “Parallel x-fault simulation with critical path tracing technique,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 879–884.

[36] K. Scheibler, D. Erb, and B. Becker, “Improving test pattern generation in presence of unknown values beyond restricted symbolic logic,” in *Test Symposium (ETS), 2015 20th IEEE European*. IEEE, 2015, pp. 1–6.

[37] S. Eggersglub, K. Schmitz, R. Krenz-Baath, and R. Drechsler, “Optimization-based multiple target test generation for highly compacted test sets,” in *Test Symposium (ETS), 2014 19th IEEE European*. IEEE, 2014, pp. 1–6.

[38] I. Pomeranz, “Generation of compact multi-cycle diagnostic test sets,” in *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 2013, pp. 1–1.

[39] S. Patil and P. Banerjee, “Fault partitioning issues in an integrated parallel test generation/fault simulation environment,” in *Test Conference, 1989. Proceedings. Meeting the Tests of Time., International*. IEEE, 1989, pp. 718–726.

[40] J. M. Wolf, L. M. Kaufman, R. H. Klenke, J. H. Aylor, and R. Waxman, “An analysis of fault partitioned parallel test generation,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 15, no. 5, pp. 517–534, 1996.

[41] K.-Y. Liao, C.-Y. Chang, and J. C.-M. Li, "A parallel test pattern generation algorithm to meet multiple quality objectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 11, pp. 1767–1772, 2011.

[42] K.-W. Yeh, M.-F. Wu, and J.-L. Huang, "A low communication overhead and load balanced parallel atpg with improved static fault partition method," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2009, pp. 362–371.

[43] X. Cai and P. Wohl, "A distributed-multicore hybrid atpg system," in *Test Conference (ITC), 2013 IEEE International*. IEEE, 2013, pp. 1–7.

[44] K.-Y. Liao, S.-C. Hsu, and J. C.-M. Li, "Gpu-based n-detect transition fault atpg," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 28.

[45] K.-Y. Liao, P.-J. Chen, A.-F. Lin, J. C.-M. Li, M. S. Hsiao, and L.-T. Wang, "Gpu-based timing-aware test generation for small delay defects," in *Test Symposium (ETS), 2014 19th IEEE European*. IEEE, 2014, pp. 1–2.

[46] S. Neophytou and M. K. Michael, "Two new methods for accurate test set relaxation via test set replacement," in *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*. IEEE, 2008, pp. 827–831.

[47] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. M. Reddy, "Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 12, pp. 1496–1504, 1995.

[48] J.-S. Chang and C.-S. Lin, "Test set compaction for combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 11, pp. 1370–1378, 1995.

[49] M. H. Schulz, E. Trischler, and T. M. Sarfert, "Socrates: A highly efficient automatic test pattern generation system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 1, pp. 126–137, 1988.

[50] Y. Matsunaga, "A test pattern compaction method using sat-based fault grouping," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 99, no. 12, pp. 2302–2309, 2016.

[51] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. ACM, 1998, pp. 283–289.

[52] S. Remersaro, J. Rajski, S. M. Reddy, and I. Pomeranz, "A scalable method for the generation of small test sets," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09*. IEEE, 2009, pp. 1136–1141.

[53] Z. Wang and D. Walker, "Dynamic compaction for high quality delay test," in *VLSI Test Symposium, 2008. VTS 2008. 26th IEEE*. IEEE, 2008, pp. 243–248.

[54] B. Krishnamurthy and S. B. Akers, "On the complexity of estimating the size of a test set," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 750–753, 1984.

[55] B. Benware, C. Schuermyer, S. Ranganathan, R. Madge, P. Krishnamurthy, N. Tamarapalli, K.-H. Tsai, and J. Rajski, “Impact of multiple-detect test patterns on product quality,” in *null*. Citeseer, 2003, p. 1031.

[56] E. J. McCluskey and C.-W. Tseng, “Stuck-fault tests vs. actual defects,” in *Test Conference, 2000. Proceedings. International*. IEEE, 2000, pp. 336–342.

[57] I. Pomeranz and S. M. Reddy, “Forming n-detection test sets without test generation,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 2, p. 18, 2007.

[58] S. Venkataraman, S. Sivaraj, E. Amyeen, S. Lee, A. Ojha, and R. Guo, “An experimental study of n-detect scan atpg patterns on a processor,” in *VLSI Test Symposium, 2004. Proceedings. 22nd IEEE*. IEEE, 2004, pp. 23–28.

[59] C.-W. Tseng and E. J. McCluskey, “Multiple-output propagation transition fault test,” in *Test Conference, 2001. Proceedings. International*. IEEE, 2001, pp. 358–366.

[60] X. Kavousianos and K. Chakrabarty, “Generation of compact test sets with high defect coverage,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1130–1135.

[61] J. Geuzebroek, E. J. Marinissen, A. Majhi, A. Glowatz, and F. Hapke, “Embedded multi-detect atpg and its effect on the detection of unmodeled defects,” in *Test Conference, 2007. ITC 2007. IEEE International*. IEEE, 2007, pp. 1–10.

[62] S. Biswas, P. Srikanth, R. Jha, S. Mukhopadhyay, A. Patra, and D. Sarkar, “On-line testing of digital circuits for n-detect and bridging fault models,” in *Test Symposium, 2005. Proceedings. 14th Asian*. IEEE, 2005, pp. 88–93.

[63] B. Vaidya and M. B. Tahoori, “Delay test generation with all reachable output propagation and multiple excitations,” in *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*. IEEE, 2005, pp. 380–388.

[64] K. R. Kantipudi and V. D. Agrawal, “On the size and generation of minimal n-detection tests,” in *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*. IEEE, 2006, pp. 6–pp.

[65] F. Oboril and M. B. 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*. IEEE, 2012, pp. 1–12.

[66] L. Lai, V. Chandra, R. Aitken, and P. Gupta, “Bti-gater: An aging-resilient clock gating methodology,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 4, no. 2, pp. 180–189, 2014.

[67] D. R. Bild, R. P. Dick, and G. E. Bok, “Static nbti reduction using internal node control,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 4, p. 45, 2012.

[68] U. R. Karpuzcu, B. Greskamp, and J. Torrellas, “The bubblewrap many-core: popping cores for sequential acceleration,” in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 447–458.

[69] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The nbti-aware processor," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 2007, pp. 85–96.

[70] E. Gunadi, A. A. Sinkar, N. S. Kim, and M. H. Lipasti, "Combating aging with the colt duty cycle equalizer," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 103–114.

[71] S. Gupta and S. S. Sapatnekar, "Employing circadian rhythms to enhance power and reliability," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 3, p. 38, 2013.

[72] F. Oboril and M. B. Tahoori, "Aging-aware design of microprocessor instruction pipelines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 5, pp. 704–716, 2014.

[73] F. Oboril, F. Firouzi, S. Kiamehr, and M. Tahoori, "Reducing nbti-induced processor wearout by exploiting the timing slack of instructions," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2012, pp. 443–452.

[74] F. Firouzi, S. Kiamehr, and M. B. Tahoori, "Nbti mitigation by optimized nop assignment and insertion," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012, pp. 218–223.

[75] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 276.

[76] J. Shin, V. Zyuban, Z. Hu, J. A. Rivers, and P. Bose, "A framework for architecture-level lifetime reliability modeling," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 534–543.

[77] M. Jenihhin, G. Squillero, T. S. Copetti, V. Tihhomirov, S. Kostin, M. Gaudesi, F. Vargas, J. Raik, M. S. Reorda, L. B. Poehls *et al.*, "Identification and rejuvenation of nbti-critical logic paths in nanoscale circuits," *Journal of Electronic Testing*, vol. 32, no. 3, pp. 273–289, 2016.

[78] H. Kim, S. B. K. Boga, A. Vitkovskiy, S. Hadjitheophanous, P. V. Gratz, V. Soteriou, and M. K. Michael, "Use it or lose it: Proactive, deterministic longevity in future chip multiprocessors," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 4, p. 65, 2015.

[79] K. Bhardwaj, K. Chakraborty, and S. Roy, "An milp-based aging-aware routing algorithm for nocs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012, pp. 326–331.

[80] X. Fu, T. Li, and J. A. Fortes, "Architecting reliable multi-core network-on-chip for small scale processing technology," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 111–120.

[81] D. I. August, "Parallelizing sequential code," *IEEE Micro*, vol. 32, no. 4, pp. 6–7, 2012.

[82] A. Darte, Y. Robert, and F. Vivien, *Scheduling and automatic Parallelization*. Springer Science & Business Media, 2012.

[83] C. Gil, J. Ortega, J. L. Bernier, and M. D. Gil, “Parallel test pattern generation using circuit partitioning in a shared-memory multiprocessor,” in *International Workshop on Applied Parallel Computing*. Springer, 1998, pp. 167–171.

[84] S. Neophytou, S. Hadjitheophanous, and M. K. Michael, “On the impact of fault list partitioning in parallel implementations for dynamic test compaction considering multicore systems,” in *Design and Test Symposium (IDT), 2013 8th International*. IEEE, 2013, pp. 1–6.

[85] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.

[86] S. N. Neophytou and M. K. Michael, “Test set generation with a large number of unspecified bits using static and dynamic techniques,” *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 301–316, 2010.

[87] S. R. Nassif, “Modeling and analysis of manufacturing variations,” in *Custom Integrated Circuits, 2001, IEEE Conference on*. IEEE, 2001, pp. 223–228.

[88] H.-J. Wunderlich, *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*. Springer Science & Business Media, 2009, vol. 43.

[89] Z. Navabi, “Digital system test and testable design,” *E-ISBN*, pp. 97 814 419–97 875 485, 2011.

[90] D. B. Armstrong, “A deductive method for simulating faults in logic circuits,” *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 464–471, 1972.

[91] E. G. Ulrich and T. Baker, “The concurrent simulation of nearly identical digital networks,” in *Papers on Twenty-five years of electronic design automation*. ACM, 1988, pp. 318–323.

[92] W.-T. Cheng and M.-L. Yu, “Differential fault simulation-a fast method using minimal memory,” in *Design Automation, 1989. 26th Conference on*. IEEE, 1989, pp. 424–428.

[93] C.-Y. Lee, “Representation of switching circuits by binary-decision programs,” *Bell Labs Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[94] O. H. Ibarra and S. K. Sahni, “Polynomially complete fault detection problems,” *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 242–249, 1975.

[95] J. P. Roth, “Diagnosis of automata failures: A calculus and a method,” *IBM journal of Research and Development*, vol. 10, no. 4, pp. 278–291, 1966.

[96] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE transactions on Computers*, no. 3, pp. 215–222, 1981.

[97] H. Fujiwara, “Fan: A fanout-oriented test pattern generation algorithm,” in *IEEE International Symposium on Circuits and Systems*, 1985, pp. 671–674.

[98] I. Dear, C. Dislis, A. P. Ambler, and J. Dick, “Economic effects in design and test,” *IEEE Design & Test of Computers*, vol. 8, no. 4, pp. 64–77, 1991.

[99] S. N. Neophytou and M. K. Michael, “Test pattern generation of relaxed n-detect test sets,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 410–423, 2012.

[100] W. J. Dally and B. Towles, “Route packets, not wires: On-chip interconnection networks,” in *Design Automation Conference, 2001. Proceedings.* IEEE, 2001, pp. 684–689.

[101] S. Nassif, K. Bernstein, D. J. Frank, A. Gattiker, W. Haensch, B. L. Ji, E. Nowak, D. Pearson, and N. J. Rohrer, “High performance cmos variability in the 65nm regime and beyond,” in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*. IEEE, 2007, pp. 569–571.

[102] K. J. Kuhn, “Reducing variation in advanced logic technologies: Approaches to process and design for manufacturability of nanoscale cmos,” in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*. IEEE, 2007, pp. 471–474.

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

[104] J. Blome, S. Feng, S. Gupta, and S. Mahlke, “Self-calibrating online wearout detection,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 109–122.

[105] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai, “Detecting emerging wearout faults,” in *Proc. of Workshop on SELSE*, 2007.

[106] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, “Architectural core salvaging in a multi-core processor for hard-error tolerance,” in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 93–104.

[107] X. Li, J. Qin, and J. B. Bernstein, “Compact modeling of mosfet wearout mechanisms for circuit-reliability simulation,” *IEEE Transactions on Device and Materials Reliability*, vol. 8, no. 1, pp. 98–121, 2008.

[108] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, “Daemonguard: O/s-assisted selective software-based self-testing for multi-core systems,” in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 45–51.

[109] Z. Zhang, A. Greiner, and S. Taktak, “A reconfigurable routing algorithm for a fault-tolerant 2d-mesh network-on-chip,” in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 441–446.

[110] T. Schonwald, J. Zimmermann, O. Bringmann, and W. Rosenstiel, “Fully adaptive fault-tolerant routing algorithm for network-on-chip architectures,” in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 527–534.

[111] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design*. Computer science press New York, 1990, vol. 2.

[112] M. Bushnell and V. D. Agrawal, “Essentials of electronic testing for digital, memory and mixed-signal vlsi circuits,” vol. 17. Springer, 2000.

[113] D. U. Becker, “Efficient microarchitecture for network-on-chip routers,” Ph.D. dissertation, Stanford University, 2012.

[114] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[115] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: characterization and architectural implications,” in *the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[116] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula, “Predictive modeling of the nbti effect for reliable design,” in *Custom Integrated Circuits Conference, 2006. CICC’06. IEEE*. IEEE, 2006, pp. 189–192.

[117] A. Krstic and K.-T. T. Cheng, *Delay fault testing for VLSI circuits*. Springer Science & Business Media, 2012, vol. 14.

[118] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, “Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template,” in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 11–22.

[119] P. M. Reddy, S. Hadjitheophanous, V. Soteriou, P. V. Gratz, and M. K. Michael, “Minimal exercise vector generation for reliability improvement,” in *On-Line Testing and Robust System Design (IOLTS), 2017 IEEE 23rd International Symposium on*. IEEE, 2017, pp. 113–119.