NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Architectures for Dependable Modern Microprocessors

Nikolaos A. Foutris

ATHENS

February 2015

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αρχιτεκτονικές Αξιόπιστης Λειτουργίας Σύγχρονων Μικροεπεξεργαστών

Νικόλαος Α. Φουτρής

ΑΘΗΝΑ

Φεβρουάριος 2015

PhD THESIS

Architectures for Dependable Modern Microprocessors

Nikolaos A. Foutris

ADVISOR: Dimitris Gizopoulos, Associate Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:
Dimitris Gizopoulos, Associate Professor UoA
Antonis Paschalis, Professor UoA
Mihalis Psarakis, Assistant Professor UniPi

SEVEN-MEMBER EXAMINATION COMMITTEE

(Signature) (Signature)

Dimitris Gizopoulos,
Associate Professor UoA

Antonis Paschalis,
Professor UoA

(Signature) (Signature)

Mihalis Psarakis,
Assistant Professor UniPi

Manolis Katevenis,
Professor UoC

(Signature) (Signature)

Dionisios Pnevmatikatos,
Professor TUC

Nectarios Koziris,
Professor NTUA

(Signature)

Nikolaos Bellas,
Associate Professor UTH

Examination Date 27/2/2015

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αρχιτεκτονικές Αξιόπιστης Λειτουργίας Σύγχρονων Μικροεπεξεργαστών

Νικόλαος Α. Φουτρής

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Δημήτρης Γκιζόπουλος, Αν. Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:
Δημήτρης Γκιζόπουλος, Αναπληρωτής Καθηγητής ΕΚΠΑ
Αντώνης Πασχάλης, Καθηγητής ΕΚΠΑ
Μιχάλης Ψαράκης, Επίκουρος Καθηγητής ΠΑ.ΠΕΙ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

(Υπογραφή)

Δημήτρης Γκιζόπουλος,
Αν. Καθηγητής ΕΚΠΑ

(Υπογραφή)

Αντώνης Πασχάλης,
Καθηγητής ΕΚΠΑ

(Υπογραφή)

Μιχάλης Ψαράκης,
Επ. Καθηγητής Πα.Πει

(Υπογραφή)

Μανόλης Κατεβαίνης,
Καθηγητής Παν. Κρήτης

(Υπογραφή)

Διονύσιος Πνευματικάτος,
Καθηγητής Πολυτεχνείο Κρήτης

(Υπογραφή)

Νεκτάριος Κοζύρης,
Καθηγητής ΕΜΠ

Νικόλαος Μπέλλας,
Αν. Καθηγητής Παν. Θεσσαλίας

Ημερομηνία εξέτασης 27/2/2015

ABSTRACT

The evolution of semiconductor technology and computer architecture has radically transformed our world throughout the last decades. However, the combination of technology scaling and extreme chip integration, along with the compelling requirement to diminish the time-to-market window, has rendered microprocessors more prone to design bugs and hardware faults. The goal of this thesis is to provide solutions to the validation challenges posed from the microprocessor products throughout the life-cycle of a chip.

Microprocessor validation is grouped into the following categories, based on where they intervene in a microprocessor's lifecycle: (a) *silicon debug*: the first hardware prototypes are exhaustively validated, (b) *manufacturing testing*: the final quality control during massive production and before chip shipping, and (c) *in-field verification*: runtime error detection techniques to guarantee correct operation in the field. The contributions of this thesis are the following:

- *Silicon debug*: We propose the employment of *deconfigurable* microprocessor architectures along with a technique to generate *self-checking* random test programs to: (a) avoid the time- and resource-consuming simulation step, (b) triage the redundant debug sessions, and thus to accelerate silicon debug [8] [10].
- *Manufacturing testing*: We propose a self-test optimization strategy for multithreaded, multicore microprocessors to: (a) speedup test program execution time, (b) enhance the fault coverage of hard errors, and thus to make manufacturing testing more efficient [11].
- *In-field verification*: We measure the effect of permanent faults performance components. Then, we propose a set of low-cost hardware-based mechanisms for the detection, diagnosis and performance recovery in the front-end speculative structures [2] [5] [6].

The share of silicon debug in the overall microprocessor chips development cycle is rapidly expanding [2]. The validation step that detects the vast majority of design bugs is the one that stresses the silicon prototypes by applying huge numbers of random test programs. Despite its bug detection capability, this step is constrained by the extreme computing needs for random test program simulation. Moreover, another major bottleneck and source of "noise" of this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behaviour adds long delays in the debug flow since each failing random program must be separately examined, although it does not usually bring new debug information. This thesis addresses both challenges of silicon debug. A self-checking methodology is proposed for generating random test programs (exploiting the ISA diversity property) that detect bugs by comparing the results of equivalent instructions combined with a technique to triage the failing test programs into categories with common failure modes. The proposed framework: (a) improves bug detection efficiency, (b) reduces the redundant debug session, and thus overall accelerates silicon debug.

When a sufficient level of design bugs coverage is reached the microprocessor design enters the production stage, where a final quality control is performed to detect manufacturing defects in massive production. Functional self-testing forms an integral

part of manufacturing test flow due to (a) at-speed testing: test application and response collection are performed at the processor's actual speed, enabling screening of delay defects that aren't detectable at lower frequencies; and (b) non-intrusive nature: does not add any extra hardware. Multithreaded (MT) SBST methodology proposes a novel self-test optimization strategy for multithreaded, multicore microprocessor architectures. The proposed self-test program execution optimization aims to: (a) take maximum advantage of the available execution parallelism provided by multiple threads and multiple cores, (b) preserve the high fault coverage that single-thread execution provides for the processor components; and (c) enhance the fault coverage of the thread-specific control logic. MT-SBST methodology significantly speeds up self-test time, while at the same time it improves the overall fault coverage.

The combination of design complexity, shrinking time-to-market windows, and wear-out effects increases the failure probability of modern designs in the field and leads microprocessor manufacturers to integrate numerous in-field verification mechanisms. Trends such as low-voltage operation and process scaling are expected to significantly increase the rate of faults experienced by silicon. Their impact on a core's non-cache SRAM structures, such as the speculation components, has not been accurately quantified. Faults in these structures will not affect correctness, but can cause severe performance degradation and variability among otherwise identical cores. We first classify and quantify the impact of permanent faults in the performance components of modern microprocessors. Then, we propose low-cost microarchitectural mechanisms that exploit the self-verification property of speculative structures to achieve performance recovery.

Modern microprocessors implement extremely complex architectures, making the validation process a major challenge for the semiconductor industry. This thesis introduces various novel methodologies to address the validation challenges posed throughout the life-cycle of a chip. The proposed techniques make the validation process more efficient and are easily applicable to the existing industrial flow.

SUBJECT AREA: Computer Architecture

KEYWORDS: Dependability, Silicon debug, Testing, Errors, Bugs

ΠΕΡΙΛΗΨΗ

Η ραγδαία εξέλιξη των ολοκληρωμένων κυκλωμάτων, από την πλευρά της τεχνολογίας υλικού αλλά και της αρχιτεκτονικής υπολογιστών έχουν εκτινάξει το κόστος, σε ανθρωποώρες και υπολογιστική ισχύ, που απαιτείται για την διασφάλιση της ορθής λειτουργίας ενός επεξεργαστή. Σε συνδυασμό με τους αυστηρούς χρονικούς περιορισμούς που υπάρχουν για την ανάπτυξη ολοκληρωμένων κυκλωμάτων η επαλήθευση της ορθής λειτουργίας των επεξεργαστών καθίσταται μία εξαιρετικά απαιτητική και ακριβή διαδικασία. Ως εκ τούτου, η ανάπτυξη μεθόδων που θα επιταχύνουν την διαδικασία της επαλήθευσης της ορθής λειτουργίας των επεξεργαστών είναι επιβεβλημένη.

Με κριτήριο το στάδιο του κύκλου ζωής ενός επεξεργαστή, από την στιγμή κατασκευής των πρωτοτύπων και έπειτα, οι τεχνικές ελέγχου ορθής λειτουργίας διακρίνονται στις ακόλουθες κατηγορίες:

- *Silicon Debug*: Τα πρωτότυπα ολοκληρωμένα κυκλώματα ελέγχονται εξονυχιστικά για τον εντοπισμό σχεδιαστικών και κατασκευαστικών σφαλμάτων του υλικού. Βασικές προκλήσεις του silicon debug που χρίζουν αντιμετώπισης είναι οι εξής: (1) η αποφυγή της προσομοίωσης των τυχαίων προγραμμάτων. Η διαδικασία της προσομοίωσης είναι αρκετές τάξεις μεγέθους πιο αργή από την εκτέλεση στο πραγματικό υλικό, με αποτέλεσμα να περιορίζει το πλήθος των σεναρίων ελέγχου ορθής λειτουργίας που μπορούν να δοκιμαστούν. (2) Η ανάπτυξη τεχνικών που θα ομαδοποιούν τα προγράμματα ελέγχου σφαλμάτων, τα οποία εντοπίζουν το ίδιο σφάλμα, είναι ιδιαίτερα σημαντική για την ομαλή και εντός προθεσμιών ολοκλήρωση του ελέγχου ορθής λειτουργίας ενός ολοκληρωμένου κυκλώματος. (3) Η ανίχνευση και αντιμετώπιση σφαλμάτων υλικού που αποκρύπτουν τον εντοπισμό νέων [8] [10].
- *Manufacturing Testing*: Αποτελεί τον τελικό ποιοτικό έλεγχο που διενεργείται κατά την μαζική παραγωγή των ολοκληρωμένων κυκλωμάτων. Βασικές προκλήσεις του manufacturing testing που χρίζουν αντιμετώπισης είναι οι εξής: (1) Ο εντοπισμός κατασκευαστικών σφαλμάτων ή αστοχιών υλικού. (2) Γρήγορο εντοπισμό σφαλμάτων καθώς η χρονική μετατόπιση της εισαγωγής ενός προϊόντος-επεξεργαστή στην αγορά έχει καταστροφικές συνέπειες για το ίδιο το προϊόν και την κατασκευάστρια εταιρία [11].
- *In-field verification*: Περιλαμβάνει τεχνικές, οι οποίες διασφαλίζουν την λειτουργία του επεξεργαστή σύμφωνα με τις προδιαγραφές του. Προκλήσεις που χρίζουν αντιμετώπισης είναι οι εξής: (1) Ανάλυση της επίδρασης στην απόδοση των σφαλμάτων υλικού που είτε έχουν ξεφύγει από τα προηγούμενα στάδια κατασκευής του κυκλώματος ή λόγω της φθοράς του κυκλώματος. (2) Ανάπτυξη μηχανισμών για την ανίχνευση και ανοχή σφαλμάτων υλικού [2] [5] [6].

Ο σκοπός της διδακτορικής διατριβής είναι να προταθούν λύσεις για την αντιμετώπιση των προκλήσεις που υπάρχουν σε κάθε ένα από τα προαναφερθέντα στάδια του κύκλου ζωής ενός επεξεργαστή. Οι προτεινόμενες τεχνικές συμβάλλουν στην βελτίωση της αποτελεσματικότητας της διαδικασίας επαλήθευσης ορθής λειτουργίας καθώς και καθίσταται δυνατή η άμεση υιοθέτησή τους από την βιομηχανία.

Η αναγκαιότητα χρήσης του silicon debug στο κύκλο ζωής ενός επεξεργαστή συνεχώς αυξάνεται. Αρχικά, προτάθηκε μία μεθοδολογία για την επιτάχυνση της διαδικασίας

εντοπισμού σφαλμάτων, κατά την φάση του ελέγχου των πρωτοτύπων κυκλωμάτων, μέσω της κατασκευής λογισμικού αυτό-δοκιμής. Η κεντρική ιδέα αυτής της μεθόδου έγκειται στην αξιοποίηση της έμφυτης ποικιλομορφίας των αρχιτεκτονικών συνόλου εντολών, δηλαδή της ιδιότητάς τους να υλοποιούν μία λειτουργία με περισσότερους από ένα τρόπους (ή διαφορετικά με περισσότερες από μία διαφορετικές εντολές). Επιπρόσθετα, προτάθηκε μία μέθοδο για τον αυτόματο εντοπισμό τυχαίων προγραμμάτων που δεν περιέχουν νέα -χρήσιμη- πληροφορία σχετικά με την γενεσιουργό αίτια ενός σφάλματος για τους μηχανικούς. Ο προτεινόμενος μηχανισμός βασίστηκε στην λειτουργία της από-διαμόρφωσης, δηλαδή την δυνατότητα να απενεργοποιούνται τμήματα της λογικής του κυκλώματος χωρίς να επηρεάζεται η λειτουργικότητα του επεξεργαστή. Τα προγράμματα αυτοδοκιμής ομαδοποιούνται σε κατηγορίες σύμφωνα με την ακολουθία των τμημάτων λογικής που αποδιαμορφώθηκαν από το κύκλωμα, έτσι ώστε να εκτελεστούν σωστά. Ως εκ τούτου, οι μηχανικοί αποσφαλμάτωσης του κυκλώματος μελετούν μόνο ένα πρόγραμμα από κάθε κατηγορία για να εντοπίσουν την γενεσιουργό αίτια του σφάλματος. Τα πειραματικά αποτελέσματα πιστοποίησαν την δυνατότητα των προτεινόμενων μεθόδων στον (α) στον εντοπισμό σχεδιαστικών σφαλμάτων και (β) στην βελτίωση της διαδικασίας αποσφαλμάτωση του κυκλώματος και κατά συνέπεια στην επιτάχυνση silicon debug.

Η μεθοδολογία Multithreaded (MT) SBST προτείνει μία καινοτόμο μέθοδο για την βελτιστοποίηση και επιτάχυνση της στρατηγικής έλεγχου ορθής λειτουργίας των πολυνηματικών και πολυπύρηνων επεξεργαστών μέσω της χρήση λογισμικού αυτοδοκιμής. Το λογισμικό αυτοδοκιμής αποτελεί αναπόσπαστο τμήμα του manufacturing testing καθώς (α) επιτρέπει την εκτέλεση του λογισμικού αυτοδοκιμής στην συχνότητα λειτουργίας του επεξεργαστή και (β) δεν προσθέτει νέο υλικό. Στην ερευνητική εργασία αυτή προτάθηκε μία μέθοδος που αποσκοπεί (α) να εκμεταλλευτεί στο μέγιστο τις δυνατότητες παραλληλισμού που παρέχουν τα πολλαπλά νήματα και πυρήνες του επεξεργαστή, (β) να διατηρήσει σε υψηλά επίπεδα το ποσοστό κάλυψης ελαττωμάτων υλικού (άνω του 90%) και (γ) να βελτιστοποιήσει το ποσοστό κάλυψης ελαττωμάτων υλικού στα τμήματα λογικής όπου σχετίζονται με την πολυνηματική και πολυπύρηνη εκτέλεση. Η μεθοδολογία MT-SBST επιταχύνει σημαντικά τη διαδικασία ελέγχου ορθής λειτουργία ενός επεξεργαστή, ενώ παράλληλα βελτιώνει συνολικά το ποσοστό κάλυψης ελαττωμάτων.

Η συνεχώς αυξανόμενη πιθανότητα εμφάνισης σφαλμάτων υλικού κατά την διάρκεια λειτουργία ενός επεξεργαστή (in-field verification), οδήγησε στην εισαγωγή μηχανισμών επαλήθευσης της ορθής τους λειτουργίας. Στα πλαίσια της διδακτορικής διατριβής, αναλύθηκε σε βάθος η επίδραση που έχουν τα μόνιμα σφάλματα (μονά και σε ορισμένες περιπτώσεις πολλαπλά) στους εξής μηχανισμούς: (α) μηχανισμός πρόβλεψης διακλάδωσης, και (β) μηχανισμός εκ των προτέρων προσκόμισης δεδομένων, στην λειτουργία του συστήματος. Επιπρόσθετα, προτάθηκαν τεχνικές για την ανίχνευση και ανοχή μόνιμων σφαλμάτων υλικού στους μηχανισμούς πρόβλεψης διακλάδωσης. Ειδικότερα, η ανίχνευση επιτυγχάνεται μέσω της αξιοποίησης της αυτόματης διόρθωσης των μηχανισμών πρόβλεψης διακλάδωσης, ενώ η ανοχή μέσω της αναδιαμόρφωσης του υλικού. Τα πρωτότυπα πειραματικά αποτελέσματα αυτών των εργασιών κατέδειξαν με εμφατικό τρόπο την αρνητική επίπτωση των σφαλμάτων υλικού στην απόδοση ενός επεξεργαστή καθώς και της αποτελεσματικότητας των προτεινόμενων μηχανισμών ανίχνευσης και ανοχής σφαλμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Αξιοπιστία, Σφάλματα Υλικού, Σχεδιαστικά σφάλματα

στην Ευθυμία

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Dimitris Gizopoulos, who steered me into the research path. His guidance and continuous encouragement have been invaluable to the conduction of my dissertation. The interaction with Professor Dimitris Gizopoulos provided me with several skills and knowledge that would be very helpful in my future career and life, and I am infinitely graceful to him for that.

I also like to thank the member of my advising committee: Professors Mihalis Psarakis and Antonis Paschalis. Since my undergraduate studies, I have really enjoyed working with Professor Mihalis Psarakis and I truly appreciate his support and advises. Although, I haven't worked personally with Professor Antonis Paschalis, I have always respected him. Also, I would like to thank them for their positive feedback that made the completion of this dissertation possible. Moreover, I would like to thank all the people that I had the opportunity to collaborate with throughout these years. Finally, I am deeply graceful to my friends for their continuous assistance and support, will always remain in my mind.

Finally, I would like to thank my family, who always stood by me. I am waiting for the day I will be able to give back to them even a small percentage of what they so generously offered to me all these years. I am blessed for having such a wonderful family. Last but not least, completion of this work would not be possible without the support of Efthymia Kastanidou who always encouraged me in the times of failure and celebrated my successes. I am grateful for her support and I would like to dedicate my dissertation to her. Thanks a lot Efthymia.

LIST OF PUBLICATIONS

[1] N.Foutris, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Versatile Architecture-Level Faults Injection Framework for Reliability Evaluation: A First Report”, In IEEE International On-Line Testing Symposium (IOLTS), 2014.

[2] N.Foutris, D.Gizopoulos, A.Chatzidimitriou, J.Kalamatianos, V.Sridharan, “Performance Assessment of Data Prefetchers in High Error Rate Technologies”, In IEEE Workshop on Silicon Errors in Logic – System Effect (SELSE-10), 2014.

[3] M.Kaliorakis, M.Psarakis, N.Foutris, D.Gizopoulos, “Accelerated Online Error Detection in Many-core Microprocessor Architectures”, in IEEE International VLSI Test Symposium (VTS), 2014.

[4] M.Kaliorakis, M.Psarakis, N.Foutris, D.Gizopoulos, “Parallelizing Online Error Detection in Many-core Microprocessor Architectures”, Joint Euro-TM/Median Workshop on Dependable Multicore and Transactional Memory Systems (DMTM), 2014.

[5] N.Foutris, D.Gizopoulos, J.Kalamatianos, V.Sridharan, “Assessing the Impact of Hard Faults in Performance Components of Modern Microprocessors”, in IEEE International Conference on Computer Design (ICCD), 2013.

[6] N.Foutris, D.Gizopoulos, J.Kalamatianos, V.Sridharan, “Measuring the Performance Impact of Permanent Faults in Modern Microprocessor Architectures”, in IEEE International On-Line Testing Symposium, 2013.

[7] M.Kaliorakis, N.Foutris, D.Gizopoulos, M.Psarakis, “Online Error Detection in Multiprocessor Chips: A Test Scheduling Study”, in IEEE International On-Line Testing Symposium, 2013.

[8] N.Foutris, D.Gizopoulos, X.Vera, A.Gonzalez, “Deconfigurable Microprocessor Architectures for Silicon Debug Acceleration”, in ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013.

[9] T.Ramirez, E.Herrero, N.Axelos, J.Carretero, N.Foutris, D.Sanchez, X.Vera, “Mitigating Lower Layer Failures with Adaptive System Reconfiguration”, in IEEE International Symposium on Mixed Design of Integrated Circuits and Systems (MIXDES), 2012.

[10] N.Foutris, D.Gizopoulos, M.Psarakis, X.Vera, A.Gonzalez, “Accelerating Microprocessor Silicon Debug by Exposing ISA Diversity”, in ACM/IEEE International Symposium on Microarchitecture (MICRO), 2011.

[11] N.Foutris, M.Psarakis, D.Gizopoulos, A.Apostolakis, X.Vera, A.Gonzalez, “MT-SBST: Self-Test Optimization in Multithreaded Multicore Architectures”, in IEEE International Test Conference (ITC), 2010.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Στη σημερινή πραγματικότητα, τον 21ο αιώνα, κάθε άνθρωπος περιβάλλεται από υπολογιστικά συστήματα, ζει με αυτά, τα χρησιμοποιεί, είναι ιδιωτικά ή δημόσια αγαθά, είναι μικρά, ίσως μεγάλα, μεγαλύτερα, εν γένει διάφορα υπολογιστικά συστήματα που κατακλύζουν τη ζωή ενός ατόμου. Θα μπορούσε κανείς να αναφερθεί στις «ηλεκτρονικές συσκευές». Ακόμη πιο ενδιαφέρουσες όμως είναι οι «έξυπνες ηλεκτρονικές συσκευές». Το κοινό χαρακτηριστικό όλων αυτών είναι η ύπαρξη ενός ενσωματωμένου «εγκεφάλου». Στον «εγκέφαλο» αυτό αποδίδονται τα χαρακτηριστικά που διέπουν τα υπολογιστικά συστήματα. Με άλλα λόγια, οι διαρκώς αυξανόμενες ανάγκες των καταναλωτών οδηγούν τις εξελίξεις, ενώ παράλληλα διαμορφώνουν τις απαιτήσεις στη βιομηχανία. Μία έξυπνη λοιπόν συσκευή είναι αυτή που μεταξύ άλλων είναι δυνατός ο έλεγχός της από το χρήστη εξ ’αποστάσεως, μπορεί να διαλειτουργήσει με άλλες ηλεκτρονικές συσκευές, να προσαρμόζει τη λειτουργία της αυτόματα, να εξοικονομεί ενέργεια και αναμφίβολα να είναι ασφαλής. Αυτά είναι μόνο ορισμένα, ένα πολύ μικρό υποσύνολο χαρακτηριστικών και δυνατοτήτων που δύναται να φέρουν οι ηλεκτρονικές συσκευές. Σε κάθε περίπτωση, ο απώτερος σκοπός ανάπτυξης των συσκευών αυτών είναι η διευκόλυνση της ζωής των καταναλωτών, με σεβασμό στο τρίπτυχο κόστος-αποδοτικότητα-αποτελεσματικότητα.

Τα σύγχρονα λοιπόν τεχνολογικά επιτεύγματα, όπως ενδεικτικά αναφέρθηκαν παραπάνω κατέστησαν αναγκαία, ή ενδεχομένως λειτούργησαν και ως καταλύτης για τη ραγδαία ανάπτυξη των ολοκληρωμένων κυκλωμάτων, τη δομική δηλαδή μονάδα, τα κύτταρα των υπολογιστικών συστημάτων. Το αντίτιμο αυτής της εξέλιξης από την πλευρά της τεχνολογίας υλικού αλλά και της αρχιτεκτονικής υπολογιστών έχει εκτινάξει το κόστος, σε ανθρωποώρες και υπολογιστική ισχύ, που απαιτείται για τη διασφάλιση της ορθής λειτουργίας ενός επεξεργαστή. Σε συνδυασμό με τους αυστηρούς χρονικούς περιορισμούς που υπάρχουν για την ανάπτυξη ολοκληρωμένων κυκλωμάτων η επαλήθευση της ορθής λειτουργίας των επεξεργαστών καθίσταται μία εξαιρετικά απαιτητική, χρονοβόρα και ακριβή διαδικασία. Ως εκ τούτου, η ανάπτυξη μεθόδων που θα επιταχύνουν τη διαδικασία της επαλήθευσης της ορθής λειτουργίας των επεξεργαστών είναι επιβεβλημένη και απολύτως εναρμονισμένη με τις ανάγκες του επιχειρηματικού και καταναλωτικού περιβάλλοντος.

Με κριτήριο το στάδιο του κύκλου ζωής ενός επεξεργαστή, από την στιγμή κατασκευής των πρωτοτύπων και έπειτα, οι τεχνικές ελέγχου ορθής λειτουργίας διακρίνονται στις ακόλουθες κατηγορίες:

- Silicon Debug: Τα πρωτότυπα ολοκληρωμένα κυκλώματα ελέγχονται εξονυχιστικά για τον εντοπισμό σχεδιαστικών και κατασκευαστικών σφαλμάτων του υλικού. Βασικές προκλήσεις του silicon debug που χρίζουν αντιμετώπισης είναι οι εξής: (1) η αποφυγή της προσομοίωσης των τυχαίων προγραμμάτων. Η διαδικασία της προσομοίωσης είναι αρκετές τάξεις μεγέθους πιο αργή από την εκτέλεση στο πραγματικό υλικό, με αποτέλεσμα να περιορίζει το πλήθος των σεναρίων ελέγχου ορθής λειτουργίας που μπορούν να δοκιμαστούν. (2) Η ανάπτυξη τεχνικών που θα ομαδοποιούν τα προγράμματα ελέγχου σφαλμάτων, τα οποία εντοπίζουν το ίδιο σφάλμα, είναι ιδιαίτερα σημαντική για την ομαλή

και εντός προθεσμιών ολοκλήρωση του ελέγχου ορθής λειτουργίας ενός ολοκληρωμένου κυκλώματος. (3) Η ανίχνευση και αντιμετώπιση σφαλμάτων υλικού που αποκρύπτουν τον εντοπισμό νέων [8] [10].

- Manufacturing Testing: Αποτελεί τον τελικό ποιοτικό έλεγχο που διενεργείται κατά την μαζική παραγωγή των ολοκληρωμένων κυκλωμάτων. Βασικές προκλήσεις του manufacturing testing που χρίζουν αντιμετώπισης είναι οι εξής: (1) Ο εντοπισμός κατασκευαστικών σφαλμάτων ή αστοχιών υλικού. (2) Γρήγορο εντοπισμό σφαλμάτων καθώς η χρονική μετατόπιση της εισαγωγής ενός προϊόντος-επεξεργαστή στην αγορά έχει καταστροφικές συνέπειες για το ίδιο το προϊόν και την κατασκευάστρια εταιρία [11].
- In-field verification: Περιλαμβάνει τεχνικές, οι οποίες διασφαλίζουν την λειτουργία του επεξεργαστή σύμφωνα με τις προδιαγραφές του. Προκλήσεις που χρίζουν αντιμετώπισης είναι οι εξής: (1) Ανάλυση της επίδρασης στην απόδοση των σφαλμάτων υλικού που είτε έχουν ξεφύγει από τα προηγούμενα στάδια κατασκευής του κυκλώματος ή λόγω της φθοράς του κυκλώματος. (2) Ανάπτυξη μηχανισμών για την ανίχνευση και ανοχή σφαλμάτων υλικού [2] [5] [6].

Ο σκοπός αυτής της διδακτορικής διατριβής είναι να προταθούν τεκμηριωμένες λύσεις για την αντιμετώπιση των προκλήσεων που ενυπάρχουν σε κάθε ένα από τα προαναφερθέντα στάδια του κύκλου ζωής ενός επεξεργαστή. Ειδικότερα, οι προτεινόμενες τεχνικές συμβάλλουν στη βελτίωση της αποτελεσματικότητας της διαδικασίας επαλήθευσης ορθής λειτουργίας ενός οποιουδήποτε επεξεργαστή. Εκτιμάται, ότι ο στόχος αυτός επιτυγχάνεται με την ολοκλήρωση αυτής της πρωτότυπης διατριβής και επιπλέον ο έμπειρος αναγνώστης θα διαπιστώσει ένα ακόμη σημαντικό επίτευγμα: πρόκειται για ένα σύνολο αποτελεσματικών τεχνικών οι οποίες είναι δυνατό να υιοθετηθούν και να εφαρμοστούν άμεσα από τη βιομηχανία. Με άλλα λόγια, μία σημαντική πτυχή της διδακτορικής αυτής διατριβής είναι η έμφαση στη διευθέτηση πραγματικών προβλημάτων, με ρεαλιστικές, πρακτικές και αποτελεσματικές λύσεις.

Το silicon debug αποτελεί ένα αναπόσπαστο και συνεχώς αυξανόμενης σημασίας και διάρκειας τμήμα του κύκλου ζωής ενός επεξεργαστή. Ενδεικτική είναι η συνεχώς αυξανόμενη πίεση που ασκείται στους μηχανικούς υλικού για να ολοκληρώσουν την αποσφαλμάτωση ενός επεξεργαστή. Στο πλαίσιο αυτό θα πρέπει να ληφθεί επιπλέον υπόψη η εξαιρετικά μεγάλη πολυπλοκότητα των σύγχρονων επεξεργαστών, σε συνδυασμό με τα στενά χρονικά περιθώρια που τίθενται. Επιπρόσθετα, οι τεχνικές ελέγχου ορθής λειτουργίας ενός επεξεργαστή πριν την παραγωγή των πρωτοτύπων κυκλωμάτων (pre-silicon verification) βασίζονται αποκλειστικά σε εργαλεία προσομοίωσης. Ωστόσο, παρόλο την αποτελεσματικότητα των τεχνικών αυτών και την μεγάλη υπολογιστική ισχύ που χρησιμοποιείται για την επίτευξη του στόχου, η χαμηλή απόδοση των τεχνικών προσομοίωσης αποτελεί βασική συστημική αδυναμία. Πιο συγκεκριμένα, ένας τεράστιος αριθμός προγραμμάτων ελέγχου ορθής λειτουργίας εκτελούνται αδιάκοπα στο πρωτότυπο μοντέλο του επεξεργαστή με σκοπό την ανίχνευση οποιουδήποτε σχεδιαστικού ή κατασκευαστικού σφάλματος. Στην συνέχεια, κάθε φορά που εντοπίζεται ένα σφάλμα, η διαδικασία αποσφαλμάτωσης ξεκινά. Ο στόχος της διαδικασίας αυτής είναι ο προσδιορισμός της πηγής του σφάλματος καθώς και η μόνιμη διόρθωσή του. Είναι προφανές ότι η ανάπτυξη τεχνικών οι οποίες θα ανιχνεύσουν εγκαίρως και θα διορθώσουν τα σχεδιαστικά σφάλματα

πριν την μαζική παραγωγή του μοντέλου του επεξεργαστή αποτελεί καθοριστικό παράγοντα για την αποτυχία ή μη ενός ολοκληρωμένου κυκλώματος.

Η μαζική εκτέλεση προγραμμάτων ορθής λειτουργίας με τυχαιότητα αποτελεί ένα από τους πιο αποδοτικούς τρόπους εντοπισμού σχεδιαστικών σφαλμάτων. Παγκοσμίως, οι μεγαλύτερες εταιρίες ολοκληρωμένων κυκλωμάτων έχουν κατασκευάσει εργαλεία αυτόματης παραγωγής τυχαίων προγραμμάτων αποσκοπώντας στην ταχύτερη και πληρέστερη κάλυψη όλων των πιθανών σεναρίων ελέγχου ενός επεξεργαστή. Ωστόσο, παρόλο την ευρεία υιοθέτηση αυτής της μεθόδου, υπάρχουν ορισμένες εν γένει αδυναμίες. Αυτές είναι οι ακόλουθες:

1. Ο εντοπισμός ενός σχεδιαστικού ή κατασκευαστικού σφάλματος προϋποθέτει την εκτέλεση του προγράμματος ελέγχου ορθής λειτουργίας με τυχαιότητα αφενός στον πρωτότυπο μοντέλο του επεξεργαστή και αφετέρου σε έναν αρχιτεκτονικό προσομοιωτή (ο οποίος εξάγει το σωστό-αναμενόμενο αποτέλεσμα). Στην συνέχεια, τα αποτελέσματα από τις δύο πηγές συγκρίνονται έτσι ώστε να προσδιοριστεί ο εντοπισμός ή μη ενός σφάλματος (διαφορά στα αποτελέσματα συνεπάγεται τον εντοπισμό σφάλματος). Συνεπώς, η διαδικασία του silicon debug περιορίζεται από την απόδοση του προσομοιωτή, ο οποίος σε κάθε περίπτωση είναι πολλές τάξεις μεγέθους πιο αργός από ένα πραγματικό μηχάνημα.
2. Μία ακόμη πηγή καθυστέρησης της ολοκλήρωσης του silicon debug αποτελεί ο μεγάλος αριθμός από προγράμματα ελέγχου ορθής λειτουργίας που εντοπίζουν το ίδιο σχεδιαστικό ή κατασκευαστικό σφάλμα, τα οποία δεν μεταφέρουν κάποια νέα πληροφορία για την κατάσταση του επεξεργαστή. Παρόλα αυτά οι μηχανικοί υλικού πρέπει να μελετήσουν κάθε ένα από αυτά ξεχωριστά για να εντοπίσουν την γενεσιουργό αιτία του σφάλματος καθυστερώντας την όλη διαδικασία.

Στο πλαίσιο αυτής της διδακτορικής διατριβής προτάθηκε μία μεθοδολογία για την επιτάχυνση της διαδικασίας εντοπισμού σφαλμάτων, κατά τη φάση ελέγχου των πρωτοτύπων κυκλωμάτων, μέσω της κατασκευής λογισμικού αυτό-δοκιμής. Η κεντρική ιδέα αυτής της μεθόδου έγκειται στην αξιοποίηση της έμφυτης ποικιλομορφίας των αρχιτεκτονικών συνόλου εντολών, δηλαδή της ιδιότητάς τους να υλοποιούν μία λειτουργία με περισσότερους από ένα τρόπους (ή διαφορετικά με περισσότερες από μία διαφορετικές εντολές). Αυτή η ερευνητική εργασία αποσκοπεί στην επιτάχυνση της διαδικασίας εντοπισμού σφαλμάτων, κατά τη φάση του ελέγχου των πρωτοτύπων κυκλωμάτων. Τα πειραματικά αποτελέσματα (τα οποία εξήχθησαν στην πιο διαδεδομένη αρχιτεκτονική υπολογιστών, x86) πιστοποίησαν τη δυνατότητα της προτεινόμενης μεθόδου, αφενός στον εντοπισμό (η προτεινόμενη μέθοδος έχει υψηλότερα ποσοστά κάλυψης σφαλμάτων σε σύγκριση με υπάρχουσες τεχνικές) και στην αποφυγή σφαλμάτων που αποκρύπτουν τον εντοπισμό νέων και αφετέρου στην επιτάχυνση της διαδικασίας επαλήθευσης ενός επεξεργαστή (η προτεινόμενη μέθοδος είναι πολλές τάξεις μεγέθους γρηγορότερη έναντι της βασικής διαδικασίας που υιοθετείται από τις εταιρείες). Επιπρόσθετα, προτάθηκε μία μέθοδος για τον αυτόματο εντοπισμό τυχαίων προγραμμάτων που δεν περιέχουν νέα -χρήσιμη- πληροφορία σχετικά με την γενεσιουργό αίτια ενός σφάλματος για τους μηχανικούς. Ο προτεινόμενος μηχανισμός βασίστηκε στην λειτουργία της αποδιαμόρφωσης (deconfiguration), δηλαδή της δυνατότητας να

απενεργοποιούνται τμήματα της λογικής του κυκλώματος χωρίς να επηρεάζεται η λειτουργικότητα του επεξεργαστή. Οι σύγχρονοι επεξεργαστές ενσωματώνουν πολλά τμήματα πλεονάζουσας λογικής τα οποία δεν συμβάλλουν στην υλοποίηση των βασικών λειτουργιών του αλλά βελτιώνουν την απόδοση. Παράλληλα, αξιοποιώντας τη δυνατότητα των προγραμμάτων αυτό-δοκιμής για την ανίχνευση σφαλμάτων κατά τη διάρκεια λειτουργία του κυκλώματος, ο προτεινόμενος μηχανισμός αποδιαμορφώνει σταδιακά τα πλεονάζοντα τμήματα της λογικής έως ότου το πρόγραμμα δεν εντοπίσει κάποιο σφάλμα (δηλαδή, το τμήμα της λογικής που περιέχει το σφάλμα έχει απενεργοποιηθεί, με αποτέλεσμα το πρόγραμμα να μην το εντοπίζει). Στη συνεχεία, τα προγράμματα αυτό-δοκιμής ομαδοποιούνται σε κατηγορίες σύμφωνα με την ακολουθία των τμημάτων λογικής που αποδιαμορφώθηκαν από το κύκλωμα, έτσι ώστε να εκτελεστούν σωστά (δηλαδή, τα προγράμματα που μέσω τις ίδιας ακολουθίας αποδιαμόρφωσης «έκρυψαν» το σχεδιαστικό σφάλμα ανήκουν στην ίδια κατηγορία). Ως εκ τούτου, οι μηχανικοί αποσφαλμάτωσης του κυκλώματος μελετούν μόνο ένα πρόγραμμα από κάθε κατηγορία για να εντοπίσουν την γενεσιουργό αίτια του σφάλματος. Τα πειραματικά αποτελέσματα (στην πιο διαδεδομένη αρχιτεκτονική υπολογιστών x86-64) πιστοποίησαν τη δυνατότητα της προτεινόμενης μεθόδου, αφενός στην ομαδοποίηση των προγραμμάτων αυτό-δοκιμής που δεν εμπεριέχουν νέα –χρήσιμη- πληροφορία για την αποσφαλμάτωση του κυκλώματος και αφετέρου στην επιτάχυνση της διαδικασίας επαλήθευσης ενός επεξεργαστή. Ως εκ τούτου, οι προτεινόμενες μέθοδοι πετυχαίνουν: (α) τον εντοπισμό σχεδιαστικών σφαλμάτων και κατασκευαστικών σφαλμάτων και (β) τη βελτίωση της διαδικασίας αποσφαλμάτωση του κυκλώματος και κατά συνέπεια στην επιτάχυνση silicon debug.

Αφού ολοκληρωθεί ο εξονυχιστικός έλεγχος των πρωτότυπων μοντέλων ενός ολοκληρωμένου κυκλώματος, ακολουθεί το στάδιο του manufacturing testing. Το manufacturing testing αποτελεί αναπόσπαστο κομμάτι της σχεδιαστικής αλυσίδας κατά το οποίο εκτελείται ο τελευταίος ποιοτικός έλεγχος του ολοκληρωμένου κυκλώματος κατά την μαζική παραγωγή του επεξεργαστή. Ο στόχος του manufacturing testing είναι ο εντοπισμός οποιουδήποτε κατασκευαστικού σφάλματος το οποίο μπορεί να θέσει σε κίνδυνο την ορθή λειτουργία του επεξεργαστή.

Η βιομηχανία ολοκληρωμένων κυκλωμάτων έχει προσανατολιστεί στην ανάπτυξη πολυπύρηνων και πολυνηματικών επεξεργαστών, οι οποίοι αν και λειτουργούν σε χαμηλότερες συχνότητες παρέχουν υψηλή υπολογιστική ισχύ καθώς εκμεταλλεύονται τα πολλαπλά νήματα και τους πολλούς πυρήνες για τον παραλληλισμό την εκτέλεσης ενός προγράμματος. Παράλληλα, όλες οι τεχνικές που έχουν επινοηθεί για την αντιμετώπιση των προβλημάτων αξιοπιστίας των σύγχρονων μικροεπεξεργαστών πρέπει να ακολουθήσουν την μετάβαση από την εποχή των υπολογιστικών συστημάτων με μονό πυρήνα σε πολλαπλών. Ειδικότερα, οι μελλοντικές τεχνικές ελέγχου ορθής λειτουργίας πρέπει να αξιοποιήσουν τον έμφυτο παραλληλισμό των σύγχρονων επεξεργαστών, έτσι ώστε να μειώσουν τη συνολική διάρκεια των τεχνικών ελέγχου ορθής λειτουργίας, βελτιώνοντας το χρόνο διάθεσης στην αγορά, αλλά χωρίς να υποβαθμιστεί η αποτελεσματικότητα τους σε σχέση με το ποσοστό κάλυψης σφαλμάτων.

Στο πλαίσιο αυτής της διδακτορικής διατριβής προτάθηκε η μεθοδολογία Multithreaded (MT) Software-Based Self-Testing (SBST). Η μεθοδολογία αυτή αποσκοπεί στην βελτιστοποίηση και επιτάχυνση της στρατηγικής έλεγχου ορθής λειτουργίας των πολυνηματικών και πολυπύρηνων επεξεργαστών μέσω της

χρήσης λογισμικού αυτό-δοκιμής (functional self-testing). Η υιοθέτηση του λογισμικού αυτό-δοκιμής αποτελεί αναπόσπαστο τμήμα του manufacturing testing καθώς: (α) επιτρέπει την εκτέλεση του λογισμικού αυτοδοκιμής στη συχνότητα λειτουργίας του επεξεργαστή και (β) της μη-παρεμβατικής συμπεριφοράς στη λειτουργία του (δεν προσθέτει νέο υλικό). Πιο συγκεκριμένα, ο έλεγχος ορθής λειτουργίας με τη χρήση λογισμικού αυτό-δοκιμής εκτελείται ως εξής: ένα πρόγραμμα εφαρμόζει ένα σύνολο από δεδομένα εισόδου (τα οποία ενεργοποιούν κάθε πιθανή κατάσταση του κυκλώματος) και στη συνέχεια συλλέγει τα αποτελέσματα εξόδου (από την κύρια μνήμη του συστήματος) τα οποία και συγκρίνει με τα σωστά-αναμενόμενα έτσι ώστε να διαπιστωθεί εάν έχει εντοπιστεί κάποιο κατασκευαστικό σφάλμα. Στην περίπτωση εντοπισμού ενός κατασκευαστικού σφάλματος το συγκεκριμένο προϊόν αποσύρεται από την παραγωγή. Στην ερευνητική αυτή εργασία προτάθηκε μία μεθοδολογία που αποσκοπεί στα εξής:

1. Να αξιοποιήσει στο μέγιστο τις δυνατότητες παραλληλισμού που παρέχουν τα πολλαπλά νήματα και πυρήνες του επεξεργαστή. Τα πειραματικά αποτελέσματα μας δείχνουν ότι ο προτεινόμενος αλγόριθμος χρονοπρογραμματισμού (στον επεξεργαστή OpenSPARC T1) επιταχύνει το χρόνο εκτέλεσης του λογισμικού αυτοδοκιμής, κατά 3.6X φορές σε επίπεδο πυρήνα, ενώ συνολικά σε επίπεδο επεξεργαστή έως 6.0X σε σύγκριση με την εκτέλεση του λογισμικού σε ένα μόνο νήμα εκτέλεσης. Επιπλέον, σε σύγκριση με μια απλή πολυνηματική εκτέλεση του λογισμικού αυτό-δοκιμής ο προτεινόμενος αλγόριθμος μειώνει την διάρκεια εκτέλεσής κατά 33% και 20% σε επίπεδο πυρήνα και επεξεργαστή αντίστοιχα.
2. Να διατηρήσει σε υψηλά επίπεδα το ποσοστό κάλυψης ελαττωμάτων υλικού (σχεδόν 90% σε ολόκληρο τον επεξεργαστή ο οποίος αποτελείται από 1.5 εκατομμύρια λογικές πύλες) και
3. Να βελτιστοποιήσει το ποσοστό κάλυψης ελαττωμάτων υλικού στα τμήματα λογικής όπου σχετίζονται με την πολυνηματική και πολυπύρηνη εκτέλεση. Η μεθοδολογία MT-SBST επιταχύνει σημαντικά τη διαδικασία ελέγχου ορθής λειτουργία ενός επεξεργαστή, ενώ παράλληλα βελτιώνει συνολικά το ποσοστό κάλυψης ελαττωμάτων.

Η συνεχώς αυξανόμενη πιθανότητα εμφάνισης σφαλμάτων υλικού κατά τη διάρκεια λειτουργίας ενός επεξεργαστή (in-field verification), οδήγησε στην εισαγωγή μηχανισμών επαλήθευσης της ορθής τους λειτουργίας. Στο πλαίσιο αυτής της διδακτορικής διατριβής, υλοποιήθηκε ένα εργαλείο αυτόματης εισαγωγής μόνιμων σφαλμάτων (permanent) στην αρχιτεκτονική x86-64 (η πιο διαδεδομένη αρχιτεκτονική υπολογιστών, χρησιμοποιείται από τις εταιρίες Intel και AMD). Το εργαλείο αυτό ενσωματώθηκε στον ευρέως διαδεδομένο αρχιτεκτονικό προσομοιωτή PTLsim. Κατόπιν, αναλύθηκε σε βάθος η επίδραση που έχουν τα μόνιμα σφάλματα (μονά και σε ορισμένες περιπτώσεις πολλαπλά) στους εξής μηχανισμούς:

- Μηχανισμός πρόβλεψης διακλάδωσης (branch prediction unit): Ο μηχανισμός αυτός προβλέψει την επόμενη εντολή που θα εκτελεστεί από τον επεξεργαστή βασιζόμενος στο τρέχον ιστορικό εκτέλεσης.
- Μηχανισμός εκ των προτέρων προσκόμισης δεδομένων (data prefetching). Ο μηχανισμός αυτός αναζητά συγκεκριμένα πρότυπα στον

τρόπο πρόσβασης στην ιεραρχία μνήμης. Στην περίπτωση όπου διαπιστωθεί ότι ένα συγκεκριμένο πρότυπο επαναληφθεί αρκετές φορές (το πλήθος καθορίζεται κατά τη σχεδίαση του μηχανισμού), τότε προχωράει στην πρόωρη προσκόμιση δεδομένων από την μνήμη του επεξεργαστή έτσι ώστε να αποφευχθούν μελλοντικές καθυστερήσεις στην εκτέλεσης λόγω αστοχιών στη μνήμη δεδομένων.

Η ιδιαιτερότητα των σφαλμάτων στους μηχανισμούς αυτούς έγκειται στο γεγονός ότι επηρεάζουν αποκλειστικά την απόδοση του επεξεργαστή και όχι την ορθότητα της εκτέλεσης. Κατά συνέπεια, η εκτέλεση ενός προγράμματος μπορεί να καθυστερήσει να ολοκληρωθεί εξαιτίας των σφαλμάτων στους μηχανισμούς αύξησης της απόδοσης. Τα πειραματικά αποτελέσματα υποδηλώνουν ότι ένα μεγάλο πλήθος σφαλμάτων στους προαναφερθέντες μηχανισμούς επηρεάζουν την απόδοση του επεξεργαστή (έως το 96% των μόνιμων σφαλμάτων αναλόγως το μηχανισμό και το εκτελούμενο πρόγραμμα). Παράλληλα, η επίδραση των σφαλμάτων στην απόδοση του συστήματος μπορεί να είναι καταστροφική. Για παράδειγμα, σφάλματα στο μηχανισμό εκ των προτέρων προσκόμισης δεδομένων μπορεί να επιβαρύνει την απόδοση ενός συστήματος ως 26%.

Εκτός από την αξιολόγηση των μόνιμων σφαλμάτων στους μηχανισμούς αύξησης της απόδοσης, προτάθηκαν τεχνικές για την ανίχνευση και ανοχή μόνιμων σφαλμάτων υλικού στους μηχανισμούς πρόβλεψης διακλάδωσης. Ειδικότερα, η ανίχνευση επιτυγχάνεται μέσω της αξιοποίησης της αυτόματης διόρθωσης των μηχανισμών πρόβλεψης διακλάδωσης, ενώ η ανοχή μέσω της αναδιαμόρφωσης του υλικού. Τα πρωτότυπα πειραματικά αποτελέσματα αυτής της διδακτορικής διατριβής καθιστούν εμφανή τη θετική επίπτωση των προτεινόμενων μηχανισμών για την ανίχνευση και ανοχή σφαλμάτων υλικού.

Τα υπολογιστικά συστήματα έχουν διεισδύσει, περισσότερο από ποτέ, στις καθημερινές δραστηριότητες της κοινωνίας μας. Παρόλα αυτά, οι τεχνολογικές εξελίξεις, τόσο στην τεχνολογία υλικών όσο και στην αρχιτεκτονική υπολογιστών, που μας οδηγούν σε αυτά τα αξιοθαύμαστα επιτεύγματα, αυξάνουν την αναξιοπιστία των υπολογιστικών κυκλωμάτων θέτοντας, ταυτοχρόνως, σε κίνδυνο την κοινωνία μας. Συνεπώς, τόσο οι εταιρίες κατασκευής ολοκληρωμένων κυκλωμάτων όσο και η ακαδημαϊκή κοινότητα είναι αναγκαίο να εφεύρει νέες λύσεις και να αναπτύξει καινοτόμες τεχνικές που να αντιμετωπίζουν το κρίσιμο πρόβλημα της αναξιοπιστίας των σύγχρονων αλλά και μελλοντικών επεξεργαστών. Ειδικότερα, οι μελλοντικές αρχιτεκτονικές υπολογιστών θα πρέπει να διευκολύνουν όσο το δυνατόν περισσότερο το έλεγχο ορθής λειτουργίας ενός υπολογιστικού κυκλώματος σε όλες τις φάσεις τις σχεδιαστικής αλυσίδας. Αυτή η διδακτορική διατριβή αποτελεί ένα σημαντικό βήμα προς αυτή την κατεύθυνση. Ειδικότερα, προτείνονται νέες μεθοδολογίες βασιζόμενες στις ακόλουθες θεμελιώδεις σχεδιαστικές προδιαγραφές: (α) χαμηλού κόστους λύσεις σε ενέργεια αλλά και πολυπλοκότητα και (β) αυτοματοποίηση, με σκοπό την αντιμετώπιση των προκλήσεων του ελέγχου αξιοπιστίας των κυκλωμάτων σε ολόκληρο τον κύκλο ζωής τους.

Τα ερευνητικά αποτελέσματα αυτής της διδακτορικής διατριβής ανοίγουν διάπλατα το δρόμο σε νέες ερευνητικές δραστηριότητες. Ειδικότερα, στα πλαίσια του silicon debug, οι μελλοντικές τεχνολογίες εντοπισμού και διόρθωσης σφαλμάτων θα πρέπει να επικεντρωθούν στην αυτοματοποίηση και στην τυποποίηση της διαδικασίας ανίχνευσης και αποσφαλμάτωσης σχεδιαστικών σφαλμάτων. Επιπρόσθετα, αυτή η διδακτορική διατριβή απέδειξε την αποτελεσματικότητα της υιοθέτησης του λογισμικού αυτό-δοκιμής στην

επιτάχυνση του manufacturing testing διατηρώντας ένα εξαιρετικά υψηλό ποσοστό κάλυψης ελαττωμάτων. Η επιτυχία αυτής της μεθόδου θα μπορούσε να αποτελέσει μία κατεύθυνση για τους μελλοντικούς μικροεπεξεργαστές. Επιπρόσθετα, η συνεχώς αυξανόμενη ανάγκη για υπολογιστικά συστήματα υψηλών επιδόσεων ωθεί του αρχιτέκτονες επεξεργαστών στην προσθήκη πολλών μηχανισμών αύξησης της απόδοσης στη σχεδίαση ενός επεξεργαστή. Ωστόσο, η ορθή λειτουργία αποτελεί πρωταρχικό στόχο έως τώρα έναντι της σωστή λειτουργίας αλλά στα αναμενόμενα χρονικά περιθώρια. Αυτή η διδακτορική διατριβή κατέδειξε με εμφατικό τρόπο ότι τα σφάλματα υλικού στα τμήματα της λογικής που συμβάλουν στην αύξηση της απόδοσης μπορούν να οδηγήσουν σε σημαντική απόκλιση της απόδοσης του επεξεργαστή σε σχέση με την αναμενόμενη συμπεριφορά του. Συνεπώς, οι μελλοντικοί επεξεργαστές πρέπει να ενσωματώσουν ένα πλήθος μηχανισμών για τον συνεχή έλεγχο της επίδοσης ενός συστήματος και να είναι σε θέση να αντιμετωπίζουν περιπτώσεις αστοχίας. Τέλος, μία σημαντική μελλοντική ερευνητική κατεύθυνση αποτελεί ο εντοπισμός τεχνικών οι οποίες να μπορούν να εφαρμοστούν σε όλα τα στάδια της σχεδιαστικής αλυσίδας, από τα πρωτότυπα μοντέλα ενός επεξεργαστή μέχρι την κανονική, παραγωγική, επιχειρησιακή, εμπορική του λειτουργία.

Συμπερασματικά, μια ζωτικής σημασίας πρόκληση των μελλοντικών τεχνολογιών αποτελεί η κατασκευή υπολογιστικών συστημάτων τα οποία θα λειτουργούν με αξιοπιστία σύμφωνα με τις προδιαγραφές κατασκευής τους. Αυτή η διδακτορική διατριβή προτείνει καινοτόμους μηχανισμούς αποσκοπώντας στη βελτίωση της αποδοτικότητας της επαλήθευσης της λειτουργίας ενός μικροεπεξεργαστή σε όλη την διάρκεια λειτουργίας του, με διττό στόχο, ήτοι το υψηλό ποσοστό κάλυψης σφαλμάτων και τη μικρότερη δυνατή επιβάρυνση στη σχεδίαση του υπολογιστικού συστήματος. Ελπίζουμε ότι συνεισφορές που παρουσιάζονται στο πλαίσιο αυτής της διδακτορικής διατριβής να προωθήσουν ακόμη περισσότερο την έρευνα στην ανάπτυξη αξιόπιστων υπολογιστικών συστημάτων. Παράλληλα, να βρουν ευρεία απήχηση στους μελλοντικούς επεξεργαστές.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Information and Communication Technology (ICT) systems are growing exponentially powered by the progress in semiconductor technology and in computer architecture. Semiconductor innovation has repeatedly provided more transistors (Moore's Law [73]) for roughly constant power and cost per chip (Dennard Scaling [35]), while computer architects took these rapid transistor budget increases and discovered innovative techniques to scale processor performance. A recent study [31] measured that technology scaling and architecture improvements contributed almost equally to computer performance growth, with architecture credited with ~80x improvement since 1985. Thus, the combined effect of technology and computer architecture makes microprocessors the most complex and immensely powerful application of the electronics.

The same path that is leading technology toward these remarkable achievements is also making microprocessors increasingly unreliable posing a threat to our society. Device integration, design complexity along with the compelling requirement to diminish the Time-to-Market (TTM) are expected to dramatically reduce semiconductor product quality: as the transistors and wires shrink and the circuits complexity increases, microprocessors are becoming more prone to failures, show larger differences in behaviour although they are designed to be identical and have higher susceptibility to environmental-induced phenomena. The vital challenge of future technologies is to build dependable systems. That is the goal of this thesis; to provide solutions to the dependability challenges posed from the current and future microprocessor products.

1.1 The ecosystem of a microprocessor

Nowadays, the pervasiveness of microprocessors in our society goes far beyond the wildest imagination, from their humble beginning (on 1971's Intel introduced the first commercial processor, Intel 4004 [33]). Worldwide combined shipment of electronic devices (such as, Desktop PCs, Portable PCs, Smartphones and Tablets) is projected to reach 2.5 billion units until 2017 (Figure 1).

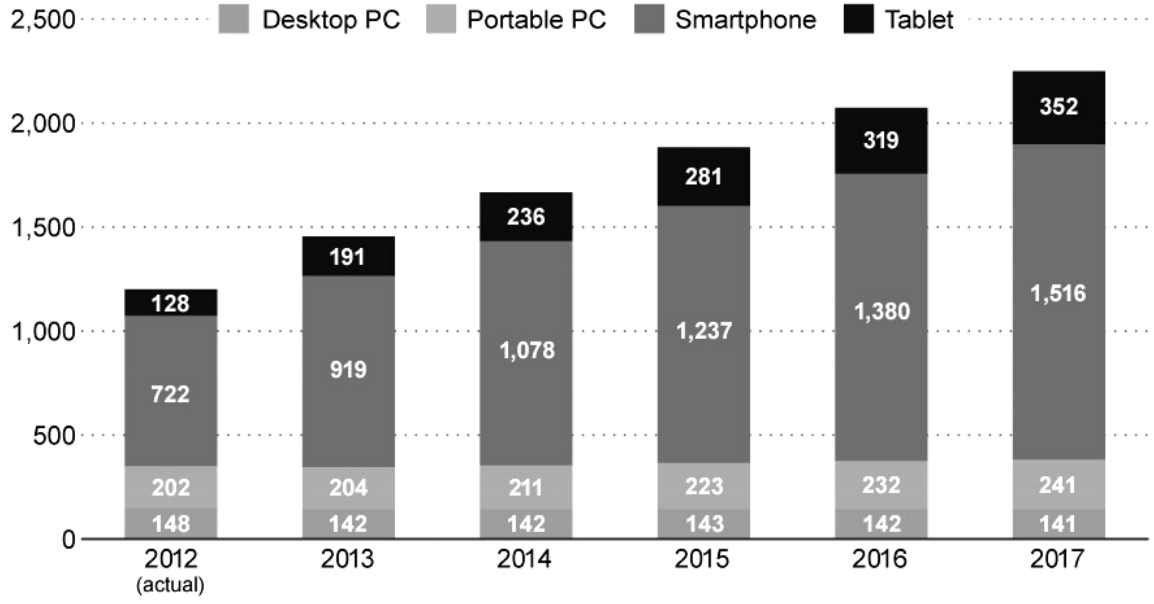


Figure 1: The evolution of global connected electronic devices. Desktop PC, portable PC, smartphone, and Tablet shipment through past to present and the forecast for the future (source: IDC, www.idc.com).

Chip manufacturers have been striving to increase microprocessor's performance by cramming more and more transistors into a silicon die. Throughout the last decades, transistors have conducted electricity along a planar surface of a silicon wafer. Today, tri-gate or multi-gate transistors mark a major change in semiconductor technology. Three dimensional transistors deliver superior levels of scalability extending the life of Moore's Law. Currently the feature size of the most new chips in massive production is in the range of 22nm and some chips are starting to use 14nm technology (Intel's, code-name Broadwell [55], microarchitecture will be shipped during the second half of 2015). The miniaturisation process of transistor technology is set to continue and we expect more than 61 billion transistors (Figure 2) in a single die by 2020 (3x increase compared to the design density we have today).

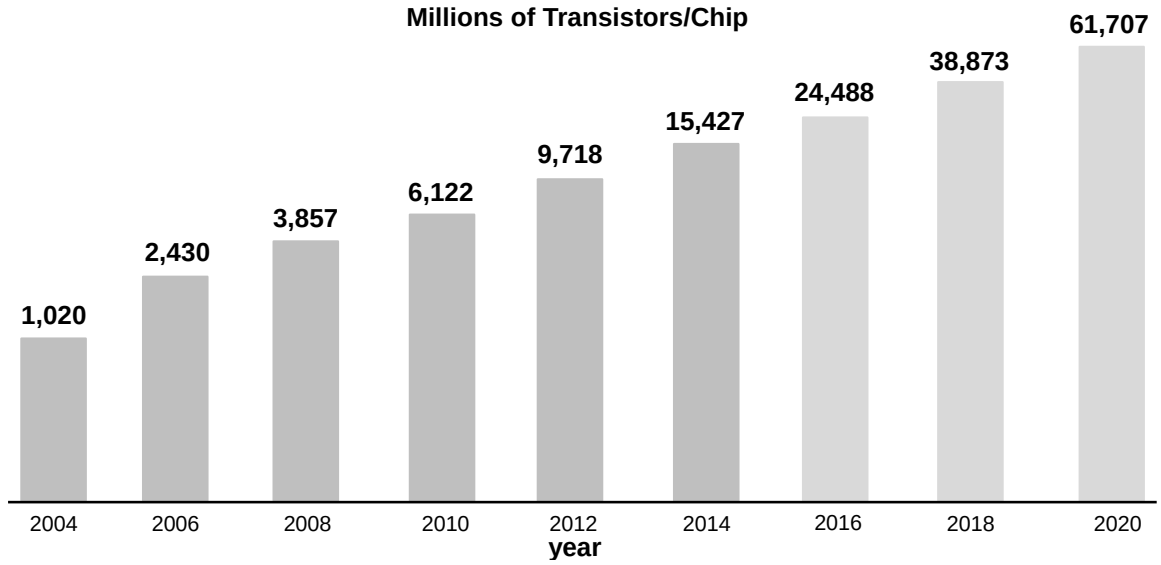


Figure 2: Current and future projections of design density, measured in millions of transistors per chip (source: International Technology Roadmap for Semiconductors, ITRS, 2008 update, Mentor Graphics).

As manufacturing technology provides higher transistor density, microprocessors exploit the additional transistors to boost their efficiency. Figure 3 shows the growth in processor performance since the mid-1980s. An arsenal of performance enhancement techniques, such as aggressively speculative mechanisms (e.g.: sophisticated branch prediction units, data and instruction prefetch mechanisms, value predictors), higher capacity caches, resource-, data-dependency handling structures and massively parallel pipelined processing elements allocate the additional area in silicon estate to exploit the available Instruction-, Thread-, and Data-Level Parallelisms (ILP, TLP, DLP, respectively). To be so powerful, processors have become extremely complex systems, making the design and manufacturing of these devices a major challenge for the semiconductor industry. Major semiconductor companies such as Intel, AMD, ARM and IBM are forced to dedicate hundreds of engineers to continue to advance microprocessor technology and deliver better performance to end-users.

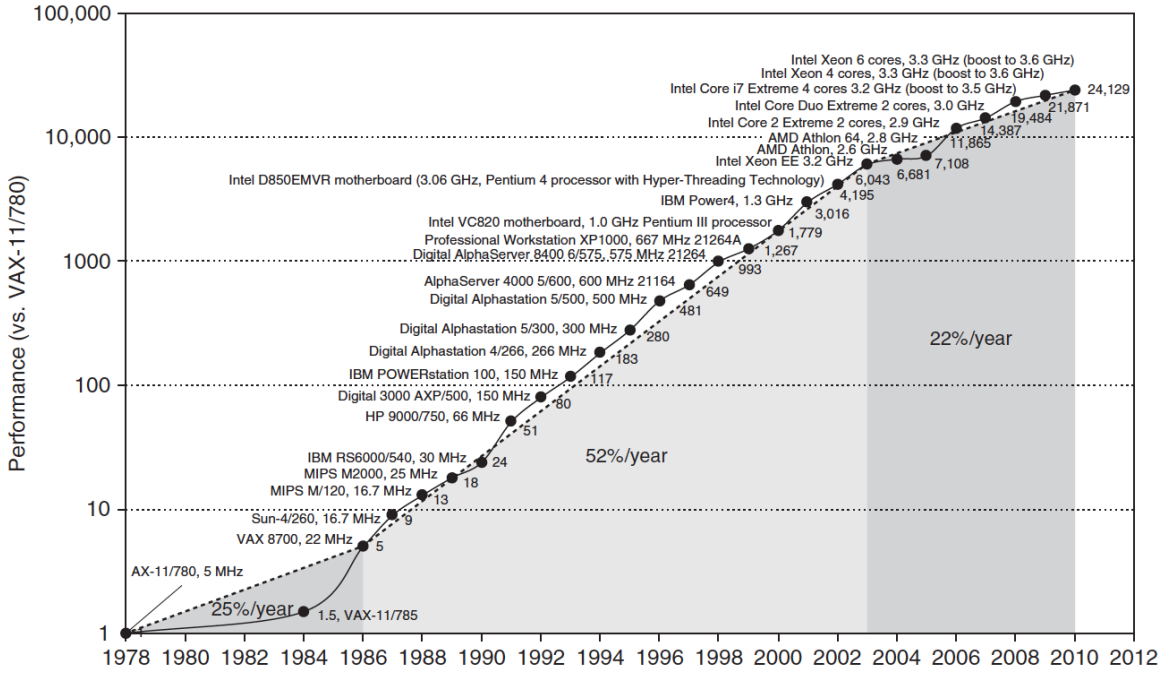


Figure 3: Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks. Prior to the mid-1980s, processor performance growth was largely technology driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. Since 2002, processor performance improvement has dropped to about 22% per year due to the following hurdles: (a) heat dissipation, (b) little Instruction-Level Parallelism (ILP) left to exploit, and (c) limitation lay by memory latency. These obstacles signal historic switch from relying solely on ILP to Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP) [47].

As microprocessor designs grow in complexity, it becomes increasingly harder to verify them and ensure that they operate properly. Design houses report [93] that today verification efforts significantly overweight design activities, since the ratio between the size of the design and debug teams has reached 2:1. We can only expect the situation to exacerbate with future performance demands, to the point that high-quality verification of microprocessors will no longer be possible with traditional means. Unless the verification demands of the modern microprocessor are answered, chips released to the public will become more and more unreliable containing significant numbers of design bugs and manufacturing defects. Clearly, an efficient verification process can make the difference between the success and the failure of a microprocessor product.

Putting it all together, a modern microprocessor has the following characteristics: (a) billions of transistors, (b) integrates complex micro-architectures, and (c) is implemented on top of unreliable fabrics. Meanwhile, users expect a microprocessor to remain reliable and to continue to deliver the rated performance. This challenge will undoubtedly require a major paradigm shift in all aspects of microprocessor design – fabrication, design, debug, and testing. This thesis provides solutions to the dependability challenges posed throughout the life-cycle of a microprocessor product.

1.2 Design life-cycle of a microprocessor

The aggressive transistor scaling and the ever-growing design density have enabled microprocessors performance to boost at a dramatic pace; however, the required effort to manufacture modern microprocessors is continuously

increasing. The microprocessor life-cycle consists of the following high-level phases (Figure 4).

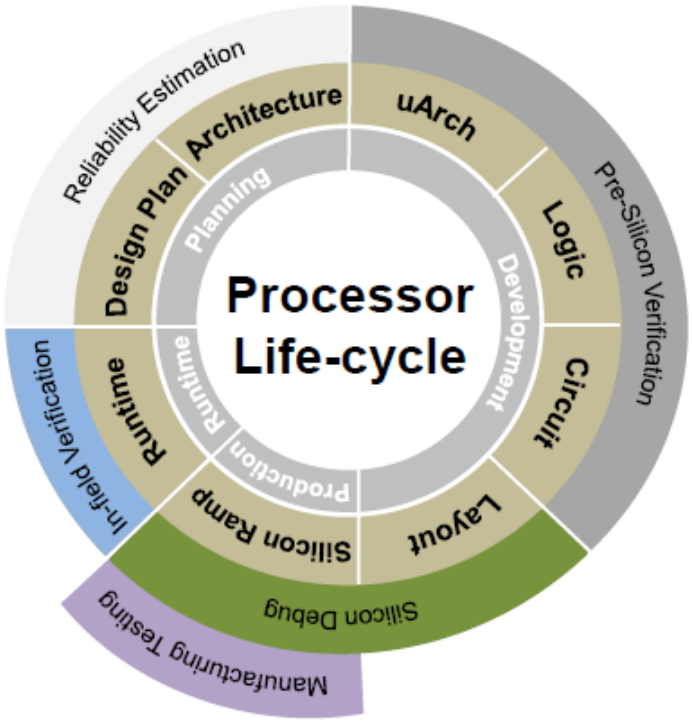


Figure 4: The life-cycle of a microprocessor product. The design flow is comprised of a series of steps that considers the design planning, the development of the specified functionalities and the implementation of the silicon die. Along with the development flow, the design is progressively verified (reliability estimation, pre-silicon verification, silicon debug, manufacturing testing, and in-field verification), to ensure that, the microprocessor adheres to the design specification.

Design planning: Computer architects define the microprocessor product design and manufacturing strategy. In particular, product requirement analysis, targeted market segment, technology selection, design methodology and tool selection constitute the vital tasks of design planning. For example, the architecture team explores the feasibility of diverse core architectures and verifies whether they provide the expected benefits, in terms of functionality, performance and power budget. In addition, the design team tests out new circuit ideas with test chips, and evaluate radical modifications in circuit and layout process. Table 1 illustrates three key product requirements specified within design planning.

Table 1: Microprocessor product requirements analysis specified during design planning phase.

Market Segment	Product	Requirement
Server	High throughput server	Performance, reliability
Desktop	High performance desktop	Performance
	Mainstream desktop	Balanced performance and cost
	Low-cost desktop	Lowest cost at required performance
Embedded	Smartphone	Ultra-low power budget

Development: Within this phase the microprocessor is developed and manufactured based on the design specifications set on the design planning phase. The development process starts with the definition of the microprocessor

features that are visible to the Operating System (OS) and to user applications, such as the Instruction Set Architecture (ISA).

Table 2: The architecture-related features of a microprocessor defined within development phase.

Features	Possible choices
Operand types	Register, Register/Memory, Memory
Data formats	Integer, Floating-point, SIMD
Data addressing modes	Absolute, Register indirect, Displacement, Indexed
Instruction encoding	Fixed or Variable, #of registers, Immediate size

Then, during microarchitecture development process the detailed implementation of the specified architectural features (i.e. assembly instructions), various logic elements (such as, functional, control and memory blocks) as well as the interaction among them are developed. Microarchitectural modifications can noticeably improve performance, while remaining transparent to the higher level of the system stack. Microprocessor characteristics and functionalities are described through an architectural model of the device, typically written in a high-level programming language, such as C++. This model represents the first formalized reference of the final system's behaviour. Then, the logic design aims at generating a formal description of the logical behaviour of all the components and the interaction among them. A Hardware Description Language (HDL), such as Verilog or VHDL, is exploited to describe and simulate the hardware design. Depending on the level of abstraction a hardware description language can range from the behaviour level (i.e. maps major microprocessor events without the time notion), the Register-Transfer Level – RTL (i.e. models the processor clock along with the detailed description of the events occurred in each clock phase) to the structural level (i.e. the gate-level implementation of the design). The last phase on the development cycle of a microprocessor incorporates the circuit and layout design process. The former generates the transistor-level specification of the logic modelled through the HDL, while the latter maps transistors and wires on the different layers of the material to make up the circuit. The layout step constitutes the transition between a simulation-based to the silicon-based design implementation.

Production: On silicon ramp the first silicon prototypes are manufactured and thoroughly validated. Design fixes that adjust microprocessor performance and functionality according to design specifications are applied. Then, the massive volume production starts.

Runtime: The last phase on the design life-cycle of a microprocessor, where the product is shipped into the market. The manufactured microprocessor is functional and adheres to the design specifications. From this point onwards, the design teams have no interaction with the developed design.

1.3 Dependability life-cycle of a microprocessor

Dependability – the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [39] – constitutes an integral part of computer architecture. Dependable operation of computing systems is a key challenge for the whole information and communication technology since almost every human activity relies largely or even completely on computing systems.

Computer architects are facing progressively more challenges in ensuring microprocessor products to be free of design bugs and hardware errors, due to the ever-growing design complexity and the continuously shortening time-to-market. Design bugs and hardware errors are detected either before chips fabrication, during pre-silicon verification, or after fabrication, during silicon debug. Without major breakthroughs, microprocessor verification will be a non-scalable, a show-stopping barrier to further progress in the semiconductor industry.

The development of a microprocessor is coupled with a sequence of verification tasks. Throughout this process, validation engineers utilize a multitude of verification tools to ensure that the design adheres to its specifications. Microprocessor dependability tasks can be grouped into five categories, based on where they intervene in a microprocessor's lifecycle (Figure 4):

Reliability Estimation: Early assessment of the expected reliability of a microprocessor is a critical task which steers the design decisions related to the required mechanisms for the in-field error detection and protection. Such fault tolerance mechanisms may impose significant area, power and performance overheads. Straightforward guard-banding of the system with inaccurate knowledge of the effect of hardware faults can easily make the costs of protection against hardware faults excessive. For example, typical memory error detection and protection techniques can have a cost (in terms of added memory capacity) which ranges from 1% to 125% depending on the detection and protection capabilities of each technique [80]. Clearly, the selection of the most appropriate protection technique depends on the required reliability levels and studies of its inherent resiliency to hardware faults. Detection and protection mechanisms against any fault model must be decided as early as possible to avoid costly re-design cycles for late integration of such mechanisms. However, early decisions on the protection mechanisms are hard to make because during the early stages of a system design a formalized model of the system is missing.

Pre-Silicon Verification: it is mainly based on simulation at different levels of abstraction [13]. Despite its maturity and the tremendous utilization of computing resources, it is impossible to guarantee that all design bugs have been fixed before tape-out [62] because only a small number of functional scenarios can be simulated during pre-silicon verification. Statistics show that 12% of design bugs slip into first silicon prototypes and almost 50% of microprocessor chips require extra unplanned tape-outs due to bugs found in the first manufactured chips [4]. During pre-silicon verification, simulation-based tests, run on RTL model, are compared to those of the known-correct (or "golden") architectural model and discrepancies, indicators of design bugs, are identified and fixed. In addition, pre-silicon verification engineers employ formal methods, which can check correctness of a design using mathematical proofs and can thus guarantee the absence of certain types of errors. Unfortunately, formal methods cannot handle complex RTL models due to their limited scalability; therefore, their usage is limited to a few small critical blocks.

Silicon debug: it targets to ensure that a chip's actual silicon implementation fully matches its specification, that is, the planned intended behaviour of the device. Silicon debug – the process of validating and debugging a new microprocessor design on its first silicon prototype chips – has evolved to a critical, time-consuming, and labour-demanding step in a chip's development flow [2]. The underlying reasons for this unmanageable complexity lie in the inability of validation techniques to keep pace with the highly integrated microprocessors.

Recent trends [37] show that the time spent from the arrival of the first silicon prototype chips to high volume production is steadily growing, while the ratio between the size of the design and the debug teams has reached 2:1. Thus, efficient silicon debug approaches that promptly detect and eliminate the majority of design bugs before volume production can make the difference between success and failure of a microprocessor product [82]. During silicon debug a comprehensive suite of test programs (such as automatically generated random test programs, legacy tests and real world applications) covering many test scenarios are executed on the prototype chips to detect any abnormal behaviour that stems from a design bug. The application of automatically generated random test programs on the prototype microprocessor chips is one of the most effective parts of silicon debug [20]. When a bug is found at this stage, the RTL model is modified to correct the issue and the chip must be manufacturing again. Furthermore, volume production may be further prolonged due to bugs that lurk behind other bugs. These blocking bugs stall the execution of the subsequent tests, since no workaround exists and therefore additional re-spins are needed.

Manufacturing testing: Manufacturing testing constitutes an integral part on microprocessor verification cycle. The reason for that is the existence and effectiveness of test metrics such as stuck-at coverage, transition fault coverage and N-detect coverage. When a sufficient level of defect coverage is reached the microprocessor design enters the production stage, where a last quality control is performed to detect any manufacturing defect. Manufacturing testing techniques aim at maximizing the fault coverage (i.e. the population of hardware faults detected through fault simulation), while minimizing test costs, in terms of time and resources. Overall, the manufacturing testing methods applied to this step attempt to achieve the target defective parts per million (DPPM) rate that high-quality product development demands. High-volume manufacturing (HVM) [107] testing of microprocessors incorporates both functional and structural test approaches. The functional testing methods, such as the Software-based Self-testing (SBST) utilize the on-chip programmable resources to apply at-speed the test stimuli and collect the test responses from memory to make the pass/fail decision. On contrast, the structural test approaches exploit the knowledge of the circuit structure and the corresponding fault model to generate the test patterns. For example, scan-based testing, replace the storage elements with scan cells, and connecting them into shift registers to provide access to the internal state of the circuit. Structural testing usually places the circuit in specific self-test mode and cause excessive test power consumption, over-testing, and thus may lead to yield loss, compared to functional testing.

In-field Verification: Technology miniaturization, design complexity, shrinking time-to-market windows, wear-out effects and the environmental impact increase the failure probability of modern design and steer microprocessor manufacturers to integrate numerous in-field verification mechanisms. Dual- and triple-modular redundancy are traditional in-field fault tolerance techniques, which can detect and correct hardware errors, but only at high costs. Others in-field fault tolerance technique to protect memories, buses or other microprocessor array structures are the parity mechanism and the error correction codes (ECC). In the literature, several in-field verification techniques have been proposed for SRAM caches [5] [28] [8] as well as mechanisms to protect pipeline flip-flops and combinational logic [23] [84] [94].

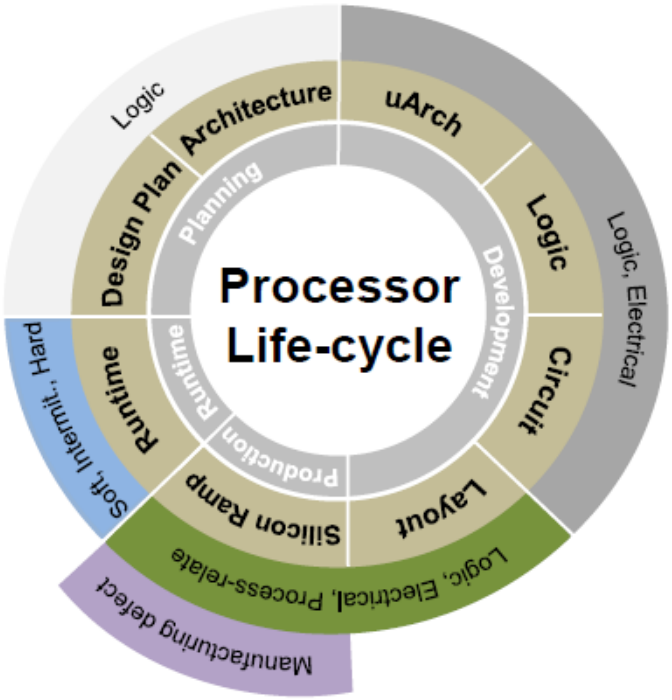


Figure 5: Design bugs and hardware errors distribution throughout microprocessor's life-cycle.

1.4 The nature of a failure

Throughout the lifetime of a microprocessor product, its silicon fabric is subject to a variety of failure mechanisms that can cause device failures. As the transistor dimensions scale to smaller sizes, the silicon failure mechanisms get aggravated. The types of failures that are expected throughout the life-cycle of a microprocessor (Figure 5) are the following:

Design bugs: Logic, electrical and process-related bugs [16] [29] may be introduced into microprocessor products during design specification and implementation phase. Follows an analysis on the common sources of bugs [50]: The limited capacity and performance of verification techniques, which do not keep pace with the growth in the amount and complexity of the developed code, along with the growth in design complexity increase the trend of having inadequately or incorrectly specified and implemented designs. Furthermore, synthesis tools may hamper the accuracy of the synthesized design. As a result, discrepancies between the intended and the developed functionality may exist. Place and route process is another source of bugs. For instance, the physical specification requirements may partial be achieved during the layout process. In addition, the combination of process variation and smaller design margins prevent microprocessor products from functioning at the intended frequency, while dynamic power consumption and crosstalk effects may randomly flip values of memory cells. Technology scaling, model inaccuracies and the lack of efficient design-rule-checking tools may increase these sources of bugs.

Several years of experience of microprocessor manufacturers have shown that numerous important design bugs escape (so called, errata bugs) in production silicon despite the extremely large efforts of the verification team. All microprocessors have known errata bugs (some of them are presented on Table 3 while the rate of bug escapes has more than doubled in the latest generation of Intel processors [29]. Furthermore, more than half of design bugs that slip into volume production have no fixes and for those that a fix exists, the in-field

workaround is often too costly [12]. Thus, effective verification methods are needed to ensure that forthcoming architectures will not suffer from severe bug escapes.

Table 3: A list of errata bugs on popular microprocessor.

Microprocessor	Description
Intel Pentium®	FDIV bug was a bug in the Intel P5 Pentium floating point unit. Certain floating point division operations produced incorrect results. According to Intel, there were a few missing entries in the lookup table used by the divide operation algorithm [83].
Intel Pentium®	On Pentium 4, If a cache hits on modified data (HITM) while a snoop is going on, and there are pending requests to defer the transaction and to re-initialize the bus, then the snoop is dropped, leading to a deadlock [52].
AMD Opteron™	AMD64 processors led to incorrect results in certain situations when a REP MOVS instruction was executed. An incorrect address size, data size or source operands segment might be used or a succeeding instruction might be skipped. This occurred only under certain conditions and led to production of incorrect results or system's freeze [53].
IBM PowerPC®	750GX processor bug caused some instructions to execute at lower frequencies (933Mhz compared to 1Ghz) [54].
Intel Core™	In the 6-series chipsets, of Sandy Bridge microarchitecture, the Serial-ATA (SATA) ports within the chipset degraded over time, potentially impacting the performance or functionality of SATA-linked devices such as hard disk drivers and DVS-drivers [56].

One of the primary driving forces to develop failure detection techniques is the cost that a company experiences as a result of fixing a bug. Figure 6 shows that the relative cost of fixing a bug increases over time. In particular, during the early stages of a project, changes do not require much rework. Only the Register-Transfer Level (RTL) is affected, so the cost is very low. Later in the design cycle, changes to RTL cause schematic change and layout changes, so the cost starts rising. After the part is send to the fab, the cost of changing the design to fix bugs includes the cost of building new masks and manufacturing new parts. After parts start being sold a serious bug may require a recall of parts at a significant expense to the company, an expense that grows with the number of parts sold. Obviously, there is a huge advantage in finding bugs as early as possible in the design process. This reduces the amount of modifications later in the process, yielding a lower cost of development.

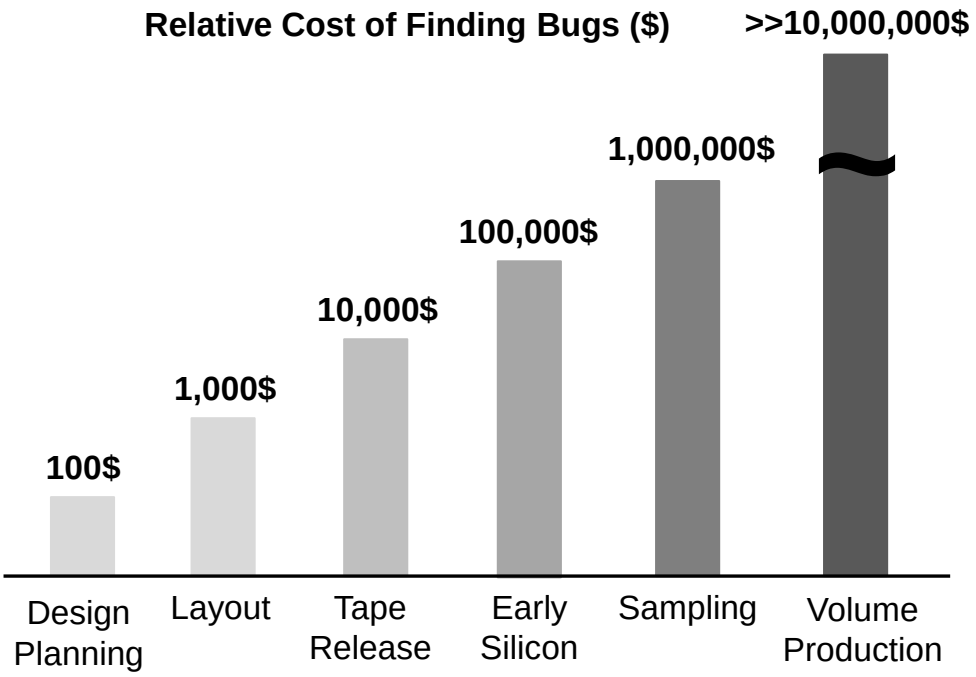


Figure 6: Relative cost of detecting bugs throughout microprocessor's life-cycle [37].

Manufacturing defects [86]: Moving into deeper nanometer scale manufacturing process, an increased amount of manufacturing-related defects will be introduced into the designs. Optical proximity effects, airborne impurities, and processing material defects can all lead to the manufacturing of faulty transistors and interconnects. Moreover, deep-submicron gate oxides have become so thin that manufacturing variations can lead to currents penetrating the gate, rendering it unusable. Manufacturing defects are also affected by the immense complexity of current and forthcoming microprocessor designs. Design complexity makes it more difficult to test for defects during manufacturing. Semiconductor industry is forced to either spend more time with parts on the tester, which reduces profits by increasing time-to-market, or risk the possibility of untested defects escaping to the field.

In-field errors [86]: Integrated circuits are implemented in miniaturized and inherently unreliable technologies. This leads to products that are more prone to transient, intermittent and permanent errors. Single-Event Upsets caused by neutrons and alpha particles that strike the bulk silicon portion of the die. Although SEUs do not break the silicon their effect in a logic glitch that can potentially corrupt computational logic or state bits. Hard errors, on the other side, appear either because of manufacturing defects that escape high-volume production manufacturing testing or because of material aging and wear-out effects. Finally, another source of hardware errors is the process variation, i.e. variations in device characteristics. In particular, process variation can cause large fluctuation in performance and power consumption in the manufactured chips. Current microprocessors show large differences in behaviour although they are designed to be identical. Process variation is expected to be amplified in the forthcoming microprocessor designs.

1.5 Contributions of this thesis

The evolution of semiconductor technology and computer architecture has radically transformed our world throughout the last decades. However, the combination of technology scaling and extreme chip integration, along with the

compelling requirement to diminish the time-to-market window, has rendered microprocessors more prone to design bugs and hardware errors. The goal of this thesis is to provide solutions to the validation challenges posed from the microprocessor products *starting from the first silicon prototypes till in-field operation* of the chip. The contributions of this thesis are the following:

- **Silicon debug**: The validation step that detects the vast majority of design bugs is the one that stresses the silicon prototypes by applying huge numbers of random tests. Despite its bug detection capability, this step is constrained by the extreme computing needs for random test program simulation (to extract the bug-free memory image to compare with the actual silicon image). Moreover, another major bottleneck and source of "noise" of this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behaviour adds long delays in the debug flow since each failing random program must be separately examined, although it does not usually bring new debug information. This thesis addresses both challenges of silicon debug. A self-checking methodology [10] is proposed for generating random test programs (exploiting the ISA diversity property) that detect bugs by comparing the results of equivalent instructions combined with a technique to group the failing test programs into categories (the failing test programs are grouped into categories depending on the microprocessor hardware components that need to be deconfigured for a random test program to be correctly executed) [8]. The proposed framework: (a) improves bug detection efficiency, (b) reduces the redundant debug session, and thus accelerates silicon debug.
- **Manufacturing testing**: Functional self-testing forms an integral part of manufacturing test flow due to its at-speed testing and non-intrusive nature. Multithreaded (MT) SBST methodology [11] proposes a novel self-test optimization strategy for multithreaded, multicore microprocessor architectures. The proposed self-test program execution optimization aims to: (a) take maximum advantage of the available execution parallelism provided by multiple threads and multiple cores, (b) preserve the high fault coverage that single-thread execution provides for the processor components, and (c) enhance the fault coverage of the thread-specific control logic. MT-SBST methodology significantly speeds up self-test time, while at the same time it improves the overall fault coverage.
- **In-field verification**: The combination of design complexity, shrinking time-to-market windows, and wear-out effects increases the failure probability of modern design and leads microprocessor manufacturers to integrate numerous runtime verification mechanisms. Modern microprocessors use a noticeable silicon estate to implement various control and data flow speculative hardware. In this thesis, an analysis on the performance degradation of control flow predictors and data prefetchers based on projected rates of faults in future technologies is presented [2] [5] [6]. Then, low-cost microarchitectural techniques to diagnose predictor faults and recover the performance loss are presented. The proposed techniques exploit the self-verification property of predictors to achieve performance recovery at lower cost than comparable techniques. The presented solutions manage to recover almost all IPC lost, virtually eliminate performance variability among cores.

Modern microprocessors implement extremely complex architectures, making the validation process a major challenge for the semiconductor industry. This thesis introduces various novel methodologies to address the validation challenges posed throughout the life-cycle of a chip. The proposed techniques make the validation process more efficient and are easily applicable to the existing industrial flow.

1.6 Thesis outline

The remainder of this thesis is organized as follows:

Chapter 2 presents the proposed mechanisms for silicon debug. In particular, the employed self-checking random test programs along with the deconfigurable microprocessor architecture to avoid the time-consuming simulation step and triage the redundant debug sessions are analysed.

Chapter 3 presents MT-SBST, a novel self-test optimization strategy for multithreaded, multicore architectures for accelerating manufacturing testing and improving fault coverage.

Chapter 4 introduces the performance impact of hard errors on a core's speculative structures along with a low-cost microarchitectural technique to diagnose predictor faults and recover the performance loss.

Finally, Chapter 5 presents the concluding remarks and discusses directions for future work.

2. SILICON DEBUG

Aggressive technology scaling and extreme chip integration, combined with the compelling requirement to diminish the time-to-market window have rendered microprocessors more prone to design bugs than ever before. As a result, silicon debug – the process of validating and debugging a new microprocessor design on its first silicon prototype chips – has evolved to a critical, time-consuming, and labour-demanding step in a chip's development flow [2]. The pressure on the debug team to deliver a correct design in the marketplace on time is higher than ever although the combination of correctness and timeliness seems almost infeasible given the complexity of modern microprocessor designs (the validation space is practically "infinite") and the available time-to-market windows [64] [72]. In fact, the share of silicon debug in chip's development cycle is steadily expanding, while design to debug team ratio has reached 2:1[50]. As a result, an efficient silicon debug approach that promptly detects and eliminates the design bugs before volume production can make the difference between success and failure of a microprocessor product [82].

Pre-silicon verification techniques are mainly based on simulation at different levels of abstraction [13]. Despite their maturity and the tremendous utilization of computing resources, the low simulation speed is the inherent weakness of this step. Simulation-based techniques are orders of magnitude slower than the actual processor speed. As a result, it is impossible to verify every test scenario prior to tape-out [62], since only short number of functional scenarios can be exercised, in a reasonable time, compare to the enormous validation space of a modern microprocessors. In fact, statistics show that 12% of design bugs slip into first silicon prototypes and almost 50% of microprocessor chips require extra unplanned tape-outs [4]. An ineffective silicon debug process easily leads to product delays or even product recalls and a severely tarnishing in the reputation for the company.

Silicon debug starts with the arrival of the first silicon prototypes and often continues well after a product has gone to volume production. A comprehensive suite of test programs (such as automatically generated random test programs, legacy tests and real world applications) covering many test scenarios are executed on the prototype chips 24 hours per day for up to a year at various frequency, voltage, and temperature operating ranges to detect anything that may lead to incorrect operation: logic bugs, electrical or process-related bugs and mask-related manufacturing defects. Every time a bug is detected the debug team is fed with the failure data. Subsequently, for each failing test program (one that does not execute correctly due to a bug), separately, a systematic debug phase is performed by the debug engineers to identify the root cause of the failure. When a sufficient number of bugs are detected and fixed, a new batch of prototypes is manufactured and debug continues on the new samples.

Massive application of automatically generated random test programs (each consisting of a few thousand instructions) on the prototype microprocessor chips is one of the most effective parts of silicon debug [20]. All major microprocessor manufacturers have built efficient random test generators that produce trillions of test programs aiming to cover all possible test scenarios defined by the design and debug teams together. Despite its bug detection efficiency, this step is constrained by extreme computing needs for random tests simulation to extract the bug-free memory image for comparison with the actual silicon image. Another major bottleneck in this phase is that large number of random test programs fail

due to the same or similar design bugs [62] [61] [99]. This redundant behaviour prolongs silicon debug phase since each failing random test program must be exclusively root-cause analysed, although it does not usually bring new debug information. Finally, volume production may be further prolonged due to bugs that lurk behind other bugs. These “blocking bugs” stall the execution of the subsequent tests, since no workaround exists for the initial bug and therefore additional re-spins are needed (new prototype chips must be manufactured without the initial bug).

This thesis introduces a silicon debug methodology for microprocessors with two major objectives: (a) increase the validation coverage by applying more tests to silicon prototypes; and (b) reduce debug time by triaging the redundant failing random test programs. The methodology does so by exploiting (1) the inherent diversity of microprocessor instruction sets (existence of equivalent ways to perform operations) to eliminate the very expensive and time consuming simulation step by employing self-checking tests; and (2) the property that allows hardware components to be deconfigured (virtually “turned off” without compromising microprocessor’s functional completeness) to bucketing the redundant failing test programs.

2.1 Silicon debug challenges

Effective silicon validation methods are needed to ensure that forthcoming architectures do not suffer from severe bug escapes due to the following challenges (Table 3). By effectively addressing these challenges the number of escaping bugs is expected to be reduced.

Simulation Limitations – Simulation offers excellent control and monitoring capabilities throughout the entire design, but the limited simulation throughput has always been a bottleneck in the microprocessors industry. Expensive server farms devote huge amounts of time and energy for simulation but only a small portion of the different modes of operation can be thoroughly excited before silicon. Table 4 summarizes the throughput of simulation, emulation and actual hardware execution [40].

Table 4: Compare simulation, emulation and silicon execution throughput.

Approach	Throughput (instructions/sec)
System simulation	~10^3
RTL simulation	10^1 – 10^3
Emulation	~10^5
FPGA prototyping	~10^6
Silicon	10^7 – 10^9

Validation tests applied to prototype chips range from random instruction tests (RIT) [20] to user applications [19]. The silicon debug phase that is based on RITs contributes tremendously to the detection of design bugs; 71% of the bugs in Intel’s Core™ 2 Duo found in first silicon prototypes are detected by RITs [20]. In a RIT-based validation a huge number of random instruction sequences (trillions of random instructions in total) are executed and aim to cover all possible architectural and micro-architectural scenarios defined by the programmer’s reference manual.

Unfortunately RIT-based silicon validation is tightly coupled with a necessary simulation step and thus suffers from the simulation throughput problems [91]. A

typical RIT-based validation flow involves the execution of tests on a golden reference model, which is often an instruction-level accurate model of the processor (an architectural simulator for example) to produce the correct (golden or expected) responses. Correct responses must be known to compare them with the actual prototype responses; in case of a mismatch, bug-hunting begins.

Effective silicon debug of future microprocessors must mitigate the simulation of random instruction tests to save time, resources, and budget while not limiting their bugs detection capability. *How does one know in a simulation-less method (without needing "golden responses") that a random test ran correctly in the silicon prototype?* The proposed methodology contributes to this challenge proposing a self-checking technique, i.e. one that does not require pre-simulated golden responses.

At this point, it is important to emphasize on a fundamental requirement that the proposed methodology adheres: keeping original random tests unmodified. Several methods for the generation of effective random instruction tests have evolved throughout the last years [4] [3] [15]. Important problems in random tests generation have been addressed, such as avoidance of the creation of fixed patterns, which introduce interference into the test. Interference can spoil the test scenario and may hinder creation of more general patterns, thus reducing bug detection coverage. An example of interference is the case where a data processing operation (e.g. integer addition) is always paired with a data movement instruction (e.g. data loading). In this case, there is a high probability that the fixed instruction sequence spoils the contents of caches and leads to cache misses that the validation scenario is not expecting. The proposed methodology leaves original RITs unmodified to fully utilize their bug detection capabilities.

Redundant random test programs – The wide-spread adoption of random instruction test generation methods by the main industry players proves the importance of this phase of silicon debug. However, the random nature of the test suites results in the generation of multiple test programs that actually detect the same or similar design bugs [62] [61] [99]. While the same debugging information is shared among all these failing test programs, in the traditional RIT-based flow the debug engineers need to analyse each of them separately, wasting valuable human effort as well as other project resources such as compute time in high performance workstations used for design simulation. Thus, it is crucial to eliminate or reduce the "dirty" or "misleading" debugging records inside them and cluster common failure modes more effectively before the debug process begins. *How can the random tests execution in a self-checking method provide more useful validation data to the debug engineers?* This thesis contributes to this challenge by proposing a hardware mechanism to triage the failing random test programs along with the detailed information about the offending instructions.

Blocking Bugs – Another major issue of silicon debug is dealing with blocking bugs. Mostly in the first stages of silicon validation (first prototypes), there are several bugs with blocking behaviour [60]. A bug is a blocking one, if it can potentially mask out the discovery of other bugs, by stalling the execution of the subsequent tests (the rest of the debug plan goes wasted), because no workaround is possible for that bug. In such a case, silicon debug proceeds only after bug-fixing, re-design, and new silicon prototypes arrive to the debug teams [32]. Volume production may be seriously delayed and the overall development cycle and time-to-market will be prolonged if multiple such silicon re-spins are necessary before a design is considered sufficiently bug-free. *How can a self-*

checking RIT-based method help reducing the effect of blocking bugs? This thesis contributes to this challenge by providing equivalent instructions workarounds for the offending instructions.

2.2 Self-checking validation by exploiting ISA diversity

Microprocessor design validation is a time consuming and costly task that tends to be a bottleneck in the release of new architectures. The validation step that detects the vast majority of design bugs is the one that stresses the silicon prototypes by applying huge numbers of random test programs. Despite its bug detection capability, this step is constrained by extreme computing needs for random tests simulation to extract the bug-free memory image for comparison with the actual silicon image.

2.2.1 Scope of the self-checking validation

The proposed methodology is applied during the silicon debug phase of the microprocessor dependability cycle for the detection of logic and electrical bugs. The proposed approach aims to detect failures in silicon prototypes through the comparison of equivalent instructions responses. Furthermore, it refines the validation data provided to the debug process, by replaying the failing random tests. The contributions of the proposed methodology are the following:

1. A novel methodology for the generation of enhanced random instruction tests able to detect design bugs by comparing the results of equivalent instruction sequences is introduced. The methodology is therefore self-checking (does not need golden responses to compare with). A bug can make either an instruction or its equivalent to fail but a mismatch in the comparison denotes the existence of a bug in either case. Bugs can escape only when they affect the equivalent instructions in the same way, which is an extremely unlikely case. By generating equivalent instructions that do not activate the same hardware areas in the processor logic we minimize this probability. The identification of equivalent instruction sequences is a key enabler for the execution of subsequent random tests despite the existence of bugs, so our method inherently supports bypassing of blocking bugs. When an offending instruction is identified and while debug engineers look for fixes before new prototypes are produced, RIT-based validation can continue normally by avoiding the use of the problematic instruction and replacing it with its equivalent.
2. A light-weight hardware mechanism that records the mismatch between the results of two equivalent instructions to support subsequent identification of the offending instruction is proposed. Furthermore, the hardware mechanism contributes to the reduction of validation data forwarded to the debug engineers. After it records the mismatch location, the mechanism replays the RIT replacing the offending instruction by its equivalent sequence to take full advantage of the RIT's failure detection capabilities: it allows continuation of the RIT execution and identification of additional failures (thus avoids blocking bugs). By utilizing the information about the exact location of mismatches, the debug engineer can identify instructions or instruction classes that fail often and focus root cause analysis to particular structures of the microprocessor. Note

that the hardware mechanism is complementary to the bug detection method (checking of equivalent instructions) and is optional.

We evaluate the methodology experimentally on an x86-64 model with a comprehensive bug injection campaign. Our methodology successfully detected all injected bugs in the experiments while the traditional RIT-based approach and the two self-checking methods we compared with failed to do so. Furthermore, the proposed methodology accelerates the silicon debug process, compared to the traditional simulation-based flow, by increasing the prototype utilization. To do so, it introduces the concept of ISA-diversity, self-checking test programs to mitigate the simulation bottleneck.

2.2.2 ISA Diversity analysis

The bug detection philosophy is based on the existence of inherent equivalences (i.e. diversity) within modern instruction sets. ISA diversity is the extent to which operations of an ISA can be performed equivalently by more than one different ways. If the same input data are applied to equivalent instructions or instruction sequences, they will produce identical results, although they activate different logic paths in the processor logic. It is exactly this activation of different parts of the processor that enables bug detection by comparison (self-checking).

To identify the extent of ISA diversity in microprocessors, an analysis on four popular instruction set architectures: ARM, MIPS, PowerPC, and x86 along with examples of diversity and statistics for each ISA is presented in following section.

2.2.3 Diversity examples

Table 5 to 8 list examples of equivalent instruction sequences for ARM, MIPS, PowerPC, and x86 ISAs, respectively. In most cases, more than one equivalent instruction sequences exist for each original instruction, but only one alternative appears in the following Tables. We use the actual assembly instruction mnemonics of each ISA to describe the equivalent code. For uniformity, we use the same generic names (RA, RB, RC, etc.) for general purpose registers. Note that whenever an equivalent code modifies a register – which is not affected by the original instruction – its value has to be saved before and restored after the execution of the equivalent code (the save and restore instructions are omitted).

Table 5: ARM ISA diversity.

Original Instruction	Equivalent Sequence	Description

mvn RA, RB
move not
	sub RC, RC, RC
	sub RC,RC,#1
	eor RA, RC, RB
		Uses exclusive OR operation and an all 1's mask stored in RC to invert the bits of RB.

mlas RA, RB, RC, RD *multiply and accumulate*	mul RA, RB, RC adds RA, RA, RD	Splits the complex operation into a multiplication and an addition.
smuad RA, RB, RC *dual 16-bit signed multiply with add*	smulbb RA, RB, RC smultt RD, RB, RC add RA, RA, RD	Executes two signed 16-bit multiplications in the bottom and top halves of the source registers (RB, RC); then adds the intermediate products.
stmia RA!, {RB-RD} *store multiple increment after*	str RB, [RA] str RC, [RA, #4]! str RD, [RA, #4]!	Executes multiple single register store instructions which (except the first one) update the index.

Table 6: MIPS ISA diversity.

Original Instruction	Equivalent Sequence	Description
slt RA, RB, RC *set on less than*	sub RD, RB, RC srl RA, RD, 31	Executes subtraction and checks if the result is negative (R0 = zero register).
lw RA, addr(RB) *load word*	lhu RA, addr(RB) lhu RC, (addr+2)(RB) sll RC, RC, 16 or RA, RA, RC	Executes two load half-word unsigned instructions and places the second half-word to upper bytes.

lui RA, imm_{16}
load upper immediate
	addi RB, R0, 0xFFFF
	and RA, RA, RB
	addi RB, R0, imm_{16}
	sll RB, RB, 16
	or RA, RA, RB
		Resets the upper half of RA and then uses add and shift instructions to load the constant (R0 = zero register).

srlv RA, RB, RC
s*hift word right*
	rotrv RA, RB, RC
	sub RC, R0, RC
	addi RC, RC, 32
	sllv RB, RB, RC
	xor RA, RA, RB
		Rotates right instruction and then left shift to mask upper bits in the rotated result (R0 = zero register).

Table 7: PowerPC ISA diversity.

Original Instruction	Equivalent Sequence	Description
eqv RA, RB, RC *equivalent*	andc RD, RB, RC andc RE, RC, RB nor RA, RD, RE	Executes the logic operation: ¬ ((RB & ¬RC) | (¬RB & RC))
rldimi RA, RB, SH, MB *rotate left doubleword immediate then mask insert*	rldicr RC, RB, SH,63-SH clrldi RC, RC, MB rldicl RB, RB, MB,63-SH rotrdi RB, RB, MB or RA, RB, RC	Performs successive rotate and mask operations using other rotate instructions.

lwaux RA, RB, RC
load word algebraic with update indexed
	lwzx RA, RB, RC
	add RB, RB, RC
	extsw RA, RA
		Loads word first, then update RB with the new address and finally extend sign.

cntlzd RA, RB
count leading zeros
	addi RA, 0, 0
	Loop: add RA, RA, 1
	rldcl RC, RB, RA, 63
	beq Loop
	addi RA, RA, -1
		Implements a loop that rotates and masks operations to count leading zeros. Each iteration rotates RB left by RA locations and clears the 63 upper bits (and update flags). If the result is zero continues, otherwise exits.

Table 8: x86 ISA diversity.

Original Instruction	Equivalent Sequence	Description

add RA, [m32]
integer addition
	fild [m32]
	mov m32, RA
	fiadd [m32]
	fistp m32
	mov RA, [m32]
		Moves data from integer register file to FP-stack and uses the FP addition instead of integer addition.

mov RA, [m32] *load data from memory into register*	push [m32] add ESP, 0x4	Next instruction that uses RA operand should load the data from the stack. Restore stack pointer (ESP).
clc *clear carry flag*	mov RA, 0x0 bts RA, 0x0	Sets RA to zero and performs a bit test and set instruction which clears the carry flag.

jmp target
jump to target address
	mov RA, 0x1
	cmp RA, 0x1
	je target
		Sets RA to a value and performs a compare instruction which activates the ZF flag. The conditional jump (je) is then used instead of jmp.

cvtdq2pd RA, [m64] *convert packed dword integers to packed double-precision FP values*	cvtsi2sd RA, [$m64_{low}$] cvtsi2sd RB, [$m64_{high}$] movlpd m64, RA movhpd m64, RB movlpd RA, [m64] movhpd RA, [m64]	Executes two convert dword integer to scalar double precision FP values instructions followed by two load operations. The intermediate results (low 32-bits, high 32-bits) are merged into the same register.

fadd [m64]
floating-point addition
	movlpd RA, [m64]
	fst m64
	movhpd RA, [m64]
	mov $m64_{low}$, 0x0
	mov $m64_{high}$, 0x0
	haddpd RA, [m128]
	movlpd m64, RA
	fld [m64]
		$m64_{low}$ and $m64_{high}$ are consecutive memory addresses filled with zeros. Modifies operands values and replaces the floating point operation by a packed double-fp horizontal addition.

2.2.4 Diversity Statistics

We classify the instructions of the ARM, MIPS, PowerPC, and x86 ISAs in 3 categories.

- *Full Equivalence*: instructions for which there are one or more equivalent ways to realize their operation. This category includes the vast majority of arithmetic and logic instructions, data transfer instructions, compare instructions, and a large number of control flow instructions.
- *Partial Equivalence*: instructions which cannot completely diversified. No equivalent way exists to mimic entirely the original operation. This is due to: (i) different modes of operation for these instructions some of which

cannot be realized differently (e.g. various addressing modes), (ii) inherent loss of accuracy in the operation of an instruction (e.g. floating-point conversions). This category includes some of the instructions not included in the previous category, a number of floating point instructions, and some data transfer instructions that involve system storage areas.

- *No Equivalence*: instructions with no equivalences. This category includes mainly the privileged instructions that access system resources, complex control operations, input and output instructions, interrupts, exceptions and complex arithmetic instructions (mainly in CISC architectures).

Figure 7 shows the statistics of our analysis for the four popular instruction set architectures where all instructions are classified in the three categories. For each ISA, our statistics present the percentage of different instruction mnemonics that fall in each category and not the different opcodes. In many cases (particularly in x86) the same mnemonic includes several tens of different opcodes. For the x86 ISA, we focused on the general purpose instructions set and not the special ISA extensions (our intuition is that a study on the entire x86 instruction set, includes tens of thousands of instructions, will increase the amount of instruction on the full equivalence). It is evident from Figure 7 that all four ISAs have a large amount of instructions in the full equivalence category.

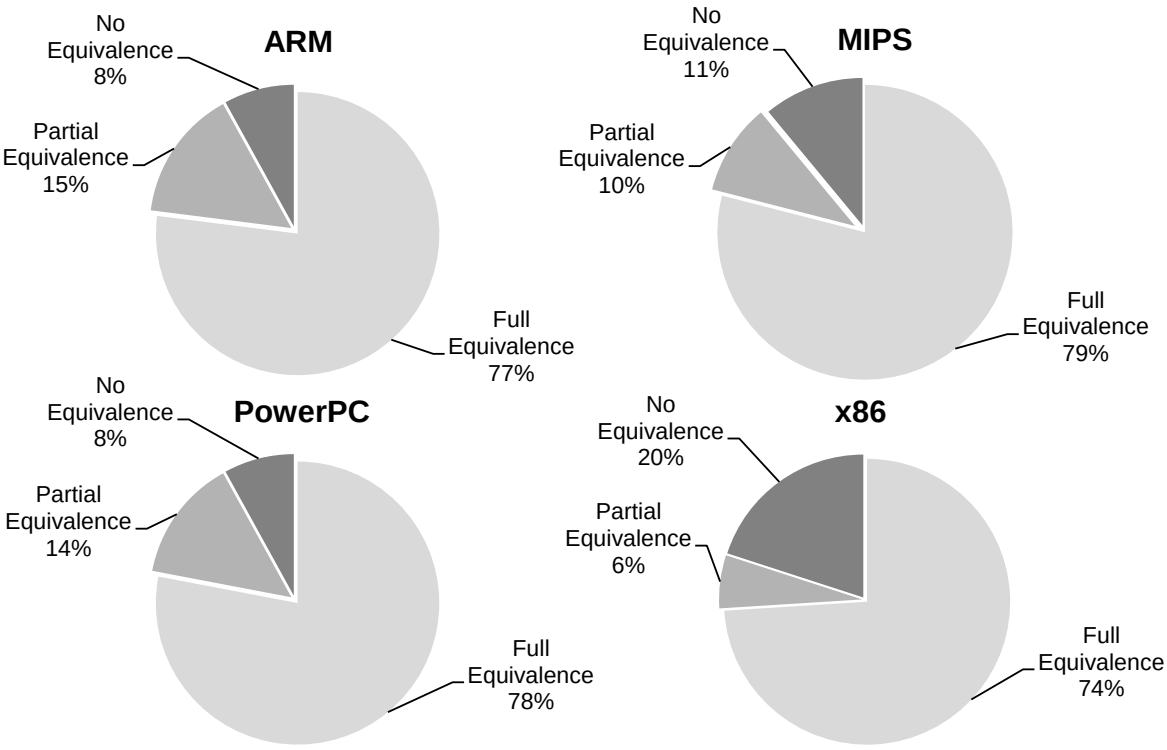


Figure 7: Diversity statistics of four popular ISAs.

2.2.5 Self-checking architecture

The proposed self-checking silicon debug methodology consists of four stages:

1) Generation of the ISA diversity database.
2) Generation of enhanced random instruction tests.
3) Hardware replay mechanism.
4) Post-processing.

We propose a novel, self-checking, diversity-based, hardware supported framework to accelerate silicon validation and improve its quality. An overview of the framework is shown in Figure 8, where our major contributions are highlighted. A detailed analysis of each feature of the methodology follows.

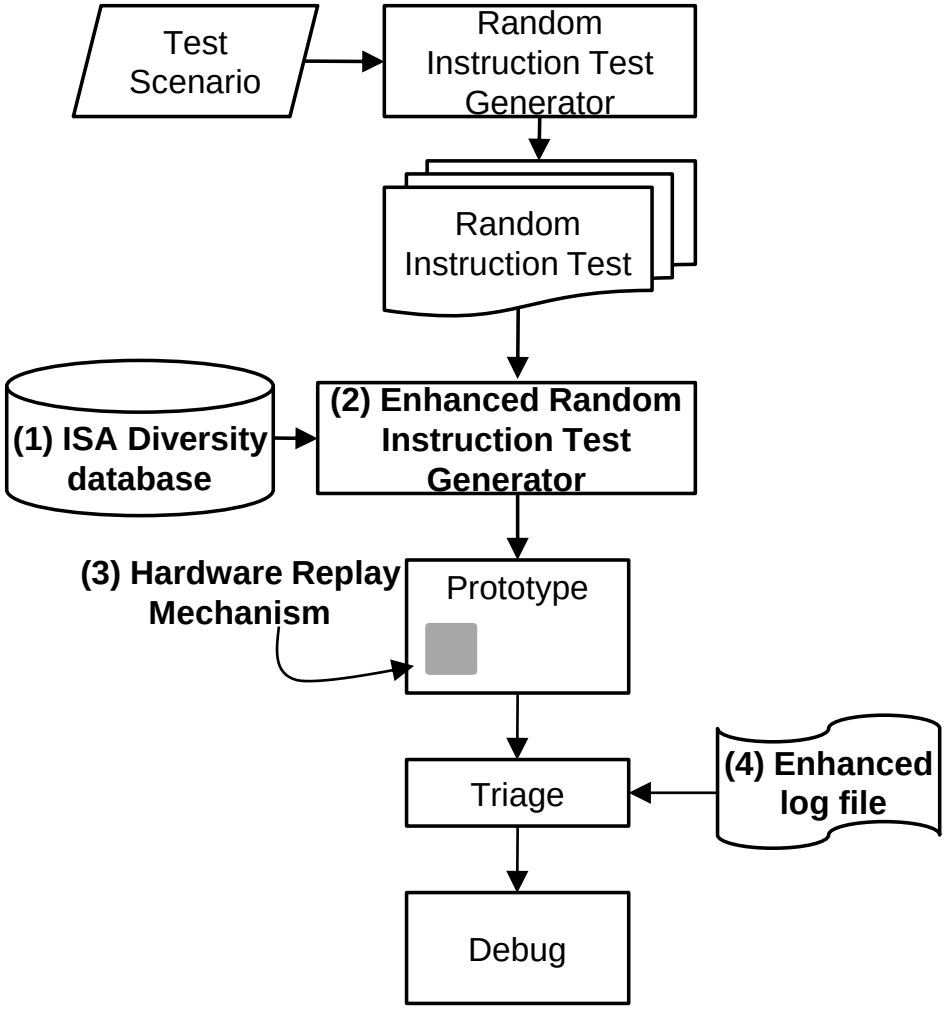


Figure 8: The proposed silicon debug framework.

(1) ISA Diversity Database Generation: The fundamental first step of the proposed methodology is the identification of ISA diversity, i.e. microprocessor instruction equivalences. Identification of equivalent instruction sequences and population of the ISA diversity database strongly depends on the designer's knowledge about the underlying architecture (detailed knowledge of the architecture, micro-architecture and microcode of the design). For this reason it is very likely to provide high quality results given the insights that the design team has on the microprocessor architectural details. The database contains for each instruction a list of equivalent instructions or instruction sequences.

(2) Generation of Enhanced Random Instruction Tests. The validation flow is fed with effective Random Instruction Tests (RITs) already generated (but not simulated) by sophisticated random generators that all microprocessor companies internally use. We pair each RIT with an Equivalent RIT (ERIT), to implement our basic bug detection concept: bugs are detected in our method by comparing the execution results of a RIT and its ERIT (a mismatch indicates a potential silicon bug). An ERIT is automatically generated from a RIT replacing its instructions with their equivalent counterparts that have been stored in the ISA diversity database. When the database contains more than one entry with

equivalent instructions for an original instruction, our approach randomly picks one of the alternatives. Instructions without equivalents are simply duplicated (in this case, only electrical bugs related to these instructions can be detected, since a logic bug will uniformly affect both equivalent ways to perform the particular operation).

Figure 9 presents the structure of the enhanced RITs that our methodology automatically generates. Each enhanced RIT consists of the following:

a) An *original* RIT which is left unmodified; we assume each RIT consists of a few thousands of instruction cycles as reported in the literature (each RIT we use in our experimental evaluation consists of 4K instruction cycles).

b) An *equivalent* RIT (ERIT) generated as described above using the ISA diversity database. For each instruction in the RIT, an equivalent instruction or instruction sequence is inserted.

c) A *checking code* that compares the stored results of the original RIT and the equivalent RIT to identify mismatches as indications of potential silicon bugs.

```
/*Original RiT code starts here.*/
1.  add     RA, [m32]
2.  st      RA          //store_i
3.  mov     RB, [m32]
4.  st      RB          //store_i+1
...
/*Equivalent RIT (eRiT) code starts here.*/
5.  fild    [m32]
6.  mov     m32, RA
7.  fiadd   [m32]
8.  fistp   m32
9.  mov     RA, [m32]
10. st      RA          //estore_i
11. push    [m32]
12. add     ESP, 0x4
13. pop     RB
14. st      RB          //estore_i+1
...
/*Checking code starts here. Compare RiT to eRiT responses*/
15. cmp     [m32], [m32]    //store_i = estore_i
16. cmp     [m32], [m32]    //store_i+1 ≠ estore_i+1
...
```

Figure 9: The structure of an x86 assembly enhanced RIT.

Bug detection in our method takes place by recording mismatches during silicon execution (through the checking code); therefore our method provides immediate bug detection. On the contrary, in a typical RIT-based flow, mismatches (due to bugs) are only detected very late and off-line when dedicated servers (i.e. validation host machines) compare the memory dumps (i.e. memory locations that are accessed or modified by the test, register files and any other data structures that can be scanned out from the silicon prototype) of the actual silicon execution with the expected memory dump contents from simulation. Figure 10 outlines the bug detection concept for the traditional and the proposed RIT-based silicon validation flow for microprocessors.

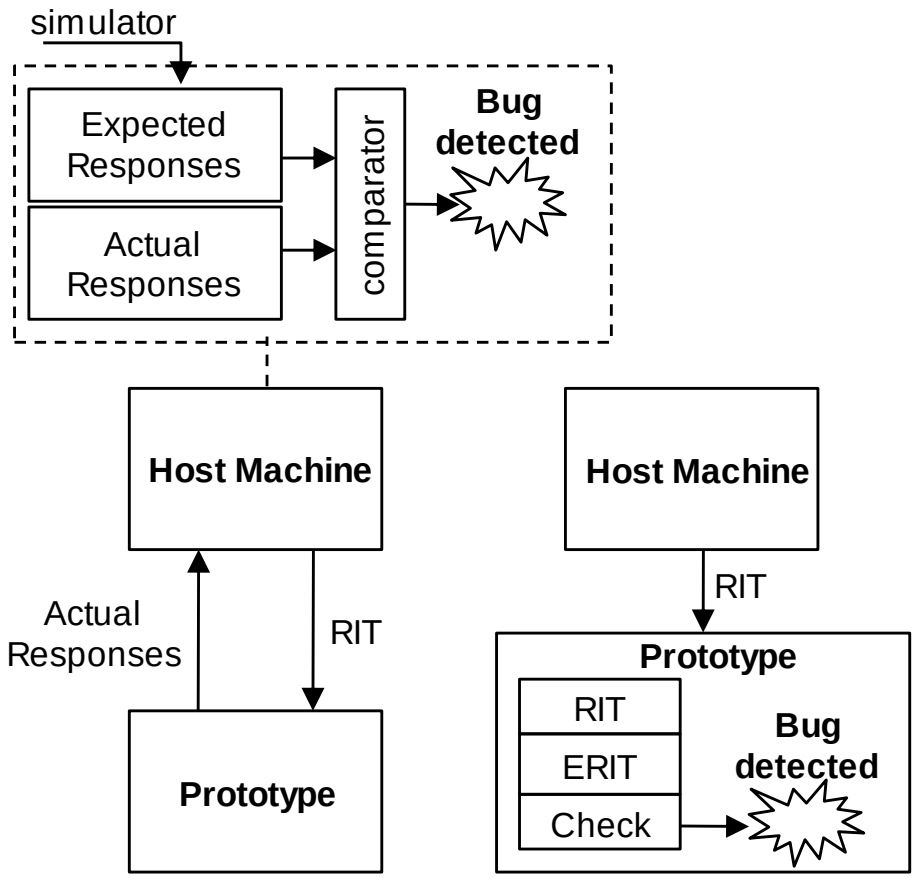


Figure 10: Traditional (left) vs. proposed (right) RIT-based validation flow.

(3) Hardware Support for Validation. In our method we take advantage of the fast bug detection that takes place during RIT execution on the prototype chip, and we support it with a hardware mechanism that is part of the microprocessor design (Figure 8). The hardware mechanism records the failing comparisons and points the execution points where mismatches happen. Moreover, when a mismatch is detected, the hardware mechanism allows *replay* (re-execution) of the RIT by replacing the execution of the offending instruction with its equivalent. If the offending instruction is the original one and its equivalent is bug-free the enhanced RIT replay produces the useful logging information we want. If this is not the case (i.e. the equivalent instruction is the offending one) the enhanced RIT replay does not produce useful logging information. Bugs in the instructions of the ERIT will be identified subsequently by other RITs that are explicitly generated to test them.

With our hardware replay (Figure 11), the test can continue execution and more bugs can be detected. Replay can happen several times for a single enhanced RIT as long as it detects more mismatches (potential bugs). In a typical RIT-based flow (without our modifications), after the first mismatch the remaining execution of the test is most probably useless since many subsequent responses are corrupted (since an output value of one operation can propagate to the input value of the subsequent operations). Other bugs that could possibly be detected by the remaining of the test are left undetected. In the replaying of the test using our hardware mechanism, the mismatch is bypassed, subsequent responses are not corrupted and if the remaining test can detect another mismatch (more bugs) it is allowed to do so. The last execution of the enhanced RIT is mismatch-free and detailed logging information is available for post-processing.

To demonstrate the hardware mechanism, we assume that an enhanced RIT (original RIT + ERIT + checking code) will be executed on the prototype chip. The original RIT includes *k* store instructions that write the results of the computation to the memory. For a typical RIT of ~4K instructions, it is realistic to assume that *k* is somewhere between 500 and 1000. Similarly, the ERIT includes *k* store instructions that write the results of the equivalent computations. The checking code compares the results stored by instruction store[i] and *estore*[*i*], where *store*[*i*] is the i^{th} store of the RIT and *estore*[*i*] is the i^{th} store of the ERIT, with *i* = 1…*k*.

The basic concept of the mechanism is that when a mismatch is detected between *store*[*i*] and estore[*i*], during replay, instead of executing the "buggy" code between *store*[*i–1*] and *store*[*i*], the processor executes the equivalent code between *estore*[*i–1*] and *estore*[*i*]. The mismatch has been just bypassed.

The hardware mechanism is shown in Figure 11 and operates as follows:

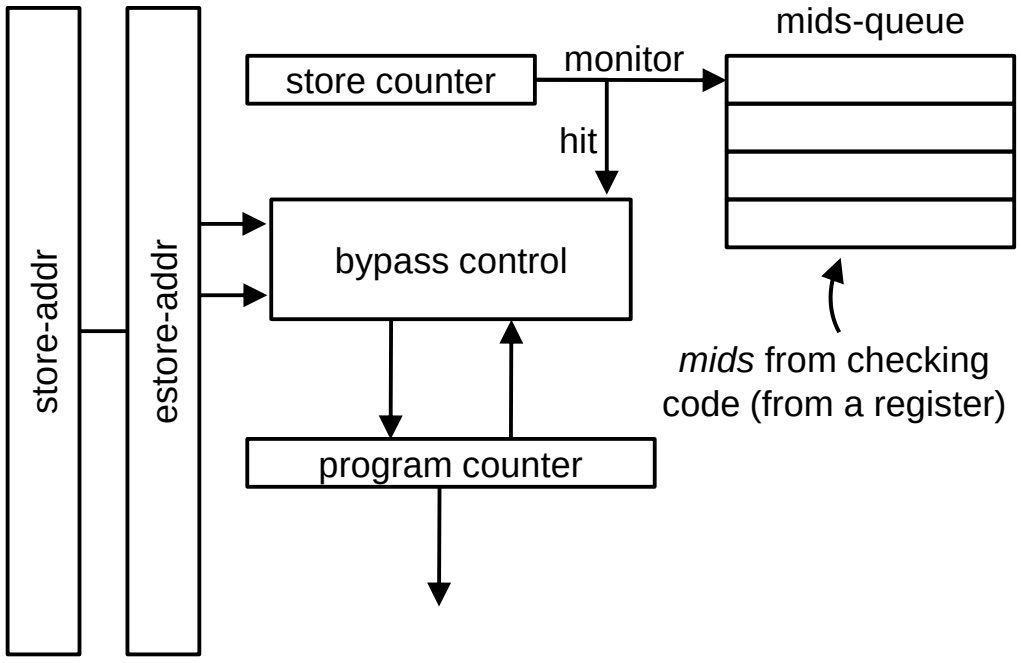


Figure 11: The structure of the proposed hardware mechanism.

First run of the enhanced RIT. The checking code finishes with the first mismatch among the *k* responses of RIT and the *k* responses of ERIT stored in variable *mid* (mismatch id), with *mid* between 0 and *k*. If *mid* = 0 (i.e. the queue is empty), then there is no mismatch and the chip passes the enhanced RIT; validation continues with the next RIT. If *mid* > 0 (i.e. the queue is not empty) the enhanced RIT will be replayed because *store*[*mid*] and *estore*[*mid*] instructions generated different results. Moreover, during the first run of the enhanced RIT the addresses of all store instructions (of the RIT and the ERIT) are saved in the *store-addr* and *estore-addr* buffers of the hardware mechanism to facilitate replay. Each of these two structures has a size of *k* words (addresses) for a total of *2k* words. The *store-addr* and *estore-addr* buffers record the address of the store instructions themselves and not the address they store the data to. The contents of the two buffers are needed to replay the enhanced RIT by manipulating the contents of the *program counter* – PC (or Instruction Pointer – IP) in hardware, as explained below.

Subsequent runs of the enhanced RIT. Every mismatch produced in previous

runs is stored in an entry of the *mids-queue*. The hardware mechanism counts the number of stores (monitoring instruction decoding) in the new run of the RIT in a *store-counter* (which is reset at the beginning of each run of the enhanced RIT). When store-counter gets equal to *mid* – 1 (i.e. before execution of the corrupted sequence) where *mid* is the mismatch id at the head of the *mids-queue* a "bypass" must happen (managed by the bypass control component), i.e. execution of the RIT code between *store*[*mid*–1] and *store*[*mid*] must be "replaced" by the execution of the ERIT code between *estore*[*mid*–1] and *estore*[*mid*] (this is why store-counter must be equal to *mid*–1). The "replacement" is done on-the-fly as follows using the store addresses information saved in buffers *store-addr* and *estore-addr*:

- After instruction *store*[*mid* –1] finishes, *PC* gets the address of the instruction following *estore*[*mid*–1].
- After the instruction before *estore*[*mid*] finishes, *PC* gets the address of instruction *store*[*mid*].

Therefore, instead of executing the "buggy" code between *store*[*mid*–1] and *store*[*mid*], the processor executes the equivalent code between *estore*[*mid*–1] and *estore*[*mid*]. The mismatch has been bypassed.

Table 9: The algorithm of the hardware replay mechanism.

```
inputs:   Set of RITs: original random instruction tests
          Set of ERITs: equivalent random instruction tests
output:   Log information {mids-queue}

for all RITs do
  execute RIT; save store addresses to store-addr buffer;
  execute ERIT; save store addresses to estore-addr buffer;
  execute checking code; compare RIT/ERIT responses;
  update mids-queue: add entry if mismatch found;
  if (mid = 0) then
    TestPassed;
  else
    while (mid > 0) do
      store-counter = 0;
      replay(RIT, ERIT);
        if (store-counter hits a mid – 1 in mids-queue) then
          PC ← estore[mid – 1] + 4;
          execute equivalent operation;
          PC ← store[mid];
        end if
        execute checking code; compare RIT/ERIT responses;
        update mids-queue: add entry if new mismatch found;
    end while
  end if
end for
```

During each replay run, this bypass process is repeated for each mid saved in the *mids-queue*. At the end of each "replay" run a new non-zero mid may be produced (in the checking code) and stored in the *mids-queue*. This means that in the subsequent replay execution one more bypassing will take place because the

RIT found one more mismatch. Eventually, the last “replay” (in which multiple bypasses took place) will produce a *mid* = 0 denoting a mismatch-free execution. Table 9 summarizes the operation of the hardware replay mechanism.

The logging information produced for the debug engineer is the queue of the mismatch identifiers *mids-queue*. For example, if at the end of the execution of an enhanced RIT (including all replays) the *mids-queue* contains integers 10, 25, 130, and 0, this means that mismatches have been detected between *store*[10] and *estore*[10], between *store*[25] and *estore*[25], and between *store*[130] and *estore*[130], and that the fourth “replay” was mismatch-free. With this logging information in hand, the debug engineer can directly locate the code portions with mismatches and focus on them for root-cause analysis. The debug engineer can also easily identify the RIT code replaced by the equivalent ERIT code for each bypass (it is of course the code before the stores with mismatches). We note also that the *mids-queue* contents must be also saved in memory so that after the end of the test they are passed as the methodology log to the debug team. Inside the hardware mechanism itself, *mid* information is passed by the checking code in one of the processor registers and the hardware mechanism records it in the *mids-queue*.

The size of the hardware mechanism depends on the size of the *store-addr* and *estore-addr* buffers (we assume a maximum size of 1,000 address entries for each buffer; equals to the average amount of store operations on 4K random test program), the size of the *mids-queue* (we assume a maximum of 10 entries; it is very unlikely for a single RIT to detect more bugs), the *store-counter*, and the bypass control logic that includes multiplexers and comparators. The hardware mechanism is deactivated after the end of silicon debug and thus it does not consume any power and does not affect performance in the field.

(4) Post-processing. As we mentioned earlier, it has been reported in the literature that the same bug can corrupt a large number of RITs. When the debug engineer is fed with many failing RIT memory dumps that are due to the same bug (in the same instruction, operation, or structure) the debug phase takes unnecessarily long time in order to determine if that failure is unique. This is an inherent inefficiency of traditional RIT-based flow since mismatches are detected much later in the server that compares the memory dumps. Our method offers a very important advantage to the post-processing phase: validation data provided by our hardware mechanism can help clustering of failure modes. This can be achieved through post-processing of the enhanced logs generated by the hardware mechanism (list of mismatch identifiers, *mids*, i.e. stores that saved different results to memory).

The list of mismatch identifiers (*mids*) is the log information our method provides. An integer *m* in the log (an entry in the *mids-queue*) means that: (a) the m^{th} pair of stores produced a mismatch, i.e. *store*[*m*] and *estore*[*m*] produced different results; (b) the code between *store*[*m*–1] and *store*[*m*] has been replaced by the code between *estore*[*m*–1] and *estore*[*m*] and the RIT continued. These two pieces of information can help the debug engineer identify the offending instructions and work on them.

Apart from the instruction bypassing realized by the hardware mechanism, our methodology and corresponding logging data provide a fast workaround solution necessary to allow validation to continue running subsequent RITs: buggy instructions can be avoided in subsequent RITs by using their bug-free equivalents from the ISA diversity database.

2.2.6 Experimental evaluation of the self-checking method

We assess our validation methodology by performing a comprehensive bug injection experiments campaign on a superscalar, out-of-order, single-core x86-compatible model in the PTLsim simulator [110]. PTLsim supports the full x86-64 instruction set of Pentium 4 (and subsequent), Athlon 64, and similar machines with all extensions (x86-64, SSE/SSE2/SSE3, MMX, x87).

For the experimental evaluation of our microprocessor validation methodology, we set up the tool chain outlined in Figure 12. Building a realistic experimental framework is an non-trivial task, since the available data regarding silicon debug models is limited. For instance, the insight on modelling real design bugs is limited due to the criticality of these data for the semiconductor industry.

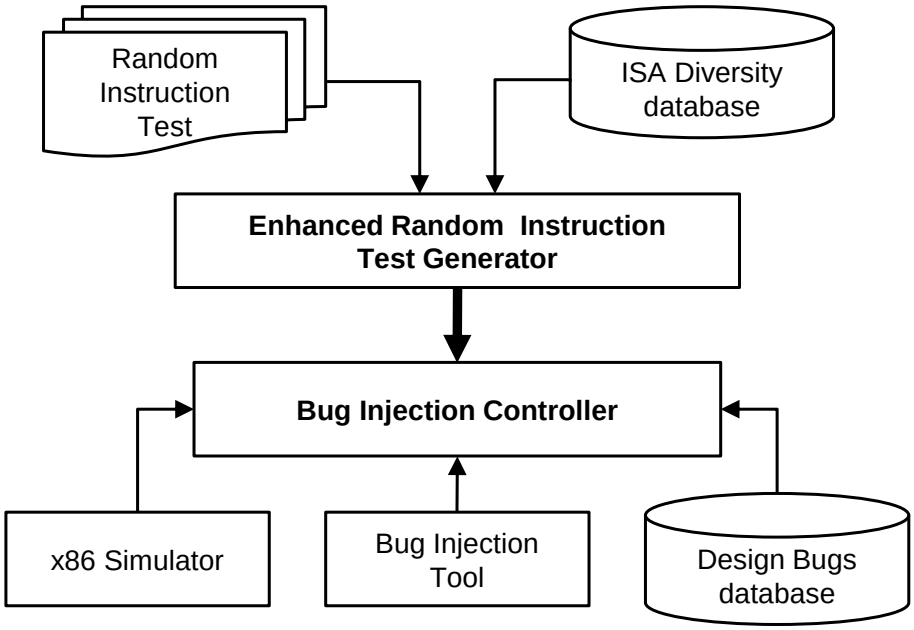


Figure 12: Experimental setup and tool-chain to evaluate the effectiveness of the self-checking methodology.

The experimental framework consists of the following:

a) The PTLsim architectural simulator for the x86 microprocessor.

b) Our RIT enhancement tool described previously that gets original RITs and produces enhanced RITs applying the equivalence-based methodology utilizing the ISA diversity database.

c) A bug injection tool that injects both logic and electrical bug at various locations throughout the entire processor. The bug injection tool uses our bug database which has been populated with bugs of either type.

For a given set of bugs in the design bugs database and a given set of enhanced RITs produced by our methodology, the experimental framework executes the enhanced RITs and records if a bug is detected or not.

We have injected both logic and electrical bugs to model different design bug conditions throughout the entire x86 architecture. For electrical bugs injection, we follow [70] which assumes that an effective and realistic way to model electrical bugs is to model them as transient bit flips at the microprocessor's flip-flops. In particular, on our experiments electrical bugs are models as a bit-flip (either for '0' to '1' or vice versa) in a random simulation clock cycle of a randomly selected memory element. On the other hand, logic bugs have a permanent effect and we

model them through a modification in the semantic correctness [102] of the architectural simulator's source code. Table 10 summarizes the types of the injected logic bugs.

Table 10: A list of logic bugs types that are injected into the simulator.

Semantic Modification	Correct Instance	Buggy Instance
Wrong operator	a = b **+** c	a = b **–** c
Wrong conditional statement	if (a **>** b)	if (a **≥** b)
Wrong signal assignment	a ← b + c	a ← d
Conceptual error	if (a > b) then a ← c	if (a > b) then a ← c + b
Wrong constant assignment	a = 0x000F	a = 0x0002

Table 11 presents a summary of the logic and electrical bugs we injected. In total, 1,025 design bugs were injected, 802 of them are logic and 223 are electrical, covering all pipeline stages and hardware components of the x86-64 microprocessor model. We injected electrical bugs mostly in the components that integrate large memory arrays, i.e. branch prediction unit, register file, etc. because the memory-dominated modules are more vulnerable to electrical bugs due their high density. The bit-flips are activated randomly in any position of a data structure. On the other hand, we injected logic bugs mostly in the control-related components where design errors in the complex conditional decisions are more likely to occur.

Table 11: Injected design bugs distribution in the components of the x86-64 processor model.

Pipeline Stage	Component	Logic bugs	Electrical bugs	Total bugs
Fetch/ Decode	Branch Predictor	71	16	**87**
	Prefetcher	29	12	**41**
	Instruction Decoder	100	–	**100**
	Microcode	62	–	**62**
	Instruction Buffer	–	18	**18**
Issue/ Execute	Integer Arithmetic	95	–	**95**
	FP Arithmetic	97	–	**97**
	Jump logic	46	–	**46**
	Load/Store logic	66	21	**87**
	Issue Queue	42	–	**42**
	Scheduler	32	–	**32**
	Register File	61	63	**124**
Retire	Reorder Buffer	101	41	**142**
Instruction & Data		–	52	**52**
	Total	**802**	**223**	**1025**

We compare our methodology with the traditional RIT-based validation flow. Moreover, we perform the same set of experiments for two other self-checking validation approaches presented in the literature that also aim to mitigate the time-consuming simulation step of RIT-based validation: (a) Reversi [105], according to which each instruction is followed by a reverse instruction; a bug is

detected when the final result is not equal to the initial one (i.e. it has not been reversed correctly); (b) QED or instruction duplication [48], according to which each instruction is duplicated and electrical bugs are detected when execution of duplicated instructions gives different results. For each of the three methods (Reversi, QED, and ours), we use the same original RIT as input and we enhance it according to the basic idea of each method.

Each of the original RIT sequences we use as an input consists of 4K instruction cycles and was generated by the tool generously provided to us by the authors of [105]. In our experiments we used 154 original RITs produced by the RIT generator, thus we ran a total of 616K random instructions in the simulator for each of the 1,025 injected bugs. Our RIT enhancement methodology increases, on average, the RIT code size by 6 times compared with the original RIT size (min = 4.3X, max = 9X, for the 154 RITs). The total number of instructions for the full campaign of 154 enhanced RITs (includes the original RITs, the equivalent RITs, and the checking code) is 3.7M of x86 instructions. The corresponding increase in RIT size by Reversi is on average 4 times (i.e. total number of instructions is approximately 2.5M instructions) and by QED is on average 3 times (i.e. total instructions is approximately 1.9M).

The results of our bug injection experiments are shown in Figure 13. Our methodology detects all 1,025 bugs injected into the simulator (bug coverage 100%) because we stopped generation of more RITs when all the injected bugs were detected. The traditional silicon validation flow detects 928 bugs (coverage 90.54%). This difference, against the proposed method, is explained by the activation of more hardware areas by the equivalent RIT. The instruction reversing method (Reversi) detects 903 bugs (coverage 88.10%) because there are cases where an instruction cannot be inverted. Furthermore, the flexibility of the ISA diversity concept to deploy equivalent instructions which activate totally different path in processor's logic provides us with the ability to avoid bug masking conditions (e.g. integer addition and subtraction happen on the same module, while in our method the equivalent addition take place on the floating point logic). Finally, the duplicated instructions approach (QED) detects 210 bugs (coverage 20.49%) because it can only detect electrical bugs, since a logic bug will act in an identical way in both original and duplicated instruction.

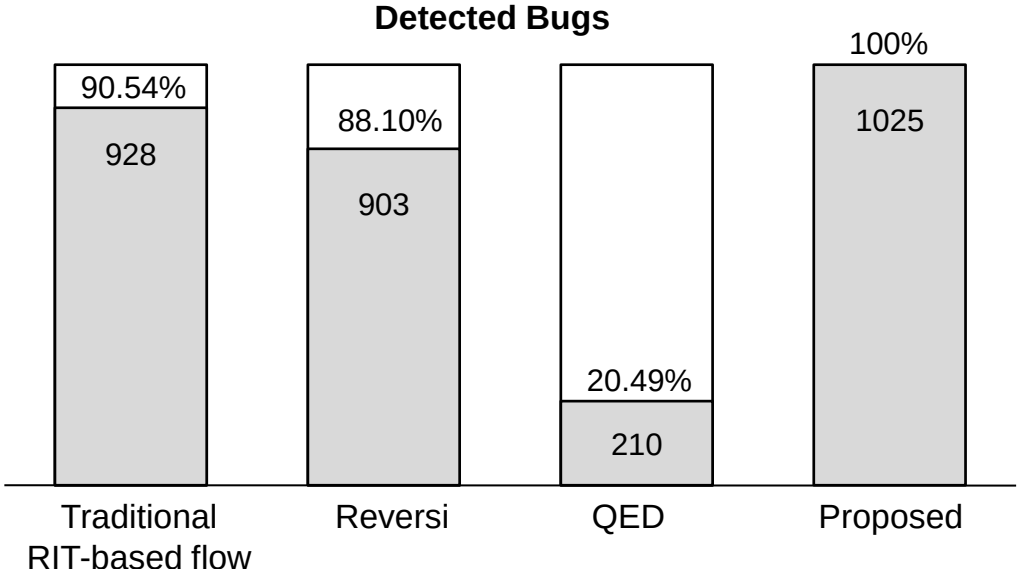


Figure 13: Design bugs coverage (1,025 bugs injected in total) for the four different methods.

For a complete validation plan (trillions of instructions), we expect our approach to have the same bug detection capability with the traditional RIT-based flow since our bug detection capability relies on the original RITs which are carefully generated by sophisticated industrial random generators. The advantage of our method is that mitigating the time-consuming simulation step it is able to apply more RITs and thus detect potential bugs earlier. Our methodology compares favourably with the other two self-checking approaches Reversi and QED. This is because Reversi is based on instructions whose effect can be reversed; this is not possible in many cases, and thus a number of bugs are not detected. QED on the other hand is based on instruction duplication and is very effective only for electrical bugs but not for logic ones.

Another advantage of the proposed method is that, it refines the validation data using the hardware replay mechanism. During our bug injection experiments, we observed that the average number of different bugs that were detected by a single RIT is about 4. Therefore, we integrated the hardware replay mechanism in the PTLsim simulator and conducted a second set of experiments: we injected all the bugs at the beginning of the simulation and executed all RITs with the highest bug detection capability. The proposed hardware mechanism detected all the injected bugs (through bypassing the offending instructions with their equivalents in the replay executions). This is a significant benefit of the proposed framework compared to the traditional flow which requires more tests to detect the same number of bugs.

Table 12 presents a comparison in terms of validation time (all timing measurements are on an Atom N270 with 1 GHz clock frequency) for the traditional RIT-based flow, Reversi, QED and the proposed method. Note that in silicon debug stage numerous (hundreds of millions for almost a year) random tests are generated; therefore Table 12: Validation times from the application of the traditional-based flow, Reversi, QED, and the proposed method. represents only a snapshot of the whole process. We discuss the different parts of the total validation time in the following.

Table 12: Validation times from the application of the traditional-based flow, Reversi, QED, and the proposed method.

Time (sec)	Traditional RIT	Reversi	QED	Proposed
Generation	4.460	6.310	5.530	7.680
Simulation	51.000	–	–	–
Execution	0.027	0.110	0.071	0.176
Total	55.487	6.420	5.601	7.856

In a typical microprocessor the total validation time of the first silicon prototypes consists of the following parts.

- *Generation time*: The time required to generate the random tests in the host machine. In our experiments, we generate 154 random tests, each one consisting of 4K instructions (summing up to 616K instructions for the traditional RIT flow, 2.4M instructions for Reversi, 1.8M instructions for QED, and 3.7M instructions for the proposed method).
- *Upload time*: The time required to upload the test from the host machine to the prototype for execution. This is typically performed through a

standard PCIe interface (or any other host debug interface). Given that the size of a random test is only a few kilobytes the PCIe throughput guarantees a nearly zero upload time and we do not include this time in Table 12.

- *Simulation time*: Only the traditional RIT-based flow needs to be simulated, since the other three approaches are self-checking.
- *Execution time*: The actual silicon execution time.
- *Download time*: The time required to download the responses (memory locations affected by the random test) of the test from the prototype to the host machine using the PCIe or other host interface. For the three self-checking methods the responses are downloaded only in the case of a failing RIT, while for the traditional RIT-based flow responses' downloading always follows each test execution. Since PCIe throughput guarantees a nearly zero download time for such small RITs, we don't include it in Table 12.
- *Compare time*: The time required to compare the actual responses with the expected ones (golden signatures). This is again very fast and we do not include it in Table 12.

The timing measurements demonstrate that the proposed method is much faster than the traditional RIT-based flow: more RITs can be applied in the same time. In addition, the longer test execution time of the proposed method compared to the two self-checking alternatives is due to the longer random tests it uses. However, this is alleviated by the improved bug detection capability of our method as shown in Figure 13.

Note that the speedup offered by our methodology by the mitigation of simulation applies only to the validation of instructions that have equivalents (more than three quarters of the ISAs). For the remaining instructions, the classic simulation-based RIT flow must be followed and thus our methodology is complementary to current industry practice.

Figure 14 roughly visualizes the timing of a traditional RIT-based flow and the timing of the proposed flow to give a clear idea of the timing advantages of the proposed method. In the host machine, we assume that generation (G) of random tests, uploading (U), simulation (S), downloading (D) and comparison (C) of the actual responses with the expected can take place in parallel. The prototype starts execution of legacy tests available from pre-silicon verification (or from previous architectures) and then executes the newly generated random tests. Although, in our experiments the upload, download and compare times are negligible because of the high throughput of PCIe interface, in Figure 14 we include them for demonstration purposes. The upload and download times can be significant if a slower interface than PCIe is used or if the size of the tests is much larger than a few kilobytes.

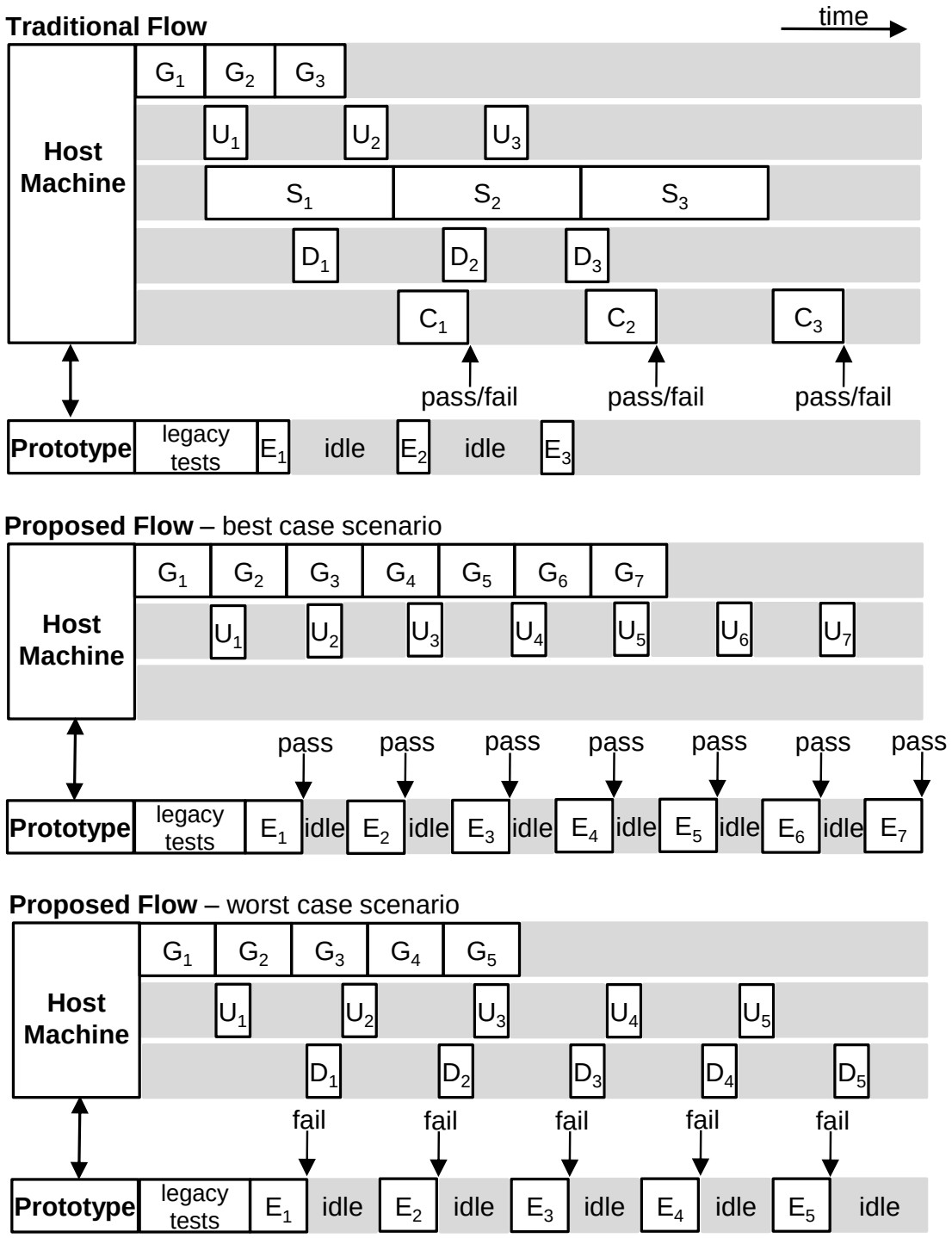


Figure 14: Traditional vs. proposed RIT-based silicon debug flow.

Figure 14 shows that the proposed methodology mitigates the simulation phase and also downloads only the responses of the failing RITs to the host machine. In a sense, the silicon prototypes are better utilized with our method and they execute more random tests during the same time. For example, Figure 14 shows that the proposed flow executes five random tests in the worst case (all failing) and seven random tests in the best case (all passing) while the traditional RIT-based flow executes only three. Thus, it accelerates silicon debug significantly.

2.3 Triage by exploiting deconfiguration ability

A critical step in silicon debug is *triage*, the process of analysing failing test programs and grouping them in “buckets” according to their failure mode. An effective triage process makes actual debug (the process of finding the root cause of the failure) much easier for the debug engineers and allows them to focus on dominant failure modes instead of spending expensive man power on test programs that fail due to the same underlying issue (the same bug).

Random test program flow constitutes a mature and efficient technique to detect bugs, however randomness is the inherent weakness of this part of the process. Random test program-based silicon debug results in the generation of many redundant test programs that fail due to the same or similar bugs. Every failing random test program consumes several hours or days of man and computational power. This debugging “noise” and overhead is expected to get worse in the future with the increasing design complexities. Clearly, the identification of dominant failure modes among the random test programs that can triage them into categories with common failure modes will significantly reduce the number of debug sessions and will therefore speed silicon debug up by several weeks or months.

The proposed methodology optimizes the triage process by exploiting the following property of many hardware components of microprocessors: a component can be “turned off” or deconfigured while the microprocessor remains functionally complete (i.e. processor’s baseline functionality is guaranteed despite the absence of the component).

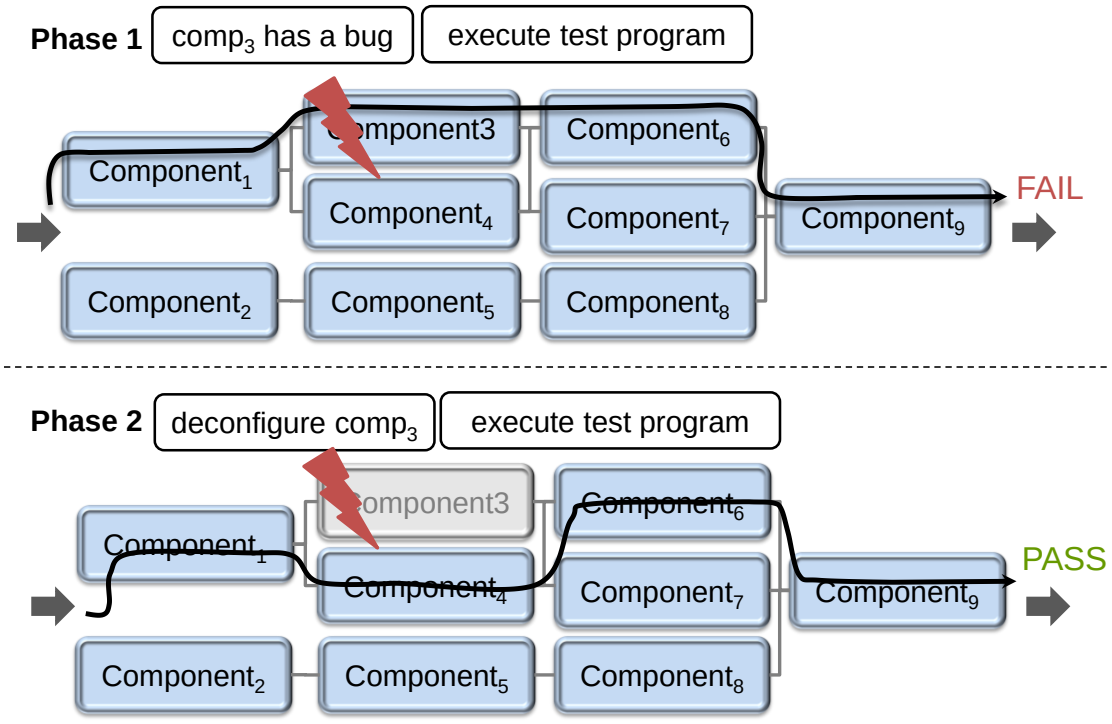


Figure 15: Silicon debug and component deconfiguration.

Our silicon debug methodology utilizes such deconfigurable microprocessor architectures and supports them by a dedicated hardware mechanism for dynamic deconfiguration of components during runtime. Random test programs are grouped in categories based on the set of components that need to be deconfigured for the test program to be correctly executed. Deconfigured components are pinpointed as potential hosts of bugs and just a few member of

each failing test programs category can be debugged for the identification of the failure root cause.

As a high-level quantitative example, Figure 16 outlines the issue of redundant test programs in a flow without (left) and with (right) a triage method. We assume that during a time interval a prototype chip executes 1 million random test programs. Among them, 1 out of 10.000 fails due to a design bug (failure rate) and 1 out 10 failing test program is redundant to another (redundant test program rate). The silicon debug flow without a triage method results in 100 debug sessions (one for each failing test program), while for the flow with a triage method, which is able to detect all the redundant test programs, the amount of debug session is reduced to only 10 (i.e. 100 failing test programs grouped into 10 categories).

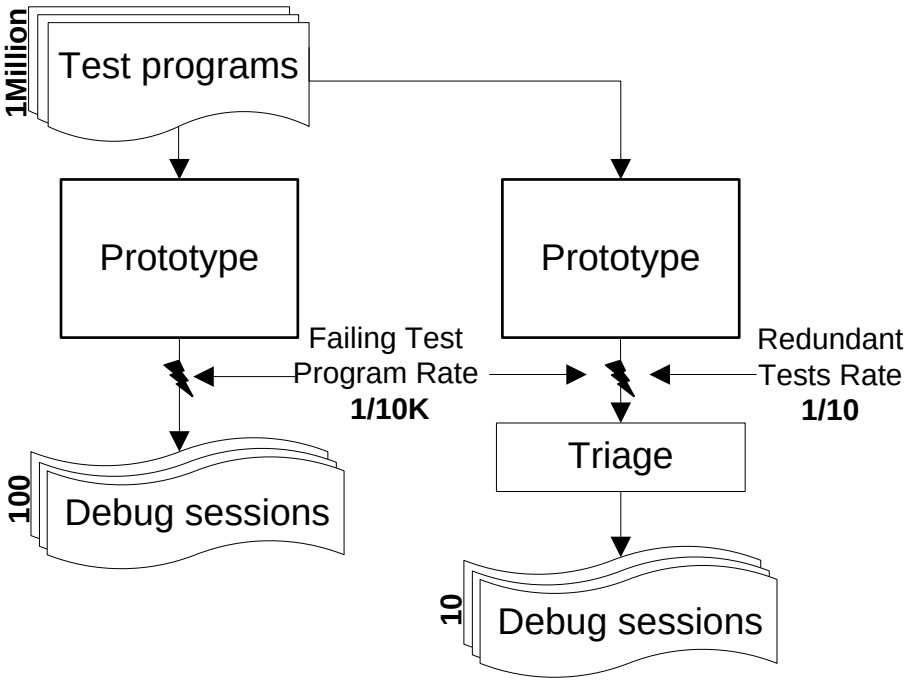


Figure 16: Redundant test program triaging concept.

As our experimental results section shows in detail (Section 2.3.7), we performed a set of experiments to calculate the degree of redundancy among the random test programs in a test suite. The experimental results show that on average 5 every 1000 failing random test programs detect the same design bug.

This thesis propose a silicon debug methodology for microprocessors with the major objective to automatically triage the failing random test programs of an overnight run in as small as possible number of “buckets” of failing tests with common failure modes. This triage obviously leads to less test programs to be debugged, and thus accelerates the overall silicon debug phase.

2.3.1 Scope of the triage methodology

The proposed methodology is applied at the silicon debug phase of the microprocessor dependability cycle for the triage of design bugs and aims to deliver a minimal set of failing tests groups after an overnight random programs campaign without manual intervention. The proposed methodology detects design bugs with the following characteristics: (a) their excitation does not depend on the operational conditions (temperature, voltage, frequency); and (b) they

continue to manifest themselves despite the deconfiguration of components from the overall design. The contributions of the proposed methodology to silicon debug are the following:

1. We propose the employment of deconfigurable microprocessor architectures along with self-checking random test programs to reduce the redundant debug sessions and make the triage step of silicon debug more efficient. Several hardware components of high performance microprocessor micro-architectures can be deconfigured while keeping the functional completeness of the design. This is the property we exploit in our silicon debug methodology for the triaging of random test programs.
2. We support our methodology by a hardware mechanism dedicated to silicon debug that groups the failing test programs into categories depending on the microprocessor hardware components that need to be deconfigured for a random test program to be correctly executed. Identical deconfiguration sequences for multiple test programs indicate the existence of redundancy among them and group them together. This grouping significantly reduces the number of failing tests that must be debugged afterwards.

The proposed methodology has been evaluated in an x86-64 microprocessor model of a publicly available architectural simulator. Experimental results prove both the validity of the claim that many random test programs fail due to the same bug, and also the large reduction in the debug time that is achieved by the effective triaging of failing tests using the proposed silicon debug methodology.

2.3.2 Microarchitectural transparency and deconfiguration opportunities

Throughout microprocessors evolution computer architects have devised numerous techniques to improve performance. Superscalar executions paths, multiple functional units, simultaneous multi-threading operating modes, out-of-order execution, dynamic scheduling, branch prediction, data and instruction prefetching are some examples of performance-enhancing mechanisms. All these techniques share a common attribute: microprocessor baseline functionality is guaranteed even without them. For instance, a core will continue to be functionally-complete even when its branch predictor is turned off or a cache memory bank is disabled. Therefore, the integration of these techniques in the hardware of a microprocessor is transparent to the instruction set architecture. Because of their sophisticated implementation such components are prone to design bugs. Our methodology aims to triage random test programs that fail due to bugs in such deconfigurable components.

The emergence of the previous techniques in modern microprocessor designs have also contributed to the evolution of Instruction Set Architectures. Sophisticated extensions of the instructions sets have been deployed, targeting to grasp the maximum performance speedup from the enhanced designs. These extensions are essentially built upon a basic set of primitive operations, such as arithmetic and logical operations and memory transactions. For example, the SIMD extension provides the ability to execute multiple arithmetic operations on a data vector, which is essentially comprised of a group of primitive operations. Therefore, equivalent instruction sequences exist (as this work has demonstrated so far), which can be used interchangeably to perform the same operation and at

the same time these modifications are transparent to the semantic correctness of an application.

It is evident that microprocessor architectures built around deconfigurable components which can be “turned off”, either directly by dedicated hardware at the micro-architecture level or indirectly by software through equivalent instruction sequences. Such deconfigurations will not compromise the functional-completeness of the microprocessor. We employ such architectures in this work to facilitate the triage step of the silicon debug process.

We studied Intel’s Nehalem micro-architecture [57] [58] to estimate the size of logic that is redundant to the baseline functionality of a processor and can be transparently deconfigured through hardware or software.

Table 13: Nehalem’s deconfigurable components.

Components	Size/Instances	Deconfiguration
L1 Instr. Cache	32KB	Hardware
L1 Data Cache	32KB	Hardware
L2 Cache	256KB	Hardware
First level iTLB	128 entries	Hardware
Second level iTLB	512 entries	Hardware
Instruction Queue	18 entries	Hardware
Branch Target Buffer	2	Hardware
Conditional Branch Predictor	2	Hardware
Return Address Stack	16 entries	Hardware
Simple Decoder (simple, frequent instructions)	3	Software
Complex Decoder (complex instructions)	1	Software
MS-ROM	1	Software
Instruction Decode Queue	28 entries	Hardware
Loop Stream Detector	1	Hardware
Micro-fusion	1	Hardware
Macro-fusion	1	Hardware
Reorder buffer	128 entries	Hardware
Reservation Stations	36 entries	Hardware
Integer Functional Units	9	Either
Floating Point Functional Units	3	Either
Load Buffer	48 entries	Hardware
Store Buffer	32 entries	Hardware

Table 13 contains the following information: the first column presents the components of the Nehalem core architecture that can be potentially deconfigured. The second column shows the component size in bytes or number of entries (e.g.: iTLB, caches), or the number of instances (e.g.: integer functional units). The last column states the way that a component can be deconfigured (hardware-, software-assisted or both). Singleton microprocessor components, modules for which no other candidate component exists to replace its functionality, can exclusively be deconfigured through the ISA diversity technique. It is evident that a very large number of Nehalem’s components, 35 in total, can be potentially deconfigured from the design.

It is evident from Table 13 that a very large number of microprocessor components, 35 in total, can be potentially deconfigured from the design either directly in hardware or indirectly through software or either way. Some storage elements cannot be completely deconfigured. A small part of them must remain enabled to guarantee the baseline functionality of the design. For example, a single entry store buffer, or a 4-entries instruction queue are enough to have a functionally complete design.

In this work, we focus on the hardware-based deconfiguration of the microprocessor components and the triage support it offers. Various [10] [106] [48] have developed self-checking random test programs that can be used for the detection of bugs in microprocessors. Any such method for the development of self-checking random programs can be employed in our methodology.

2.3.3 Triage methodology

The proposed triage methodology dynamically deconfigures several microprocessor modules during the execution of a failing random test program until it is correctly executed: i.e. the bug that causes the failure is masked by the deconfigurations. In particular, the triage methodology consists of the following steps:

1. A self-checking random test program is loaded for execution on the silicon prototype. The outputs of self-checking random test programs do not need to be compared with golden responses (from pre-silicon simulation) but rather generate a pass/fail indication at the end of their execution. This is a key requirement of the proposed methodology that facilitates re-execution of the test program without external intervention during uncontrolled overnight silicon debug runs.
2. If the test program fails, a hardware mechanism (deconfiguration controller) decides (based on a pre-defined sequence or dynamically) to deconfigure one of the deconfigurable components of the microprocessor.
3. The hardware mechanism arranges for the re-execution of the test program.
4. If the test program fails again, another deconfigurable component is “turned off” and the test program is re-executed.
5. Finally, if the test program is executed correctly (i.e. bug has been “masked” by the sequence of deconfigurations) the set of components that have been deconfigured is used as a label for a “bucket” of failing tests in the triage process. All test programs that eventually execute correctly after the same sequence of deconfigurations are grouped in the same “bucket”. Intuitively, the bug that causes the failure most probably resides within the components that have been deconfigured.

The main requirements for the proposed triage methodology to happen during an uncontrolled overnight run of huge numbers of random tests are the following:

- *Software requirement*: the random test programs must be self-checking so that the failure indication of a test is known to the hardware right at the end of its execution.
- *Hardware requirement*: the deconfigurable processor must be equipped with a mechanism which, in case of a failing test program, can: (i)

gradually (one at a time) “turn off” its components and (ii) re-execute the failing self-checking test program.

Alternatively, deconfiguration and re-execution can be partly implemented in software (by setting control register values). However, we focus on a hardware implementation because it can collect run-time information from hardware performance counters (existing or new) and it only requires a small part of the microprocessor (our deconfiguration controller explained below) to be bug-free.

The proposed deconfigurable architecture is outlined in Figure 17 and consists of the following elements:

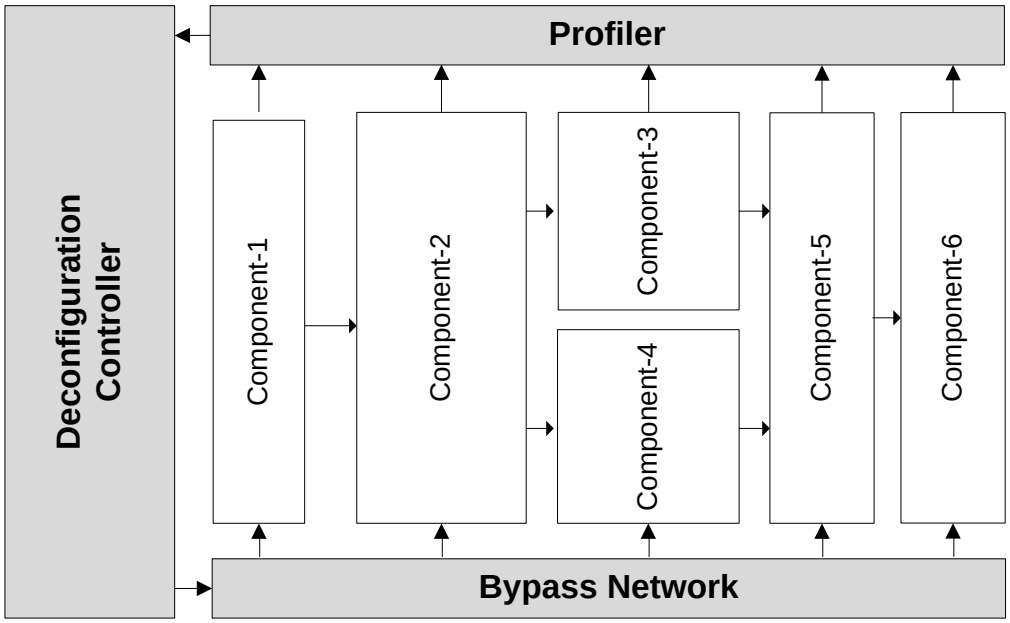


Figure 17: Proposed deconfigurable architecture.

- *Deconfiguration controller*. This is the main hardware element of the proposed architecture. It interfaces with all the deconfigurable components of the microprocessor (can be several tens of components as shown in the previous subsection) and takes dynamic decisions about the components to be deconfigured during each execution of a random test program.
- *Bypass network*. Controlled by the deconfiguration controller and performs the actual deconfiguration of the hardware components.
- *Profiler*. Each of the deconfigurable components communicates with this module which collects dynamic execution statistics of the random test programs to be considered in the deconfiguration actions. For example, for a memory element the number of write and read operations or for a functional unit the amount of activations can be suitable statistics.

After a self-checking random test fails (mechanism not shown in Figure 17), the deconfiguration controller takes the following actions:

- It selects the deconfigurable component that is considered most susceptible to contain a bug and turns it off. This decision is based on a bug susceptibility model discussed below.
- It arranges for the re-execution of the failing test program; no manual intervention is required.

- If the test program fails again, the deconfiguration controller repeats the previous two steps until the program executes correctly or there are no remaining components to be deconfigured.

The outcome of the operation of the deconfiguration controller for each self-checking random test program is a list of components that have been deconfigured. The interpretation of the list provides the following triage-related information.

1. *Empty list*. The random test program was correctly executed. No failure detected; no debug action required in the morning.
2. *List contains a subset of the deconfigurable components*. The random test program was correctly executed after components $\{C_k, C_n, C_m, C_q\}$ have been deconfigured. The list of components indicates a "bucket" of failing test programs. All test programs ending with the same list of deconfigurations are grouped together. The bug that is the root cause of the failures most likely resides within the deconfigured components. If a deconfigurable component contains more than one uncorrelated bugs the debug engineers will most probably diagnose them through during root cause analysis using simulation. Even if this does not happen, the execution of the subsequent test programs will detect and pinpoint the buggy component again.
3. *List contains all deconfigurable components*. The random test program fails even after all deconfigurable components are turned off. No triage grouping information; the random test must be separately debugged.

The *bug susceptibility model* on which the deconfiguration controller decisions are based is a flexible model that the silicon debug engineers can tune according to the stage of the silicon debug (early or late), the pre-silicon information available and the run-time statistics that can be collected by the profiler.

For each deconfigurable component C_i of the microprocessor, the deconfiguration controller calculates a bug susceptibility value S_i (higher value means a more bug-prone component). When a new component must be turned off before a failing test program is re-executed, the component with the highest Si value that has not been deconfigured yet, is selected and turned off. The information that the deconfiguration controller can use for the calculation of the Si values of the deconfigurable components is static or dynamic.

Static bug susceptibility information comes from the pre-silicon (simulation-based) debug process. When a microprocessor component C_i is new in a design or if a large number of design bugs have been already found before silicon debug (given that in pre-silicon verification each component has not been exhaustively studied due to the simulation throughput bottleneck, there is a higher probability that more design bugs exist in its design), the debug team can assign it a large bug susceptibility value S_i^{static}. Moreover, static bug susceptibility assignments can be based on the size of the components, on their complexity (larger and more complex components are more bug prone) etc., and the deconfiguration controller can be updated with new values before a new overnight run starts.

Dynamic bug susceptibility information is collected during the execution of random test programs by the profiler component of the proposed architecture. The information may contain activity monitors (from existing or new hardware performance counters or other signals of the design) that show if a component is intensively activated by the particular test program. If this is the case, this is a

useful indication that the component is more susceptible to contain the bug that causes the failure of the test program. Therefore, the dynamic bug susceptibility value $S_i^{dynamic}$ of the component C_i must be higher than others with smaller activity during the execution of the failing test program.

In total, the deconfiguration controller calculates an aggregate bug susceptibility value for each component that can be potentially deconfigured:

$$S_i = a * S_i^{static} + (1 - a) * S_i^{dynamic}$$

Parameter a can be tuned (values between 0 and 1) so that decisions lean more towards the static pre-silicon information (large *a* values) or more towards the dynamic run time statistics (small *a* values).

The component with the highest susceptibility value Si is selected to be turned off in the next re-execution of the failing random test program. Bug susceptibility is a metric that can be finely tuned, is based on both dynamic and static information, and bounds the amount of possible test programs re-executions avoiding useless deconfigurations of microprocessor components.

At the end of the repetitive re-executions the failing test is characterized by the set of components that have been deconfigured and have the highest bug susceptibility values. It is therefore, very likely, that the failure of the test is due to a bug inside these components. Triaging failing random test programs in “buckets” according to their list of deconfigured components provides useful insight to the debug teams that will start debugging the tests in the morning after an intensive overnight run.

2.3.4 Cost Implications of the Methodology

Before analysing the different deconfiguration mechanisms that can be employed in our methodology we discuss the cost implications of the methodology.

Costs related to the deconfiguration infrastructure of the architecture. The components listed in Table 13 (all very common to x86 architectures) can be potentially turned into deconfigurable ones so that the proposed silicon debug methodology is applied. Therefore, the extra hardware adds to the complexity of the design. In the case of some storage elements the need to keep at least some of their entries active while the rest of the component is deconfigured, adds further design modification costs. These hardware modifications come with a positive aspect: the existence of the deconfiguration infrastructure is an added value for the microprocessor because it can be used in the field for the permanent “shut down” of components when they are diagnosed with hard errors. Employment of deconfiguration (at different granularities) for fault tolerance in the field has been reported in the past [6] [22] [23] [84] [90] [97] [101]. Finally, the cost of deconfiguration depends also on the granularity it is applied. If subsets of entries of a component are separately deconfigured the cost may become high (both for the deconfiguration controller and the bypass network and profiler). We believe that the deconfiguration granularity given in Table 13 (common for x86 microprocessors) is suitable for the needs of silicon debug. If debug engineers are supplied with the information that a short list of 3-4 components have been deconfigured before a test is correctly executed, their debug job is much easier and focuses on the list of these components. In most cases, the list is expected to consist of just one component which is very likely the one with the bug.

Costs related to the dynamic collection of statistics. The profiler component dynamically collects run time statistics when random tests are executed. This

feature requires design effort and also increases the area of the microprocessor. If the design team decides to rely on the run time statistics the investment in the design of the profiler and the utilization of its outputs by the deconfiguration controller justifies the cost. However, if the debug team decides to use the deconfiguration controller with a priory known susceptibility values from the pre-silicon effort, the cost of the profiler is saved.

Timing overheads and time savings. The proposed methodology adds a time overhead to the random test phase of silicon debug: each failing random test is repeated one or more times. However, this minimal time overhead is absolutely justified by the large savings in the debugging time of the failing tests. For example, consider an overnight campaign of random test programs executions that is prolonged by a relatively small amount of time (measured in minutes or an hour for the entire campaign), required for the re-executions of the failing test programs. Even, assuming a large percentage of failing tests (1%) and an average number of re-executions equal to 5 (i.e. 5 components are deconfigured and a failing test program is repeated 5 times on average) the time overhead for the overnight run will be around 5% (1% times 5 re-executions). In other words, approximately 5% less random test programs will be applied during the night. Even in the worst case, where all available components have to be deconfigured (based on the study we perform on Nehalem's architecture, the total number of deconfigurable components is 35) the extra overhead from the re-executions remains small compared to the expected debug sessions savings. The successful application of the methodology will reduce the number of failing test programs that will need to be debugged from several thousands to just a few tens. This saving is measured in days or weeks of debug time and is the major contribution of the method to silicon debug.

2.3.5 Deconfigurable architecture

In this section, we review the mechanisms reported in the literature that can be employed to deconfigure the components of a high performance microprocessor. Furthermore, we discuss simple deconfiguration schemes for branch predictors and prefetchers. The deconfiguration granularity can be flexibly tuned and is only limited by the cost implications. Previous research [6] [22] [23] [84] [97] [101] proposed various techniques to deconfigure components with permanent faults from a microprocessor design.

The circularly-accessed arrays, the directly-accessed arrays and the functional units comprise the list of processor modules that are often duplicated or contain a high degree of regularity and can be deconfigured. The circularly-accessed arrays, such as the instruction fetch queue, the reorder buffer or the load and store queues, are augmented with a fault map. The fault map communicates with the pointer advancement logic, forcing it to skip an entry that is marked as faulty ("buggy" in silicon debug).

For the cases of functional components and directly-accessed memory arrays, the available deconfiguration solution is to mark the components or memory entries as permanently busy, preventing the microprocessor from issuing further requests to them.

In silicon debug the aforementioned deconfiguration techniques are clearly applicable. However, the deconfiguration granularity could be more coarse-grained (for example, deconfigure half of the reservation stations, or the entire L1 data cache and not parts of it), since localizing a bug at the level of micro-

architecture components carries enough information for the debug engineers to root cause the failure.

Modern microprocessors integrate numerous performance-increasing components. Among them, the branch predictors and the data and instruction prefetchers are the most widely used ones. These modules can be deconfigured from the design, since they do not contribute to its functional completeness. Regarding the branch predictors, a simple deconfiguration mechanism can be used, where the component can be bypassed by setting the predictor's state machines permanently to not-taken state. A similar deconfiguration mechanism can be applied to the data and instruction prefetchers. In particular, the prefetcher's queue, where all prefetching requests are buffered, can set permanently the overflow flag. Therefore, all prefetching requests will be dropped and the prefetcher is effectively deconfigured.

2.3.6 Deconfigurable controller design

The main structure of the proposed microprocessor architecture is the deconfiguration controller. The controller can calculate dynamically, at runtime, the susceptibility value of each deconfigurable module and decides the module for the next deconfiguration (i.e.: the module with the higher probability to be the source of the failure). Figure 18 outlines the structure of the deconfiguration controller in the general case where both static pre-silicon susceptibility information and dynamic run time susceptibility information is utilized.

The deconfiguration controller consists of three memory elements and a combinational part implementing the proposed deconfiguration model. The memory elements are implemented through the allocation of memory-mapped space in the main memory of the prototype. The deconfiguration controller accesses these structures for write/read operations.

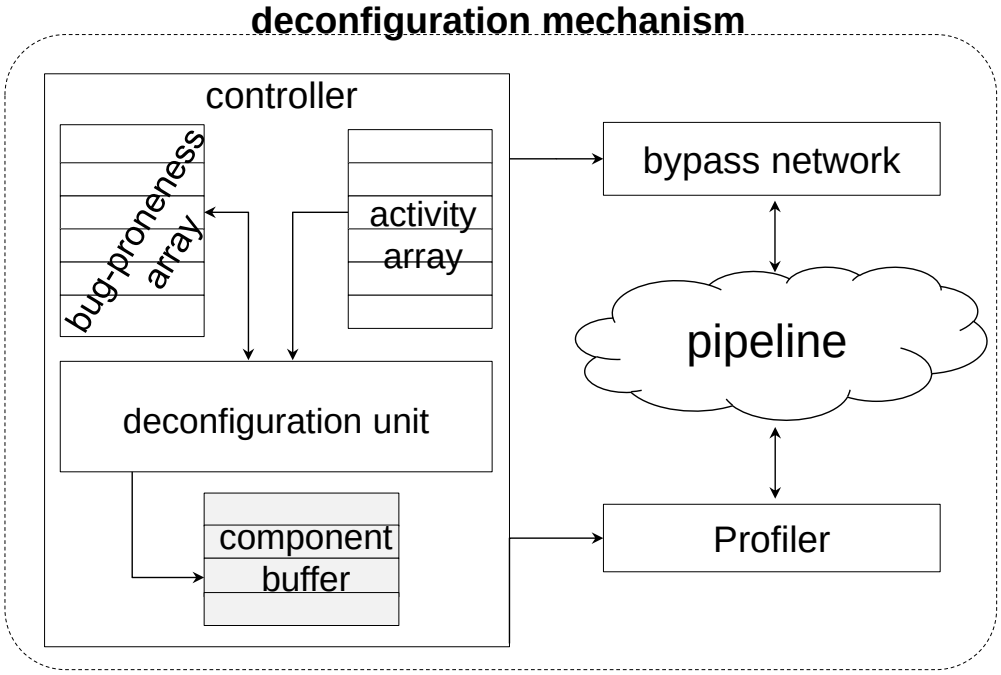


Figure 18: Deconfiguration controller block diagram.

A bug-proneness array is updated with the susceptibility value of each deconfigurable module as assessed from the designer from pre-silicon data. The array is updated at the initialization phase of the silicon prototype chip during test

program uploading. A single entry for each deconfigurable component is reserved in the bug-proneness array.

A second array, the activity array, is accessed by the profiler and stores the activity (existing or new performance counters and other signals values) of each deconfigurable module during execution of a test program. At the end of test program execution the utilization array is updated with the value of the performance counters. The activity array can contain more than one entry for each deconfigurable component depending on the counters and signals that need to be monitored for the component for a more elaborate decision at run time. Therefore the array may have a size of a few hundreds words.

The deconfiguration unit parses the two memory arrays (bug-proneness and activity) to find the component with the largest value of bug susceptibility Si as described previously. This component is assumed to be the one with the highest probability to be the source of the failure.

The deconfiguration unit output is written to the component buffer of the deconfiguration controller. Each entry of this array saves the id of the component that will be deconfigured. In every re-execution of the random test program the deconfiguration unit increases the pointer of the component buffer, writes the id of the next component to deconfigure and activates the relevant bypass logic.

At the end of the multiple hardware-enabled test program re-executions, the component buffer contents are downloaded along with the remaining memory image of the prototype on the workstation (a server that controls the validation process for a particular prototype) for further analysis in the morning. Before each re-execution of a test program the arrays are reset, since only the components that have not been already deconfigured should be considered in the estimation of the susceptibility model. Figure 19 visualizes the timeline of the operation of the proposed deconfiguration controller.

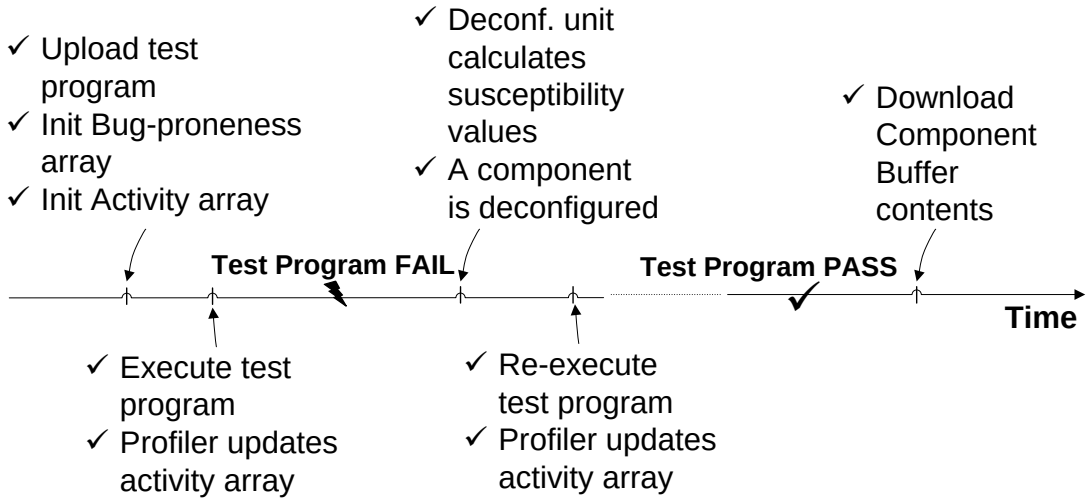


Figure 19: Methodology timeline for each test program.

2.3.7 Experimental evaluation of the triage mechanism

In this section we evaluate the proposed triage methodology. We first describe the experimental framework. Subsequently, we measure the degree of redundancy among the random test programs generated by our generator (which has been developed following guidelines that major microprocessor companies provide in the literature) for an x86-64 microprocessor using PTLsim architectural simulator [110]. Finally, using the same framework we demonstrate the benefits

of module deconfiguration in random test program triaging for silicon debug acceleration.

For the experimental evaluation of the proposed silicon debug methodology, we set up the tool chain shown in Figure 20 (dashed components are implemented from scratch to evaluate our methodology). The experimental framework consists of the following main modules:

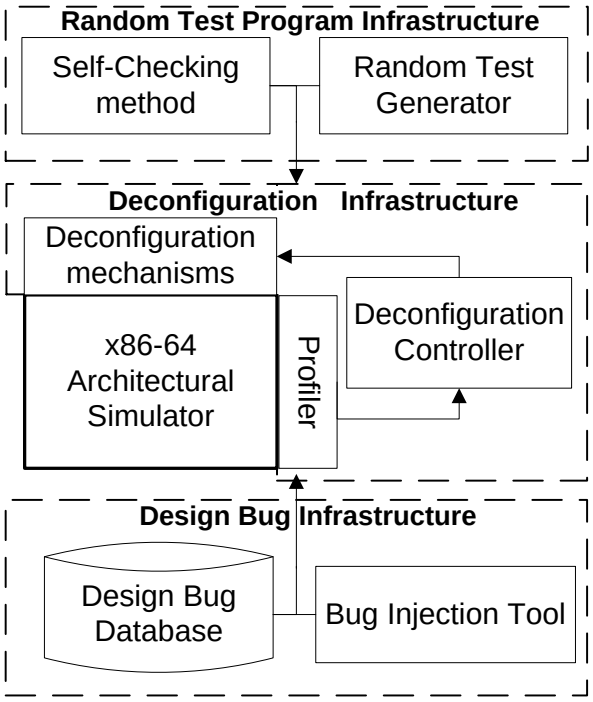


Figure 20: Experimental framework for the triage methodology.

Random Test Program Infrastructure: This module generates random, x86 assembly test programs enhanced with a self-checking capability (i.e. not needing golden responses to compare with in order to conclude about the detection of a design bug). We adopted the method presented in [10] which exploits the diversity property of microprocessor ISAs. The input of the random test program generator is a set with the basic x86 instruction templates. For example, the template of an addition operation is the following: add, <operand1>, [<operand2> | <memory>]. Registers selection, operands value initialization, data memory initialization and instructions sequence are completely randomized. The output of random test program generator is x86 assembly random test programs of 4K instructions each.

Design Bug Infrastructure: Similar to [10], the bug injection tool injects design bugs at various locations of an x86-64 superscalar, out-of-order, single-core design modelled through PTLsim architectural simulator (Figure 20). The design bug database is populated with a set of logical and electrical bugs that model different design bug conditions in the entire x86-64 architecture ([70] [102]). Table 14 summarizes the numbers of logic and electrical bugs injected in the components of the x86-64 microprocessor model. In total, 1K design bugs were injected, 500 logical and 500 electrical respectively, covering all pipeline stages and the majority of hardware components of the x86 microprocessor model.

Table 14: Injected design bugs distribution in the x86-64 microprocessor components.

Component	Electrical Bugs	Logic Bugs	Total
Instruction Fetch Queue	50	-	50
Branch Prediction Unit	50	50	100
Simple Decoder	-	100	100
Complex Decoder	-	100	100
Register Renaming	50	50	100
Issue Queues	100	-	100
Scheduler	-	100	100
Functional Unit	-	50	50
Load/Store Queue	100	-	100
Reorder buffer	150	50	200
Total	**500**	**500**	**1000**

Deconfiguration Infrastructure: The x86-64 microprocessor modelled in the PTLsim simulator integrates various structures that can be deconfigured while the processor remains functionally complete. Table 15 lists the deconfigurable components (first column) and the techniques from section 2.3.5 that have been used to implement the deconfiguration on the simulator (second column). The third column presents the initial size of each component, while the last column demonstrates the selected deconfiguration granularity. The deconfiguration granularity must not violate the basic functionality of the microprocessor. For example, the default size of instruction fetch queue is 32 entries (in our configuration) out of which 28 (at maximum) can be deconfigured (the baseline fetch width of the microprocessor must be guaranteed. In our case, the baseline fetch width is 4-way). As well as, only the redundant ALU can be also deconfigured .

Table 15: Deconfigurable microprocessor modules in the x86-64 model of the PTLsim simulator.

Component	Deconfigurable Mechanisms	Initial Size/ Entities	Deconfigurable Granularity
Instruction Fetch Queue	Fault-map	32 entries/–	28 entries altogether
Return Address Stack	Stuck-at	16 entries/–	16 entries altogether
Conditional Predictor	Stuck-at	–/2	2
Instruction Prefetcher	Stuck-at	–/1	1
Data Prefetcher	Stuck-at	–/1	1
Branch Target Buffer	Stuck-at	4K/–	4K entries altogether
Register Renaming Table	Fault-map	16x256	16x128
Issue Queue	Fault-map	16 entries	8 entries altogether
ALU	Busy mode	–/2	1

FPU	Busy mode	–/2	1
Load Queue	Fault-map	48 entries/–	44 entries altogether
Store Queue	Fault-map	32 entries/–	28 entries altogether
Re-order Buffer	Fault-map	128 entries/–	124 entries altogether

Our deconfiguration infrastructure in the experimental framework integrates a simple profiler component that monitors the activity of the deconfigurable modules and provides the dynamic bug susceptibility data to the deconfiguration controller. We have not implemented all details of the profiler because the analysis we provide in the following subsection does not depend on the type of bug susceptibility that the deconfiguration controller considers (static or dynamic). Future work can analyse the efficiency of different dynamic run time statistics collection by the profiler as well as their exact hardware costs.

The experimental evaluation of the proposed methodology is divided into two sets of experiments:

First set of experiments. The random test program infrastructure and the design bug infrastructure are used to quantify the degree of redundancy among the random test programs. We need this first set of experiments to support our claim about redundancy which is the main motivation of our work.

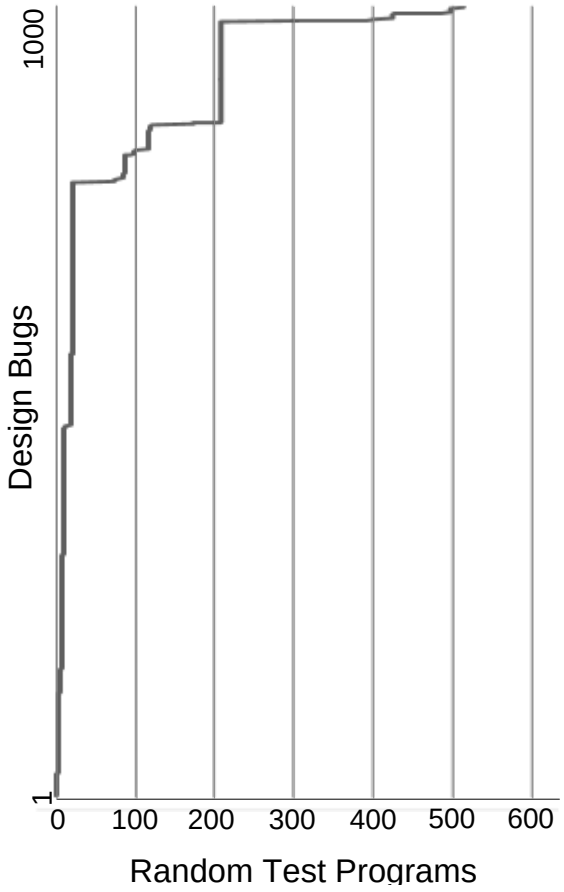


Figure 21: Number of failing test programs (among 10,000 executed) for each of the 1,000 injected design bugs.

For a given set of 1K design bugs defined in the design bug database and a given set of 10K random tests programs generated by the random test program generator, the experimental framework executes each random test program with a single design bug injected at a time and records if the bug is detected or not (test program fails). The graph of Figure 21 shows the number of test programs that fail for each injected bug. The vertical axis shows all the 1K design bugs injected into PTLsim simulator, while the horizontal axis shows the number of failing random test programs for each bug.

The large numbers of redundant test programs are evident in Figure 21. In particular: on average 52 test programs (0.52% of all 10K applied test programs) detect the same design bug (fail due to the bug existence), the maximum number of test programs that fail due to a single bug is 515 (5.15% of all 10K test programs) and the minimum is 2 (0.02% of all 10K test programs). Only 27% (273) of the 1000 injected bugs are detected by more than 30 of the 10K random tests (0.3% of the tests).

Clearly, the motivating observation of this work is valid. If this set of experiments is extrapolated for an overnight run of massive numbers of random test programs, the debug team will have to deal with a very large number of failing tests. Each and every failing test will probably need to be separately debugged a process that may take several days.

Second set of experiments. Aims to demonstrate the benefits of the deconfiguration mechanism for RIT triaging. Towards this aim, we have selected a set of 10 hard-to-detect logic bugs from the initial set of injected bugs (all 10 bugs are detected by a small number of test programs; smaller than the average case) distributed among the deconfigurable modules of PTLsim simulator.

We repeated the experiments only for the subset of the initial 1K random test programs that are affected from them (derived from the first set of experiments); these are 341 test programs. A critical difference in this set of experiments is that all 10 design bugs are together injected from the beginning of the bug injection campaign, as an attempt to model more accurately the silicon debug environment where all bugs can co-exist in the prototype chip. In this set of experiments, of course, the deconfiguration infrastructure shown in Figure 20 is enabled.

Table 16: Details for the 10 hard-to-detect design bugs.

Bug ID	Microprocessor Component	Failing Test Programs
1	Conditional Predictor	45
2	Return Address Stack	10
3	Issue $Queue_1$	32
4	Issue $Queue_2$	21
5	Floating Point Unit	50
6	Data cache	17
7	Load Queue	47
8	Store Queue	29
9	Reorder Buffer	48
10	Reorder Buffer	42
	Total	**341**

Table 16 presents details about the selected design bugs. First column is the id of each bug, while the second column gives the microprocessor component in which the bug resides. Issue Queue1 and Issue Queue2 refer to different components in the microprocessor design (Issue Queue1 for the integer cluster, and Issue Queue2 for the floating point cluster). The third column shows the number of test programs affected by each design bug when injected individually (from the first set of experiments). For example, the design bugs injected in Issue Queue2 cause 21 of the initial 10K test programs to fail. **Error! Not a valid bookmark self-reference.** describes the 10 bugs.

Table 17: Design bugs description.

Microprocessor Component	Bug Description
Conditional Predictor	Update fetch address on branch misprediction fails
Return Address Stack	Incorrect push to stack
Issue Queue$_1$	Dependent uop issued, while producer is waiting in ready to write-back state
Issue Queue$_2$	Entry not get flushed on a branch misprediction
Floating Point Unit	Incorrect rounding operation
Data cache	Valid array logic; invalid data read
Load Queue	Load to store aliasing
Store Queue	Store data before address gets valid
Reorder Buffer	Commit entry more than once
Reorder Buffer	Invalid control bit activation

Figure 22 shows the results from the execution of a subset of random test programs for the set of 10 hard-to-detect design bugs (all 10 bugs injected together – just like in a real prototype chip). In particular, it shows the different "buckets" of failing random test programs that are formed when the proposed methodology is applied (horizontal axis). The vertical axis shows the number of failing test programs of each bucket. In this set of experiments, the deconfiguration sequence is statically determined assuming pre-silicon bug susceptibility data is provided to the deconfiguration controller. The deconfiguration controller deconfigures the microprocessor modules for each pipeline stage starting from instruction fetch. Thus, the sequence of deconfigurations is the following: {Conditional Predictor, RAS, Issue Queue$_1$, Issue Queue$_2$, FPU, Data Cache, Load Queue, Store Queue, ROB}. When all the deconfigurable components from one stage are deconfigured it continues to the next stage. This process is repeated until the test program is executed correctly or all deconfigurable microprocessor components have been deconfigured.

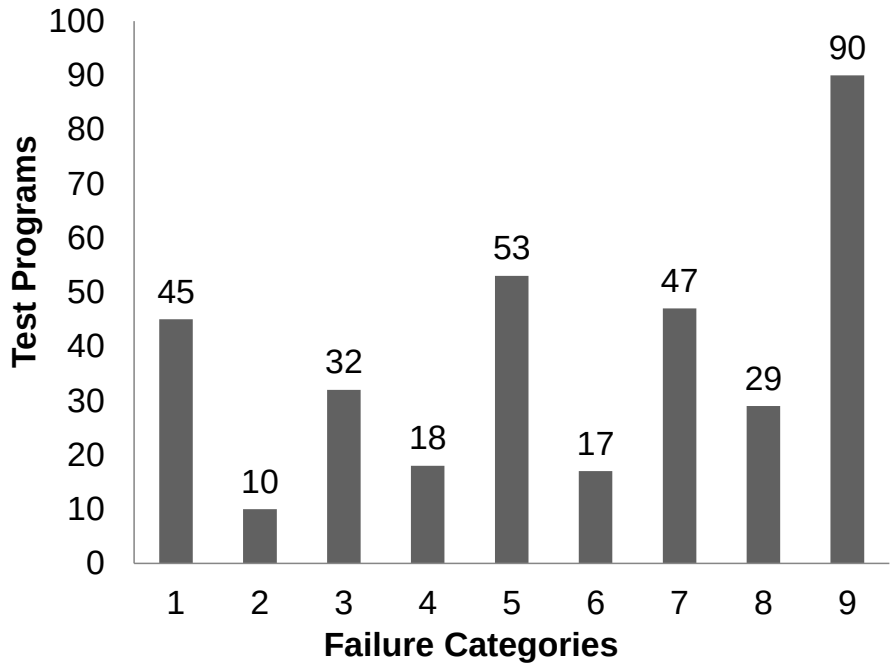


Figure 22: Failure categories for the 341 failing test programs.

The application of the proposed methodology, with the deconfiguration mechanisms enabled, results in a triaging of the 341 random test programs in 9 different failure categories shown in Figure 22. Some observations from this second set of experiments:

- Failure categories 1, 2, 3, 4, 6, 7, and 8 group the test programs that are affected exclusively from the design bugs in the following microprocessor components: Conditional Predictor, RAS, Issue $Queue_1$, Issue $Queue_2$, Data Cache, Load and Store Queues, respectively. As a result, when the deconfiguration controller turned the corresponding microprocessor component off, the bug is "masked" and the test program execution is correct. For example, the design bug in data cache unit, which manifested through the propagation of an incorrect data value, was masked when the data cache block was deconfigured from the design. Furthermore, the design bug in the load queue manifested as an invalid forwarding of loaded data to a dependent instruction. As a result, deconfiguring the load queue entries that hold that buggy information result in a correct execution of the test program.
- Failure category 5 groups 53 random test programs, while the expected number of test programs affected from a design bug in the FPU unit is 50. The reason for that is that these particular test programs (3 from Issue Queue2) were able to detect more than one design bugs (design bugs injected both in the Issue Queue and the FPU). As a result, only when both buggy microprocessor components were deconfigured the re-execution of the test program results in a correct execution.
- Failure category 9 includes the test programs that fail due to bugs 9 and 10 injected in the Reorder Buffer's logic. The deconfiguration mechanisms were unable to distinguish these design bugs into different categories, since both of them were inside the deconfiguration granularity of the ROB structure. Specifically, these bugs reside in neighbouring entries of the re-order buffer and manifest themselves as invalid dependency re-dispatching when a mispeculation happens.

Therefore, the same sequence of deconfiguration results in masking both bugs. This is still very effective because the debug team will certainly focus on the ROB component and it is very likely that it will identify the root cause of both bugs.

In a traditional silicon debug flow, without the proposed triage mechanism, the number of failing test in this set of experiments would be 341. This would be exactly the number of debug sessions that the debug team will need to examine starting the next morning. On the contrary, if the proposed deconfiguration-based silicon debug methodology is adopted, the number of failing tests remains the same (341) but the number of debug sessions would be only 9 (less than 3% of the traditional flow).

Clearly, the proposed flow has a profound impact on the effectiveness of silicon debug and greatly accelerates root cause analysis by removing the “noise” of redundant random tests that fail due to the same underlying bug. Figure 23 visualizes this reduction in the number of debug sessions when our methodology is applied.

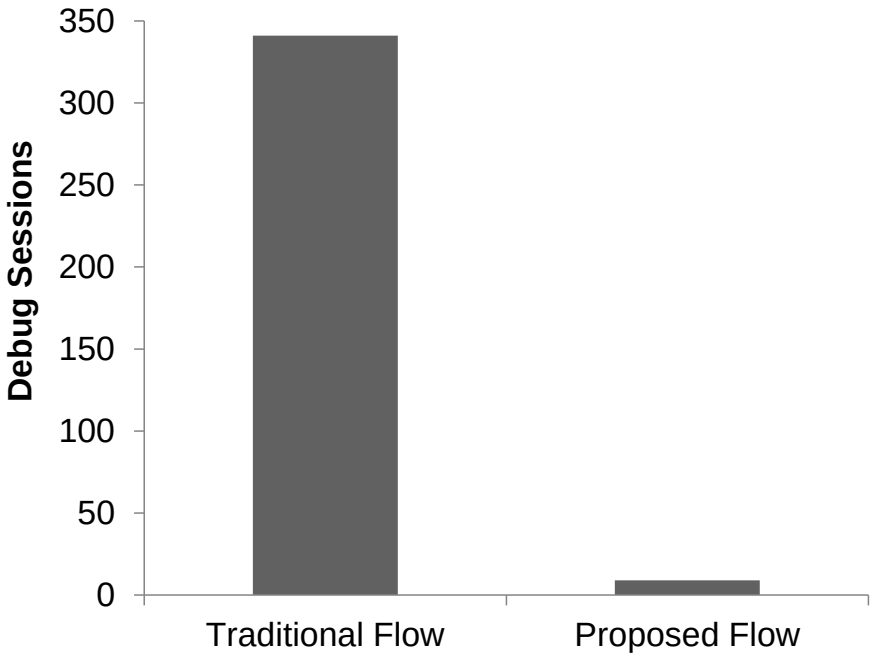


Figure 23: Number of debug sessions (number of failing test programs that must be debugged) in the traditional and the proposed flow.

2.4 Related Work

There is no work in the literature that reports employment of: (i) deconfigurable microprocessor architectures along with (ii) self-checking random test programs for the optimization of silicon debug.

Self-checking methods: Previous studies [105] [87] have proposed the generation of reversible test programs, where the program’s final state is known a priori, as a way to avoid the simulation step of golden signature production. However, generating reversible operations is not always an easy task and in some cases is partially or totally infeasible, like in the case of floating point operations. Another recent approach [48] targets to minimize the error detection latency of electrical bugs by duplicating instructions.

Software diversity: Previous approaches have adopted the concept of software diversity, as a zero-overhead alternative of design diversity, to build fault-tolerant

systems. The key idea is to modify the executed code when a hard fault is present, without spoiling the original code functionality [71]. Independent generation of programs has been also proposed as a fault tolerant approach [26]. Construction of programs with duplicated instruction and diverse data operands has been proposed as a way to detect temporal and permanent faults in the field [76]. Software implemented fault tolerance aims to provide soft error tolerance by instruction duplication [18]. Our method, for first time, utilizes the concept of ISA diversity for efficient silicon debug.

Triage: In [99] a static grouping of random test programs at generation time through the application of correlation, statistical and pattern recognition analysis methods is proposed. Differently, our methodology dynamically triages the test programs in runtime, based on the bug susceptibility of each component. The proposed framework provides enhanced log information to the debug engineers, facilitating the post-processing analysis of the failing self-checking test programs. On the contrary, the proposed methodology systematically addresses the issue of triaging through the introduction of hardware mechanisms capable to deconfigure a microprocessor design.

Debug [2] [25] [32] [50] [60] [61] [79]: Many proposals introduce various design-for-debug hooks into a design to monitor test execution and extract logging information to facilitate failure analysis. On the contrary, the proposed method acts proactively, in the silicon debug process, reducing the amount of test programs that need to be debugged, by detecting dominant failure modes among the failing random test programs. Furthermore, the massiveness of the silicon debug phase, both in test program execution throughput and in bug detection capabilities, encourages the adoption of high-level debug solutions. The proposed method addresses this challenge, in contrast to the existing research proposals that operate in a very fine granularity. It provides a unified solution for localizing the malfunctioning component throughout the microprocessor design. Obviously, the proposed methodology contributes to the acceleration of the root cause analysis through an improved triaging stage, and complements other silicon debug methods used in the industry.

Online bug detection: Previous approaches propose the use of dedicated hardware to detect and recover from bugs in the field [93] [29] [11] [102] [106]. Semiconductor industry needs bugs detected as soon as possible before massive production of the microprocessor chip. The proposed silicon debug methodology aims to satisfy this requirement.

Fault tolerance [6] [23] [22] [71] [84] [90] [97] [101]: Design deconfiguration is a well-known concept for tolerating hard errors in the field. The proposed methodology employs for first time the concept of deconfiguration in the silicon debug setup and in particular the random test program triaging step; as it is shown throughout this work, this is not a straightforward task.

2.5 Findings summary

Effective silicon debug for modern microprocessor architectures must minimize the simulation bottleneck and reduce the redundant debug sessions of random test flows to save time, resources, and budget while not limiting bug detection efficiency. We have proposed a novel, self-checking, hardware supported framework to accelerate and improve the quality of silicon debug by exploiting ISA diversity and the property of microprocessor components to be deconfigured without compromising the function completeness. Our analysis for ARM, MIPS,

PowerPC, and x86 instruction sets shows that despite their differences, modern ISAs can perform an operation with many equivalent ways. We take advantage of this ISAs property to generate random tests that detect bugs by comparing results of equivalent instructions. Moreover, several hardware components of high performance microprocessors can be “turned off” or deconfigured while the functional completeness of the design remains unaffected. We combine this property of microprocessor architectures with carefully developed self-checking random test programs to deliver a silicon debug methodology with an optimized triage stage. Redundant failing random test programs during an overnight random test programs execution campaign are grouped in classes each containing test programs that most likely fail due to the same underlying bug. This is decided based on the set of hardware components that need to be deconfigured so that each of the random tests programs is correctly executed. Experimental results, in an x86-64 microprocessor model prove the high bug detection efficiency, and also the large savings in debug time due to avoiding the simulation step of random test programs and by the effective triaging of failing tests using the proposed silicon debug methodology.

3. MANUFACTURING TESTING

The physical limits of semiconductor-based microelectronics have become a major concern in manufacturing technology. The diminishing gains in processor's performance due to the increasing gap between processor and memory speed (memory wall), the absence of enough parallelism in single instruction streams (ILP wall) and the exponential escalation in power consumption (power wall) motivate computer architects and designers to look at different directions for next processor generations.

Current industry trend is orientated towards the development of chip multiprocessors (CMP) and chip multithreaded (CMT) processors which although may operate at lower frequencies are able to deliver higher performance exploiting thread-level (intra-core) or processor-level (inter-core) execution parallelism. However, test technology has to explore the transfer from the uniprocessor era to the multiprocessor era (CMP and CMT architectures) of all test techniques, that have been recently devised to deal with the emerging reliability problems of modern microprocessors. The main objective of this transfer of techniques to multithreaded multiprocessors should be the exploitation of the execution parallelism of the new processor architectures to avoid excessive scaling of the overall test time and therefore improve time-to-market but without degrading the effectiveness of the technique in terms of fault coverage.

Software-Based Self-Testing (SBST) [80] [27] [30] [81] [38] [42] [46] [68] [109] is a testing method that has gained increasing acceptance with major microprocessor vendors and today forms an integral part of the manufacturing test flow of single-threaded processors [85]. The key idea of SBST is to exploit the instruction set architecture and on-chip programmable resources to execute self-test programs. The use of SBST methodologies contributes to the reduction of yield loss (avoids over-testing), while its non-intrusive nature does not require any processor hardware modification. In addition, at-speed testing ability enables screening of timing defects that do not manifest themselves at lower frequencies.

The effective application of SBST to multithreaded multicore architectures poses significant challenges: (i) porting of existing self-test programs from the single-threaded, single-core case to efficiently test all the individual cores; (ii) providing sufficient fault coverage for the thread-specific control logic, that constitutes a significant portion of the control logic in the new multithreaded architectures and schedules the execution of threads on it; and (iii) exploitation of thread-level and core-level parallelism to reduce test/validation execution time.

We present a complete multithreaded software-based self-testing (MT-SBST) methodology that targets both the optimization of test execution time and the improvement of the fault coverage of the thread-specific control logic. First, we assess the impact of test routine scheduling in the fault coverage of hard-to-test control structures (i.e. difficult to be controlled through test program stimuli): the thread switch logic inside each processor core and the thread-specific control logic of the shared components out of the processor cores. Subsequently, we propose a multithread scheduling algorithm that achieves a very efficient trade-off between test execution time and fault coverage of the thread-specific control logic, and is only based on easy-to-obtain run-time statistics of the single-threaded execution of the self-test program.

3.1 Scope of the MT-SBST

The proposed methodology is applied at the manufacturing testing phase of the microprocessor dependability cycle for the last quality control before chips are shipped to customers for integration in a system. Our proposed MT-SBST methodology performs the following:

- Test program development for all the functional units of a CMT multiprocessor architecture.
- Test program profiling for the single-threaded single-core execution for manufacturing testing (execution from on-chip cache memory).
- Assessment of the impact of the multithreaded execution of test program on the fault coverage of the thread-specific control logic.
- Test program scheduling to take maximum advantage of thread-level parallelism and speedup execution of its test routines for the core-internal functional units, and core-level parallelism to speedup the execution of its test routines for the core-external shared functional units. At the same time, our scheduling improves the fault coverage for those structures that are sensitive to the thread scheduling.

We provide full demonstration of the proposed methodology in the most complex publicly available CMT processor architecture, OpenSPARC T1 [77]. Our experimental results show that the proposed multithread scheduling algorithm speeds up the execution time of test program at both core-level (up to 3.6X) and processor-level (up to 6.0X) compared with the single-threaded execution. Furthermore, compared with a straightforward multithreaded execution of the test program the proposed multithreaded schedule reduces test execution time at core-level and processor-level more than 33% and 20%, respectively. On top of these significant improvements in test time, and despite its shorter execution time, the proposed MT-SBST schedule improves the fault coverage of the thread switch logic of each core by about 10% compared with the straightforward multithreaded version. Overall, our methodology guarantees high stuck-at fault coverage levels: more than 91% for the functional units (all integer functional units of the eight cores and the shared floating point unit) and more than 88% for the logic of the entire processor (including the functional units, the thread switch logic and the interconnection networking, which count about 1.5M logic gates).

3.2 SBST of single-threaded processors

The basic concept of software-based self-testing (SBST) [37] for a single-threaded single-core processor is depicted in Figure 24. Test program is executed by the processor at normal mode of operation. Test instruction sequences usually load test patterns from memory (or generate them internally) and apply the appropriate operations to excite faults in hardware components; for example, in Figure 24 the test code loads two operands (test vectors of the adder circuit) and adds them to excite a hardware fault in the adder module. Finally, in order to propagate the fault effect to observable locations the test code moves the test responses from the register file to data memory. The first hard task in SBST is to generate test instruction sequences that can adequately test the processor modules achieving high fault coverage. Several recent works propose efficient test program generation methodologies targeting different modules of single-threaded microprocessor cores, such as integer functional units [27] [30] [81]

floating-point units [109], pipeline and control logic [42] [38] and speculative mechanisms [46].

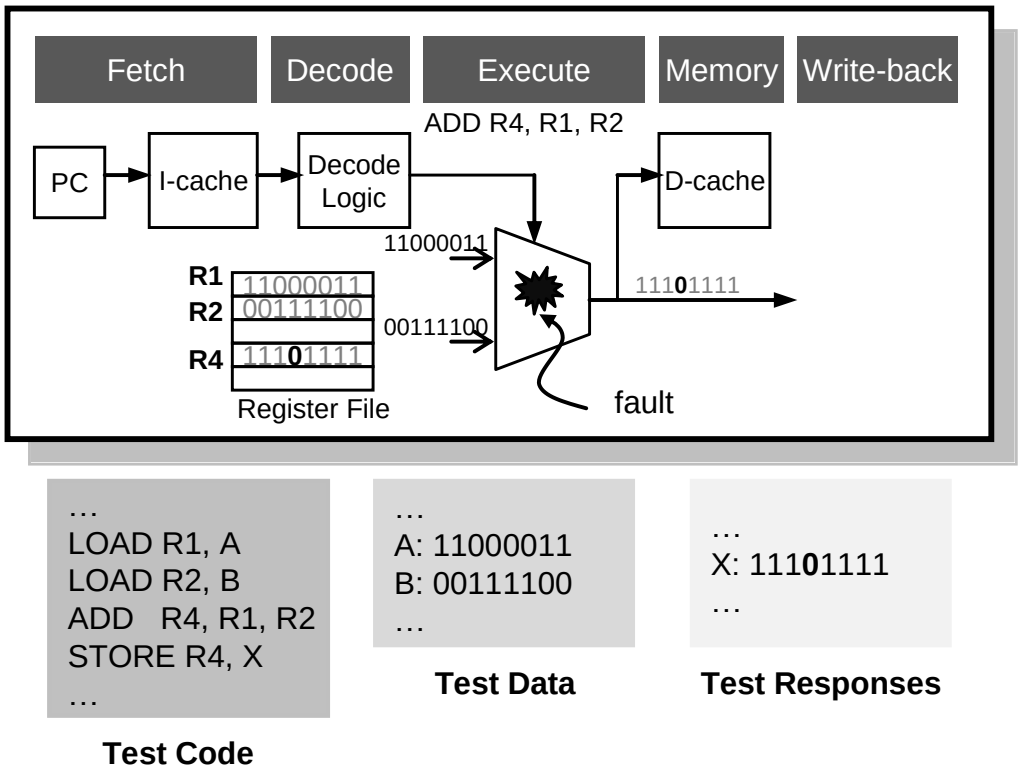


Figure 24: Software-based self-testing concept for single-threaded microprocessors.

The main advantages of SBST are:

- *Non-intrusiveness*. SBST operates in normal functional mode and does not require extra hardware.
- *At-speed testing*. SBST application and response collection are performed at the processor's full speed which enables screening of delay defects that do not manifest themselves at lower frequencies.
- *No over-testing*. SBST avoids test overkill and thus detection of defects that will be never manifested during the normal processor operation; this leads to significant yield increase.

SBST is a reusable, all-around solution for checking the microprocessor's integrity throughout its life cycle; self-test software can be executed during manufacturing testing, and periodic online testing. The role of SBST in a manufacturing test flow is complementary since it does not aim to replace the other traditional testing approaches. On the contrary, SBST improves the overall test quality combining the benefits of the other approaches: self-test program can be developed targeting low-level structural fault models and applied in native functional mode.

Figure 25 presents a typical SBST flow for manufacturing testing which comprises three steps: (1) test code and data are downloaded into on-chip instruction and data caches, respectively (for simplicity caches are not shown separately), using a low-speed, low-cost tester. Test data downloading is performed via a cache load interface at low-speed; (2) test program is executed by the processor at full speed and test responses are stored back to on-chip data cache; and (3) tester responses are uploaded into the tester memory via the low-speed cache interface for external evaluation. Self-test programs must be developed so that no cache

misses occur during test execution, a scheme called cache-resident testing [14], [80]. This allows reducing the total test cost by (i) reducing test execution time avoiding external (main) memory access cycles, and (ii) eliminating the need for expensive high-speed functional testers that would handle the memory transactions.

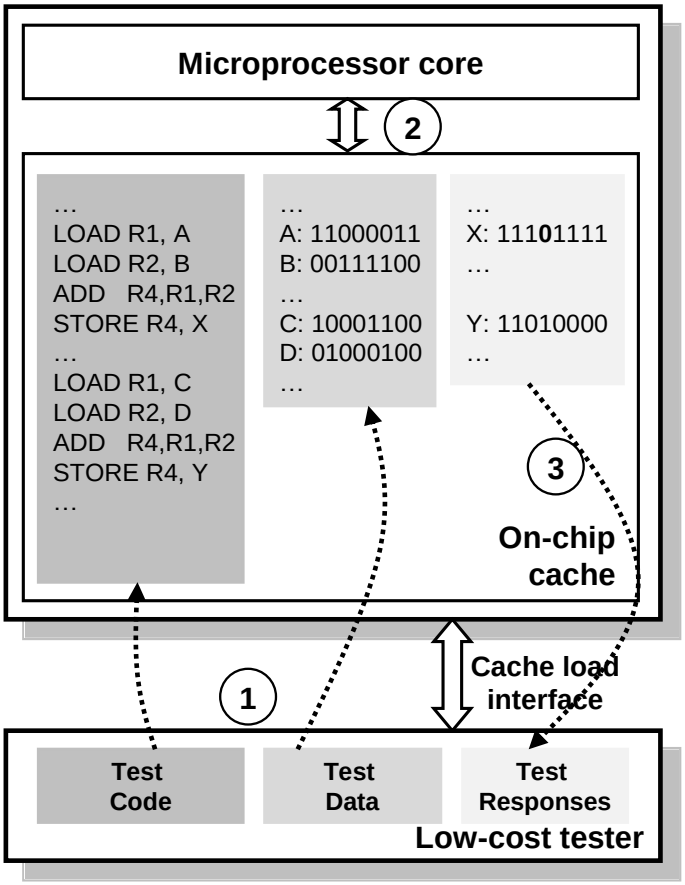


Figure 25: Manufacturing testing SBST setup.

3.3 MT-SBST preliminaries and experimental setup

For the application of SBST in a multithreaded multicore architecture, we assume the following experimental setup:

- The test program consists of a set of test routines that target all the private functional units of each processor core (i.e. functional units in the execution pipeline of each core such as ALU, multiplier, divider and shifter) and the shared functional units (i.e. a floating-point unit that all cores of T1 share).
- A single copy of the test program (test code and data) is stored in memory (either on-chip cache or main memory depending on the setup) instead of a separate copy for every core; this reduces the memory storage requirements. All processor cores have to execute the same test program to detect faults in their private units while the self-test program for the shared units must be executed once (in one core or split in more cores).
- Each processor core generates a set of separate test responses; this assumption enables the diagnosis of faulty core (the alternative is to compact all responses from all processors loosing the diagnosis

capability). This is important for manufacturing testing since it allows the binning of partially “good” chips (those containing some faulty cores) [104].

In order to reduce the execution time in an MT-SBST approach, we need to take advantage of both the available thread-level and core-level parallelism, visualized in Figure 26. Let assume four test routines for the functional units FU_1, FU_2, FU_3 and FU_4 of the processor core (these routines must be executed be each core) and one test routine for a shared-functional unit (this routine must be executed once). Exploitation of core-level parallelism enables the parallel execution of the test routines FU_1, FU_2, FU_3 and FU_4 by all n processor cores and speeds up the execution of the shared-FU test routine. If execution parallelism is not exploited, the overall test application time will scale with the number of processor cores (8 in T1 multiprocessor). Instead of having a single core to execute the shared-FU routine (top of Figure 26), the routine is split into *n* subroutines which can be executed in parallel (middle of Figure 26). We can schedule in a different way the test routines in the *n* cores to achieve the optimum utilization of the common memory subsystem and the interconnection network [9]. Next, we exploit thread-level parallelism to speedup the execution of the test routines in each core; assuming that each core supports four hardware threads in an interleaved multithreading fashion, all 4 threads are used to execute the test routines as shown in Figure 26 (bottom). The overlap of the idle intervals of one thread (i.e. due to a long latency operation or a cache miss) by another active thread is the key point for the efficient parallelization of test routines.

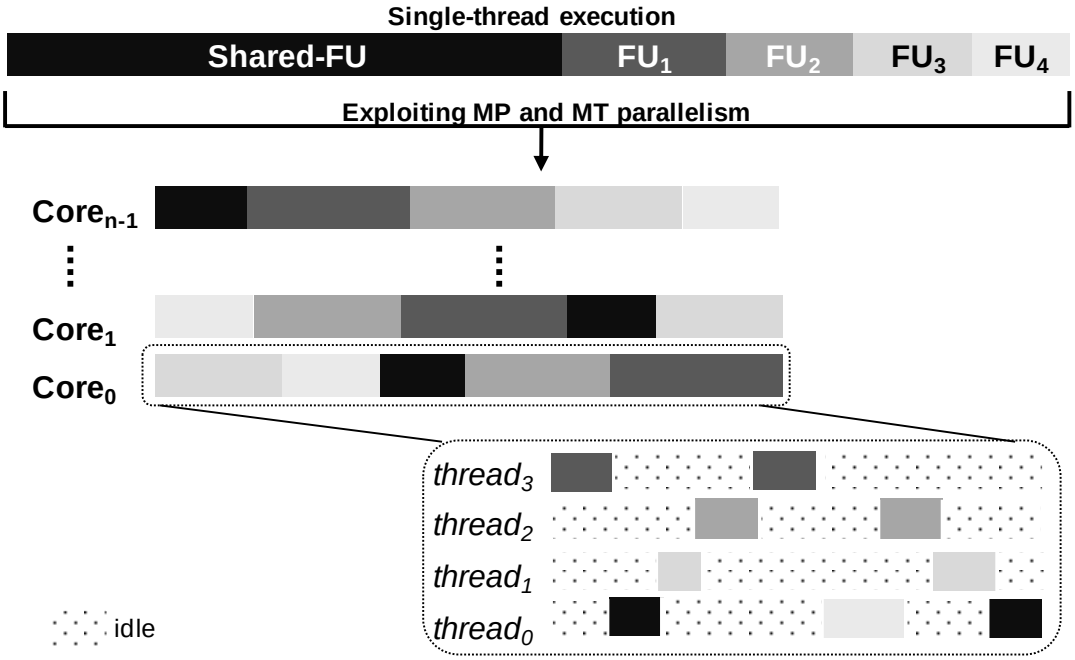


Figure 26: Exploiting MP and MT parallelism in the execution of the test program.

3.4 Proposed MT-SBST Methodology

When normal applications are developed for a multithreaded architecture the main focus is the maximization of the application throughput and processor’s resource utilization. The tuning of the application workload depends on its specific characteristics. However, self-test programs do not belong to a specific class of commercial workloads with common characteristics, and thus require separate performance analysis. We aim to tune self-test programs to the characteristics of the multithreading technology to achieve the maximum speedup, that – as our

experiments reveal – a naïve, straightforward multithreading scheduling cannot reach. The necessity for effective self-test scheduling algorithms as the one proposed in this thesis is therefore revealed.

The main objectives of the proposed methodology are: (a) to assess the test program execution characteristics for its efficient tuning towards a multithreaded architecture; (b) to analyse how the multithreaded execution of the test program affects the fault coverage of the thread-specific control logic (which is not explicitly targeted by the test routines for the functional units); and (c) to propose an efficient scheduling algorithm which reduces test program execution time without degrading its effectiveness in terms of fault coverage for the related logic. Overall, the main goal of our methodology is to achieve the best trade-off between self-test time reduction and self-test effectiveness for the thread-specific control logic. The steps of the methodology are summarized in Figure 27 and individually analysed in the following subsections.

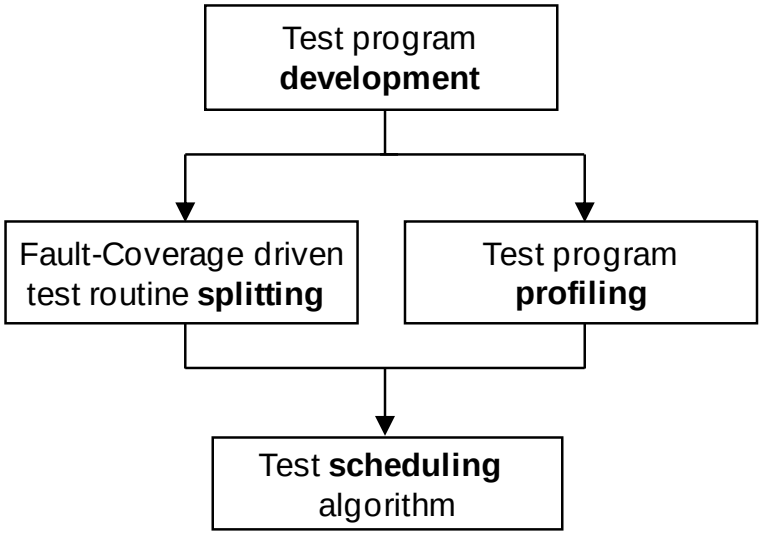


Figure 27: Proposed MT-SBST Methodology.

3.5 Test Program Development

Our demonstration vehicle is the open-source CMT processor model, OpenSPARC T1, which integrates eight 64-bit SPARC V9 processor cores, each one supporting four hardware threads [77]. Figure 28 shows the organization of the OpenSPARC T1 processor. Each CPU core implements a six-stage, single-issue execution pipeline and has a 16KB L1 instruction cache and a 8KB L1 data cache. An on-chip unified 3MB L2 cache divided in four banks is shared among all CPU cores. A crossbar switch handles communication between the CPU cores and the shared memory while at the same time provides access to a shared floating-point unit. OpenSPARC T1 uses fine-grain multithreading technology: it switches among the available threads at every cycle giving priority to the least recently executed thread.

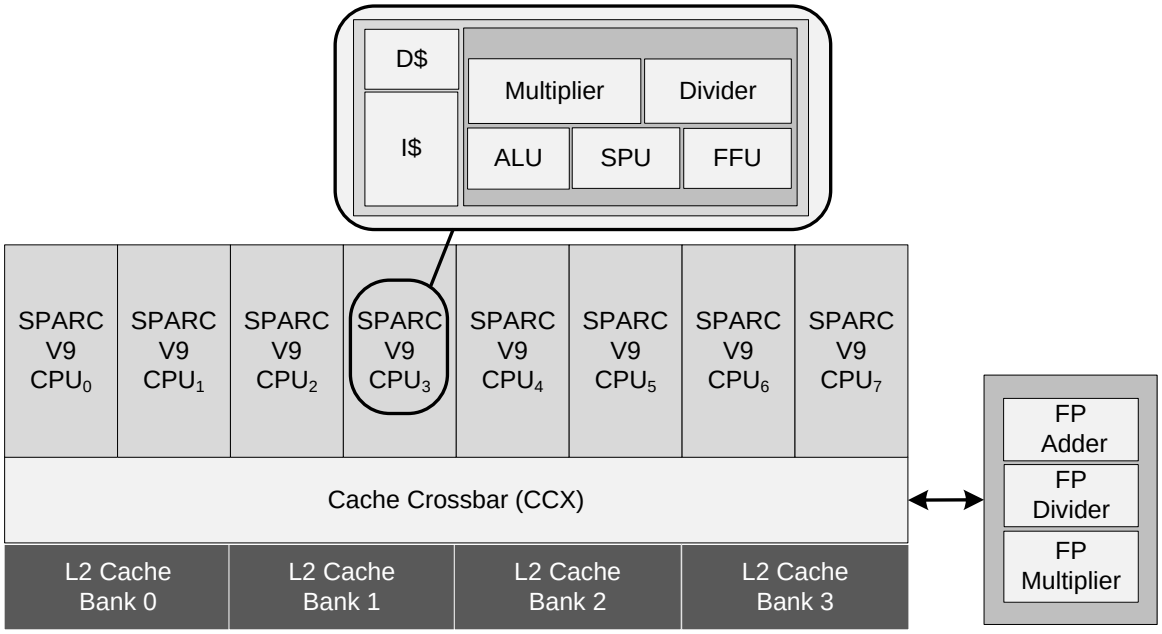


Figure 28: OpenSPARC T1 architecture.

The first step of the proposed methodology is the development of test routines that target all the complex functional units of the SPARC V9 core: ALU, shifter, integer multiplier, integer divider, stream processing unit (SPU – used for cryptography operations), and floating-point frontend unit (FFU). The test routines for these six functional units must be executed by all processor cores. We also develop separate test routine for the components of the off-core floating-point unit (FPU) of OpenSPARC T1 which must be executed only once (FP adder, FP multiplier, FP divider).

For a few functional units, like the shifter and the multiplier we adopted proven effective optimized test sets from previous SBST approaches [81] [38] for other single-core models and tune them to the functional units of SPARC V9 core. For the other modules, we either developed customized test routines (like in the cases of the ALU and the divider) or used the regression tests of the modules (like in the cases of FFU and SPU) included into OpenSPARC T1 verification suite and enhanced them with more test patterns. It is important to note that this first step of self-test program development does not affect the operation of the subsequent steps. This means that any self-test program for the individual integer and floating-point units can be used. One of the important part of this work is that our experiment work stress the limits of MT-SBST on OpenSPARC T1 using as efficient as possible self-test programs for the individual units.

Table 18 summarizes the characteristics of the functional units of the SPARC V9 core and the corresponding test routines. Second column presents the gate count of the functional units and third column demonstrates the fault coverage achieved by the corresponding test routines in a single-thread execution (results are based on the stuck-at fault model and have been calculated using Synopsys' TetraMAX tool).

Table 18: Functional units and corresponding test routines of each SPARC v9 core.

Functional units	Gate count (K gates)	Fault coverage (stuck-at %)	Single-thread execution time (K cycles) Manufacturing testing
Shifter	5.9	97.5	14.4
ALU	6.2	92.7	32.5
Divider	11.4	97.3	54.5
Multiplier	54.2	96.4	8.6
FFU	16.6	72.1	9.9
SPU	18.5	86.9	33.1
Total	**112.8**	**91.2**	**153.0**

The rightmost column show the test routine execution time in a single thread for manufacturing testing (execution from on-chip shared L2 cache). The execution time of the test routines depends on: the number of test patterns, the latency of the corresponding instructions and the development style (which affects the instruction-level parallelism of the routines – loops, etc.). Our test program achieves more than 91% fault coverage in total for all the functional units which is the highest structural fault coverage that has been ever reported by a software-based testing approach on a real open-source industrial processor such as OpenSPARC T1. Note that, the low fault coverage of the FFU is due to the partial activation of the component from the single-thread test programs, while combined with the scheduling algorithm the fault coverage is increased.

In Table 19 we present the effectiveness of the FPU routine in terms of stuck-at fault coverage only to the execution pipelines (adder, multiplier, divider) included in the shared floating-point unit. We deal with the control part of the floating-point unit later. The developed FPU routine achieves more than 92% stuck-at fault coverage on average for this complex functional unit. The total execution time of FPU routine is 2.6M clock cycles when executed from on-chip shared L2 cache.

Table 19: Modules of the shared FP unit and the corresponding test routines.

Modules	Gate count (K gates)	Fault coverage (stuck-at %)	Single-thread execution time (K cycles) Manufacturing testing
FP Add.	33.7	91.7	1300.1
FP Multiplier	60.1	92.9	520.4
FP Divider	13.6	91.0	780.2
Total	**107.4**	**92.3**	**2600.7**

The fault coverage of the functional units is not affected when the corresponding test routines are executed in a multithreaded fashion. However, this is not the case for the control logic, either the thread-specific control logic of the core or the shared FPU control logic.

3.6 Test Program Profiling

The second step of the methodology is the high-level profiling of the single-thread version of the test program that allows us to quickly assess its scaling characteristics to a multithreaded environment. All test routines are executed in a single hardware thread of a SPARC V9 core having exclusive access to the core while the other three threads are parked (i.e. exclusive single-thread performance).

Figure 29 shows the exclusive single-thread performance of all test routines for the manufacturing testing SBST setup (see explanation for the different routines of each type at the end of this section). Each bar represents the fractions of time the state machine of the hardware thread, executing the corresponding test routine, is in one of the five possible states: ready, run, wait, speculative ready and speculative run. The SPARC V9 core switches among the available threads at every cycle (i.e. fine-grain multithreading technology). A thread can be scheduled (is available) when it is in one of the following states: ready (i.e. the hardware thread is available for selection by the scheduler), speculative ready (i.e. data dependencies are expected to be resolved and the thread will soon be available), run (i.e. the hardware thread has been selected), and speculative run (i.e. the hardware thread will be selected by the scheduler) . On the other hand, a thread enters the wait state due to one of the following reasons: I-cache fill, store buffer full, long latency operation, and resource conflict (i.e. concurrent requests to a shared resource). Therefore, when executing the test routines in a single-threaded core, the core enters a wait state when the thread is unavailable. To collect runtime statistics for the thread state we used the functionality of the thread monitor unit of SPARC V9 core.

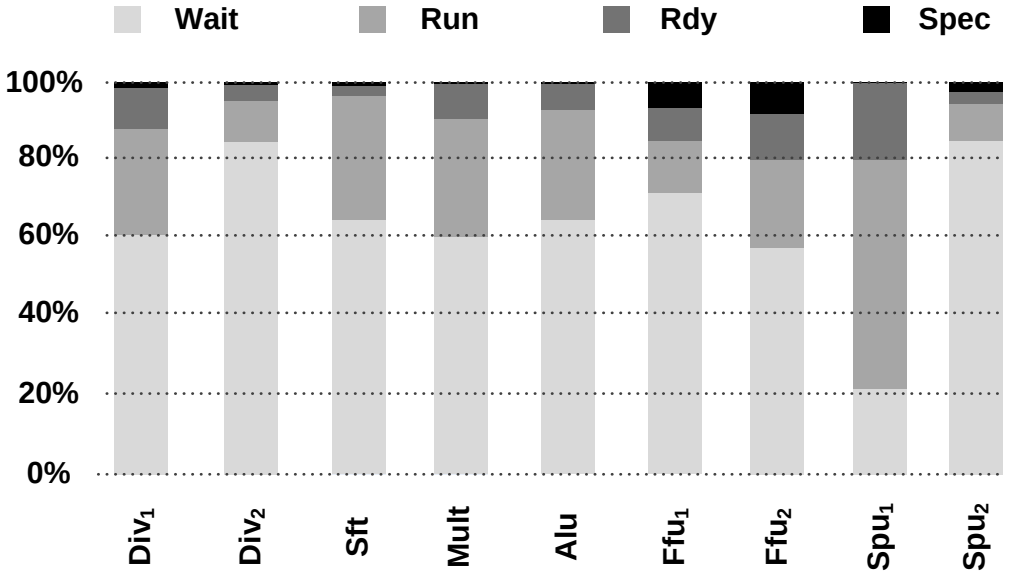


Figure 29: Test program profiling for manufacturing testing.

Test program profiling shows that the total core utilization is very low since the core is waiting for long time intervals because the thread is unavailable. In the case of manufacturing testing (Figure 29) the thread is in wait state for the 62% of the total execution time of the test program. Hence, the test program profiling stage designates the ability for performance gains when routines scheduled in multithreaded environment.

We further analysed test routine profiles to identify different execution phases, such as CPU-bound or memory-bound intervals, within a test routine execution

and then we split it into more subroutines based on these phases. This splitting procedure enabled us to schedule more efficiently the test routines into the hardware threads achieving better exploitation of TLP. In our case study, routines Div, FFU and SPU, present different runtime statistics and are split into two subroutines each, Div_1 (24.2 K) and Div_2 (30.3 K), FFU_1 (9.4 K) and FFU_2 (0.5 K) and SPU_1 (23.8 K) and SPU_2 (9.3 K), respectively (parentheses show the execution time from L2 cache in clock cycles).

3.7 Fault coverage-driven test routine splitting

At this section, we study how the multithreaded execution of test routines affects the fault coverage of the on-core (thread switch logic) and off-core (shared FPU) control logic.

On-core control logic (thread-switch logic). Thread-switch logic fault coverage increases with the activity of the four thread state machines (and therefore, the number of state evaluations and thread selections that the thread-switch logic performs). Thus, if we want to keep the fault coverage of the thread-switch logic high, we should avoid decreasing the number of state transitions of the thread state machines by forcing the four threads to enter more times in the wait state. However, this target contradicts with the test execution time reduction goal since increasing the number of resource conflicts (i.e. concurrent requests to a shared resource) will adversely affect the exploitation of CMT technology.

We start considering two routines from our basic core test program that can cause resource conflicts due to their long latency operations: multiplier and divider routines. We performed a set of fast, high-level experiments to quantify the speedup achieved if we split these test routines into two or four time-balanced subroutines and schedule two or four hardware threads to execute them in parallel. In Table 20, we compare the time of the single-threaded execution versus the multithreaded execution for these two routines for execution from L2 cache.

Table 20: Single-threaded execution vs. multithreaded execution (ET: Execution Time).

Testing setup	Routines	1-thread	2-threads		4-threads	
		ET (A) K cycles	ET (B) K cycles	Speedup (A/B)	ET (C) K cycles	Speedup (A/C)
Manufacturing testing	Multiplier	8.6	5.7	1.5	5.4	1.6
	Divider	54.5	37.1	1.5	35.9	1.5

The experimental results show that the two-threaded execution achieves significant speedup, 1.5X times, over the single-threaded execution. However, the speedup saturates at two threads since using more than two threads reduces slightly the execution time. Therefore, to improve the fault coverage of the thread-switch logic during the multithreaded execution we split the long-latency routines into subroutines that causes resource conflicts when executed in multithreaded mode. However, to achieve the best trade-off between execution time reduction and fault coverage of the thread-switch logic the number of subroutines must not exceed the number of threads at which the speedup saturates. The output of this step is a number of sets each one containing the appropriate number of subroutines that must be executed in parallel to cause resource conflicts. In our case study two sets are created: {Div_1, Div_2} and {$Mult_1$, $Mult_2$}.

Off-core control logic (shared FPU). We exploit core-level parallelism to execute the test routines for the off-core shared FPU. In order to determine an efficient multicore, multithreaded execution of FPU test routine we study how the execution time and the fault coverage scale with the number of cores and threads. Thus, we split FPU test routine into 4, 8, 16 and 32 subroutines and schedule them to different number of processor cores: 1, 4 or 8 cores each running 1 or 4 threads. Table 21 presents total execution time and combined stuck-at fault coverage of the two FPU control sub-modules: FP input that multiplexes the FPU requests from multiple cores and FPU output that arbitrates the results of FP pipelines for the single FPU-crossbar connection. Table 21 presents results for both the execution from L2 cache. Our experiments demonstrate that the fault coverage is affected by the execution of FPU test routine by multiple cores and multiple threads. This happens because the FPU control modules carry thread and core id specific information. The results suggest that the most efficient FPU routine schedule in terms of speedup and fault coverage in both setups is 8 cores each running 4 threads: a total of 32 threads executing in parallel 32 FPU time-balanced subroutines. Thus, in our proposed test scheduling the FPU test subroutines are executed in parallel by all processor cores – separately from basic core test routines – occupying all 32 threads of the CMT architecture.

Table 21: Multicore, Multithreaded execution of FPU test routine (ET: Execution Time, FC: Fault Coverage of the FPU control logic).

Schedule	1 thread		4 threads	
	ET (K cycles)	FC (%)	ET (K cycles)	FC (%)
Manufacturing Testing				
1 core	2600.7	61.9	1400.1	62.7
4 cores	920.1	89.9	490.3	91.0
8 cores	519.2	90.9	437.4	91.6

3.8 Test Scheduling Algorithm

We propose an algorithm that schedules a set of test routines $\{R_1, R_2, \ldots, R_N\}$ into *k* hardware threads targeting the best trade-off between test execution time and fault coverage. The proposed algorithm is presented in Table 22.

The first part of the algorithm partitions test routines into two groups: G_L which contains routines having waiting time fraction (WT) less than the average waiting time fraction (WT_{avg}) of all test routines and G_H which contains routines having WT more than WT_{avg}. Then, the two groups are sorted in descending order according to the execution time (ET) of their routines.

The second part of the algorithm picks up test routines from the two groups and assigns them into threads in an iterative manner. The longest test routines (with the higher ET) are scheduled first in order to produce a time-balanced scheduling. When a routine that belongs to a resource conflict group (RCG) (an RCG contains routines that perform concurrent requests to a shared resource) is selected then all the other elements of the group are scheduled in parallel. If there are routines that cannot be scheduled in parallel due to resource limitations they are not selected in the current loop iteration. For instance, in our case study, routines SPU_1 and SPU_2 cannot be executed in parallel since the co-processor implementing the SPU operations supports one outstanding SPU operation per core.

Table 22: Test Scheduling Algorithm.

Inputs: k: number of threads
Basic core test routines: $S = \{R_1, R_2, \ldots, R_N\}$
Single-threaded test program profiling results:
ET_i: execution time of routine R_i
WT_i: waiting time fraction of routine R_i
WT_{avg}: average waiting time fraction of all test routines
Routines groups causing resource conflicts: $RCG_1, \ldots, RCG_M$

Restrictions: Routines cannot be executed concurrently due to limited resources (i.e. SPU_1, SPU_2)

Output: Scheduled test routines in k threads: $\{SR_{th1}, SR_{th2}, \ldots SR_{thk}\}$

// Partition routines into two groups: G_L (low WT fraction) and G_H (high WT fraction)
for i = 1, 2, …, N **do**
  if $WT_i < WT_{avg}$ **insert** R_i to G_L ;
    else insert R_i to G_H ;
end for
Sort G_L and G_H in descending order according to ET_i
$ET_{th1}, ET_{th2}, \ldots ET_{thk} = 0$; // Accumulated ET of routines assigned to threads 1…k
$SR_{th1}, SR_{th2}, \ldots SR_{thk} = \emptyset$; // Set of routines scheduled to threads 1…k
$CXR = \emptyset$; // Set of currently executed routines by all k threads

while (G_L, G_H **not** empty) **do**
  select thread j with shortest ET_{thj};
  remove the last routine of SR_{thj} from CXR ;
  if (G_H empty) **OR**
    ((G_L **not** empty) **AND** (#routines in CXR with low WT < # of routines in CXR with high WT)) **then**
      select the longest routine R_i from G_L that does not have restriction with any routine of CXR ;
      remove R_i from G_L ;
  end if
  if (G_L empty) **OR**
    ((G_H **not** empty) **AND** (# of routines in CXR with low WT ≥ # of routines in CXR with high WT)) **then**
      select the longest routine R_i from G_H that does not have restriction with any routine of CXR ;
      remove R_i from G_H ;
  end if
  insert R_i to SR_{thj} ;
  insert R_i to CXR ;
  if R_i **belongs** to an RCG_m **then**
    remove R_i from RCG_m ;
    while (RCG_m **not** empty) **do**
      select next longest routine R_i from RCG_m ;
      select thread j with shortest ET_{thj} ;
      remove the last routine of SR_{thj} from CXR ;
      remove routine R_i from its group (G_L or G_H) ;
      insert R_i to SR_{thj} ;
      insert R_i to CXR ;
    end while
  end if
end while

The algorithm satisfies two scheduling criteria: (a) routines causing resource conflicts (belong to a resource conflict group, RCG) are executed in parallel; and (b) at any time the set of currently executed routines (CXR) contains equal number of low-WT and high-WT test routines. The first criterion targets to the improvement of the fault coverage of the thread-specific control logic and the second criterion targets to overlap the "long" waiting intervals of the half routines with the "running" intervals of the other half routines. The algorithm output is *k* sets SR_{th1}, SR_{th2}, … SR_{thk} that contain the routines scheduled to each thread.

3.9 MT-SBST experimental results

We applied the proposed scheduling algorithm to the test routines of functional units of the OpenSPARC T1 for manufacturing testing setup. For the sake of comparison, we also set up a naïve (straightforward) multithreading approach that assigns routines with the same characteristics to the same thread, i.e. routines using the multiplier (SPU and Mult), divider routines (Div), short latency operations (ALU and Sft) and floating-point operations (FFU and FPU). Both naïve and proposed multithreading approaches are based upon the same requirement: to avoid resource conflicts that degrade test program performance. Therefore, naïve approach constitutes a fair alternative of the proposed approach.

We first analyse core-level thread scheduling without considering testing of the shared FPU. The generated test routine schedules for manufacturing testing setup and the naïve scheduling approach are shown in Table 23. Each column includes the test routines scheduled in each thread of the core. Notice that the proposed schedules for the two SBST setups are different which is due to the different test program profiling.

Table 23: Schedules of core test routines.

	Routines per Thread Assignment			
Naïve scheduling	*Thread 0*	*Thread 1*	*Thread 2*	*Thread 3*
	SPU_1 SPU_2 $Mult_1$ $Mult_2$	Div_1 Div_2	ALU Sft	FFU_1 FFU_2
Proposed scheduling	**Manufacturing Testing**			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	ALU	Div_1 $Mult_2$ FFU_1	Div_2 SPU_2	SPU_1 $Mult_1$ FFU_2 Sft

In Table 24 we compare the proposed multithreaded scheduling with the single-threaded and naïve scheduling approaches in terms of execution time and stuck-at fault coverage of the thread-switch logic (recall that the coverage for the functional units is more than 91% – see Table 18 – since the coverage does not depend on the multithreaded execution). The speedup of the multithreaded approach is calculated against the test execution time of the single-threaded execution. The speedup achieved by the proposed multithreaded scheduling is up to 3.3X, very close to the ideal theoretical 4X speedup, which means that it exploits the TLP very efficiently, using only easy-to-obtain run-time statistics from the single-threaded execution and avoiding time consuming simulations.

Compared with the naive scheduling (that achieves a speedup only up to 2.2X), our methodology reduces the test time by 33%.

Table 24: Comparison of core level scheduling approaches (FC: Fault Coverage of thread switch logic).

	Single-threaded	Naïve scheduling	Proposed scheduling
Execution time (K cycles)	153.0	69.2	46.1
Speedup	–	2.2	3.3
FC (%)	32.6	67.6	75.5

Furthermore, the proposed scheduling does not degrade the fault coverage of thread-specific control logic of the core but on the contrary (due to the elaborate routines scheduling) it improves it up to about 10% compared with the naïve scheduling, thus achieving an excellent trade-off between speedup and fault detection capability.

From this point onward, we include the testing of the control part of the shared FPU (recall that the coverage for the FPU adder, multiplier and divider is more than 92% – see Table 19) in our scheduling. In naïve scheduling the FPU routine is split into 8 subroutines ($FPU_i/8$) which are executed by thread 3 of each core shown in bold in Table 25. In our approach the FPU test routine is split into 32 time-balanced subroutines ($FPU_i/32$) which are executed by all four threads of each core before the basic core test routines: all 32 threads of the architecture are occupied to execute in parallel the FPU subroutines. Note that Table 25 presents only the schedules of processor core 0 for the naïve (straightforward) approach and our proposed approach for manufacturing testing. The schedules for all processor cores can be produced directly from Table 23 scheduling the FPU subroutines before the core test routines.

Table 25: Schedules of core test routines plus shared FPU routine at processor level.

Routines per Thread Assignment				
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
Naïve scheduling	SPU_1 SPU_2 $Mult_1$ $Mult_2$	Div_1 Div_2	ALU Sft	FFU_1 FFU_2 **$FPU_{1/8}$**
Proposed Scheduling	**Manufacturing Testing**			
	Thread 0	*Thread 1*	*Thread 2*	*Thread 3*
	$FPU_{1/32}$ ALU	**$FPU_{2/32}$** Div_1 $Mult_2$ FFU_1	**$FPU_{3/32}$** Div_2 SPU_2	**$FPU_{4/32}$** SPU_1 $Mult_1$ FFU_2 Sft

Table 26 summarizes test execution time of single-threaded, naïve scheduling and proposed scheduling approaches and the speedup achieved by the multithreaded approaches over the single-threaded one. Compared with the

naïve scheduling, the proposed scheduling reduces the test execution time of the entire processor by up to 18%.

Table 26: Comparison of scheduling approaches including FPU routine.

	Single threaded	Naïve scheduling	Proposed Scheduling
Execution time (K cycles)	2753.7	588.4	483.5
Speedup	–	4.6	5.6

Finally, Table 27 presents the fault coverage for the targeted logic (about 1.5M gates of logic) of the OpenSPARC T1, which includes all the integer functional units and the on-core control logic (thread switch logic and integer pipeline control logic) of all eight CPU cores, the shared FPU (including the execution units and the thread-specific control logic) and also the interconnection network (this is not explicitly targeted by test routines). The total fault coverage for all functional units (both integer and floating-point) is 91.3%, while the total fault coverage for the entire processor is 88.6%.

Table 27: Fault coverage (IFUs: Integer functional units, FPU: Floating-point unit, CCL: Core control logic, INN: Interconnection network, FUs: Functional units of processor).

Components		Gate count (K gates)	Fault coverage (stuck-at %)		
			Single threaded	Naïve scheduling	Proposed scheduling
Core (x8)	IFUs	8×112.8	91.2	91.2	91.2
	CCL	8×28.4	62.2	71.8	82.8
Off-core	FPU	115.8	86.7	88.8	92.3
	INN	259.5	14.9	79.9	82.7
Total (FUs)		**1018.2**	**90.7**	**90.9**	**91.3**
Total (Processor)		**1504.9**	**73.4**	**86.1**	**88.6**

3.10 Related work

There is no work in the literature that studies (a) the impact of SBST test optimization strategy on a multicore, multithread architecture; and (b) measures the fault coverage of thread-specific control logic.

Multithreading: Bayraktaroglu et al. [14] proposed the conversion of existing legacy tests, either hand-written or randomly-generated to test the multithreaded cores of the CMT architecture of UltraSPARC T1. They described how a software-based cache-resident test methodology can be utilized during the manufacturing test flow of a commercial multicore chip, UltraSPARC T1, and applied by a low-cost external tester. In [14], the CPU cores of the CMT architecture execute the test program sequentially while the other cores are disabled; this scheme eliminates the need for replicating the test program for each processor core but it does not exploit either the core-level parallelism or the thread-level parallelism of the architecture, thus, it does not satisfy the main objective of a multithreaded SBST methodology. Apostolakis et al. [9] considered the application of SBST to bus-based CMP architectures consisting of simple single-threaded cores. They proposed a scheduling methodology for the test routines to exploit core-level execution parallelism and minimize the time overheads coming from the memory subsystem in order to reduce the total test

execution time. However, [9] focuses only on the execution parallelism among different cores in order to improve test program performance for the processor.

Multiprocessor: A first approach on studying the application of SBST in CMT architecture for manufacturing testing, was proposed in [10] where thread-level parallelism is exploited to reduce self-test execution time.

3.11 Findings summary

We present the application of SBST in multithreaded, multicore architectures as a natural extension to single-core, single thread SBST. The proposed MT-SBST methodology leverages the existing thread-level parallelism (TLP) for test optimization. We analyse the impact of multithreaded test execution on fault coverage and propose a methodology to speed up the test execution time by exploiting execution parallelism without degrading the fault coverage of the control logic (but on the contrary improving it). Comprehensive experiments on OpenSPARC T1 demonstrate that our methodology speeds up the test time of a 4-threaded core by 3.3. Compared with a straightforward multithreaded scheduling the proposed methodology achieves significant time reduction, 33% at the core-level and 18% at the processor-level. Overall, our methodology guarantees high fault coverage, more than 91% fault coverage for the functional units and more than 88% for the entire OpenSPARC T1 processor logic (more than 1.5M gates of logic).

4. IN-FIELD VERIFICATION

High-performance microprocessor architectures consolidate all available design techniques towards a single aim: delivery of the highest performance given the power constraints of an individual market segment. Towards this aim, a continuously increasing number of transistors are integrated in microprocessor chips along with sophisticated circuit design and architectural techniques to improve performance. Chip manufacturing technologies, however, have already entered an era where permanent, intermittent, and transient hardware faults have unacceptably high rates due to manufacturing defects, process variation, environment impact, and device wear-out and aging [74]. Under these circumstances, the microprocessor chip manufacturing cost is seriously affected by reduced yield levels and the inevitable overheads for fault tolerance.

Silicon technology process scaling has been shown to increase the rate of hard faults in microprocessor SRAM arrays [59] [18] [21]. One of the most important trend for reducing processor power consumption is low voltage operation which further increases the rate of hard faults in SRAM arrays [28] [1] [108]. Several hard fault tolerance techniques for SRAM caches have been proposed [5] [28] [66] [89] [100] as well as mechanisms to protect pipeline flip-flops and combinational logic [23] [84] [94] [71] [90] [95] [41] [75]. However, many non-cache structures such as those in control and data flow speculative hardware are also implemented using large SRAM arrays (e.g. branch target buffer size is 2.5K entries in AMD's Jaguar core [92], which equals to 9% of the non-cache core area, accounting for 44% of the non-cache SRAM area).

Although these speculation mechanisms do not affect program correctness, faults that lead to extra mispeculations can significantly degrade performance [46] [44] [49] [63] [7]. The exact behaviour depends on the location of the fault (in an array entry or in the control part of the component) and the component-access pattern by an application or benchmark. Transient faults, such as those caused by high-energy particle strikes, minimally affect performance because they incur few extra mispredictions and thus will not be investigated in this study. However, hard (permanent) faults can significantly reduce performance especially when faulty array entries are accessed frequently.

Performance loss from faults reduces performance without corresponding reductions in power, decreasing overall efficiency. Performance variability across identical cores is undesirable in many settings including data centre and HPC deployments. For instance, estimation of the Total-Cost-of-Ownership (TCO) of a data centre can be done using estimation frameworks as the one reported in [45]; these frameworks show that, under certain configurations, performance variability increases both system cost and power consumption and worsens the system's environmental impact. Moreover, large parallel workloads running on HPC environments often execute at the speed of the slowest node [34] meaning that performance variability among nodes can substantially reduce the overall throughput of the system. Performance variability is also undesirable in the mobile and desktop markets [51] since it reduces the ability to provide performance guarantees in real-time systems.

We measure the effect of permanent faults in the arrays and the control logic of performance components: branch predictors (BPs), branch target buffers (BTBs), the return address stack (RAS), and data and instruction prefetchers (DP and IP). Soft errors are exempt from this study, since their transient impact has insignificant effect on microprocessor's performance. Then, we propose a set of

low-cost hardware-based mechanisms for the detection, diagnosis and performance recovery in the BP and BTB structures.

4.1 Scope of proposed techniques

The reliability evaluation and proposed techniques are applied at in-field verification phase of the microprocessor dependability cycle. In particular, we perform the following tasks:

- Classify the behaviour of microprocessor speculative mechanisms in the presence of faults in the four categories (output error, performance errors, benign errors and crash).
- Demonstrate that faults such as those from low-voltage operation and process scaling can lead to substantial performance loss and variability in these structures. We assess the impact of hard faults in the conditional Branch Predictor (BP), the branch target buffer (BTB), and the return address stack (RAS), using expected rates of hard faults for future technologies and low-voltage operation.
- We propose a set of low-cost hardware-based mechanisms for the detection, diagnosis and performance recovery in the aforementioned structures. Our objectives are to recover the fault-free execution time by mitigating the impact of faults on IPC, and to minimize variation among cores with different fault locations in control flow predictors. Our techniques leverage the observation that the inherent self-verifying nature of all these components offers an opportunity for low-cost diagnosis of faults. Therefore, we overload the self-verification mechanism to trigger hard fault detection and diagnosis in each predictor. Once a hard fault is identified, we employ limited spatial redundancy to minimize the number of additional mispredictions and recover performance.

Our analysis delivers the following key insights. First, performance components with single permanent faults do not lead to functional errors and that most faults (44% to 96%) cause only performance fluctuation. Second, hard faults in a stride data prefetcher can affect microprocessor performance significantly (up to 26%) and increase inter-core performance variability (more than 4.5%). Likewise, in the branch prediction unit the performance loss (up to 13%) and variability (more than 16%) can be significant. Finally, the low-cost hardware-based solutions for the detection, diagnosis and performance recovery of multiple hard faults in all front-end speculation components (BPs, BTBs, RAS, BHR) achieves to mitigate almost the entire IPC loss due to faults.

4.2 Background analysis

4.2.1 Performance components

The quest for higher performance at lower power continues as we keep stacking more transistors on a die. All major compute elements (CPUs, GPUs, etc.) employ an array of structures whose main (and, in some cases, only) purpose is to provide higher performance (caches, BPs, prefetchers, load/store speculation, execution units, out-of-order schedulers, hardware-based multi-threading, etc.). These mechanisms are used to hide the ever-increasing memory latency (caches, prefetchers, load/store speculation) and boost parallelism exploited either at the instruction level (OoO schedulers, execution units, BPs) or at the

thread level (multi-threading). The effect of permanent errors on some of these structures has been studied in the past [49] [17]. Most of the reliability studies have focused on structures based on the area they occupy (caches), their immediate performance impact (execution units), and their ability to influence functional correctness (caches, schedulers).

We examine the impact of permanent faults on the front-end structures of high-performance microprocessors. These include branch outcome predictors, BTB, and RAS. Modern microprocessors employ all these techniques in an effort to provide a continuous stream of instructions to their execution units. The accuracy of these mechanisms is a key metric in achieving high instruction-level parallelism (ILP). Permanent faults in their arrays or control logic can cause the mis-prediction rate to fluctuate, leading to performance changes. High mis-prediction rates can cause additional memory traffic (especially for conditional and indirect branch prediction), which in turn can amplify the side effects on the core IPC and power. Independent of the actual nature of the workload (compute- or memory-dominated), the steady supply of instructions to the back end of the core will remain critical in maintaining high performance at a low power cost.

We also steer focus towards structures that affect performance only. A prefetcher is one such case. If not accurate enough, it can degrade performance (and increase power) by polluting the cache and by wasting shared resources (miss information/status handling registers (MSHRs), tag/DC and command/data bus bandwidth, victim buffers, etc.). If the prefetcher is accurate enough, it reduces the average memory latency and lowers the need for larger data caches. However, permanent faults can cause variations in the prefetched address stream, which in turn can lead to large IPC fluctuations if the prefetcher is accurate and the data working set of the application does not fit in the data cache. Permanent errors can change prefetcher coverage (by dropping pending prefetch requests and training opportunities), prefetch request timeliness (by issuing requests earlier or later than their error free equivalent), and prefetch accuracy (by perturbing prefetch address-generation logic).

4.2.2 SRAM arrays failure probabilities

Technology modelling in roadmaps predicts extremely small single-bit failure probabilities for combinational logic even beyond the 12nm node [74]. However, several orders of magnitude higher numbers of hard faults in SRAM arrays are expected in the following two contexts:

- Chips manufactured in current and future technologies (e.g., 22nm) that operate at reduced voltage levels for power reduction purposes [28] [1] [108].
- Forthcoming chips manufactured in more defective technologies (16nm, 12nm) [101].

In both contexts, the failure probability (Pfail) of a single SRAM cell is expected to fall between 10^{-6} and 10^{-4} [28] [74] [108], a substantial increase over the SRAM cell Pfail in 32nm, as shown in Table 28.

Table 28: SRAM cell Pfail for four technology nodes [74].

Node	SRAM cell Pfail	Node	SRAM cell Pfail
32nm	7.30E-09	16nm	5.50E-05
22nm	1.50E-06	12nm	2.60E-04

To put this in perspective, Figure 30 shows the expected number of faults in 100Kbit and 300Kbit SRAM arrays (within range of BP and BTB arrays used in commercial processors; see Introduction) for the technology nodes of Table 28, expressed as a cumulative probability. As we can see in Figure 30, the average number of faults in a 100Kbit array in 32nm is only 0.001, while the same array in 16nm is expected to have 5.5 faults on average.

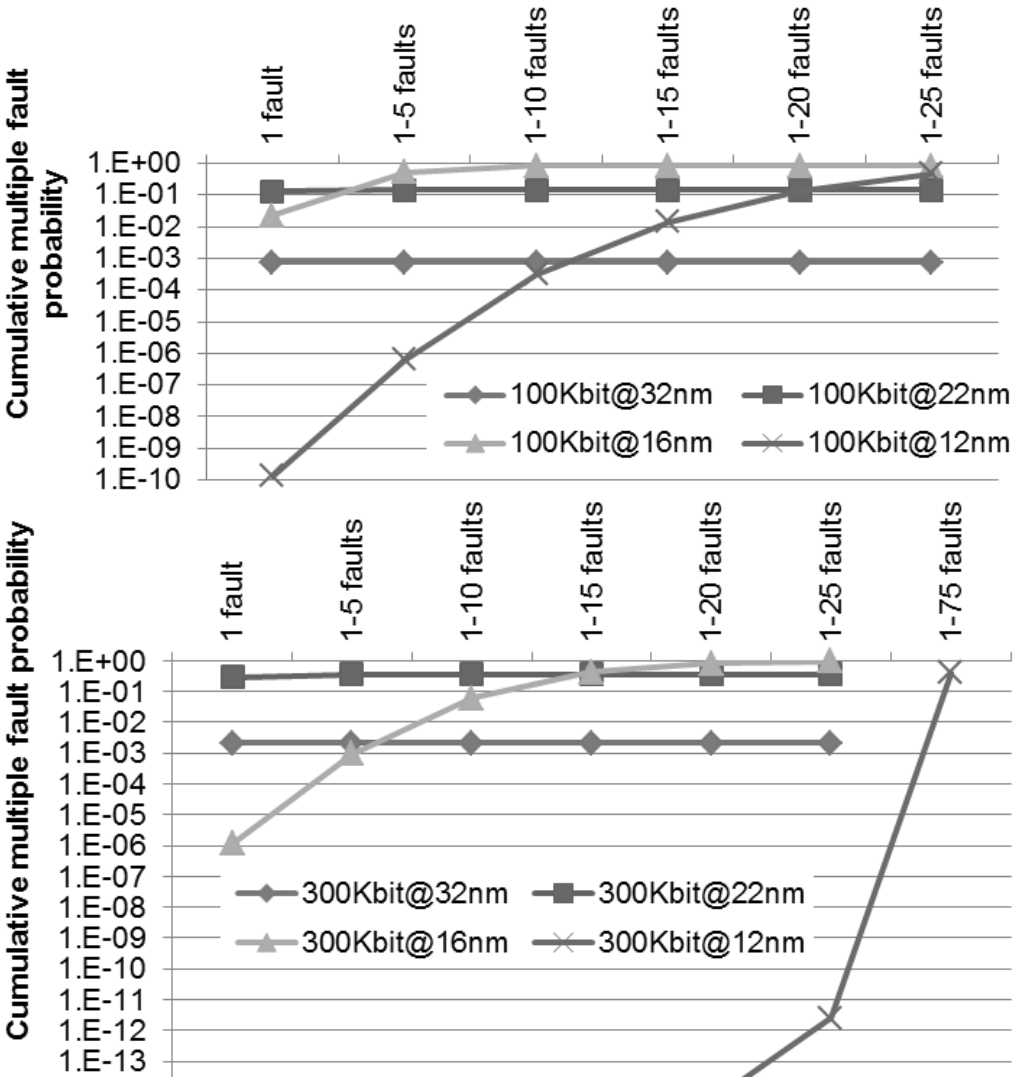


Figure 30: Cumulative probability of 1…k hard faults for 100Kbit (top) and 300Kbit (bottom) SRAM arrays.

4.2.3 Fault classes

We classify the outcomes of each fault injection simulation based on the impact of the fault on the simulated system. Follows the analysis of fault effects classes. These represent typical classes (and corresponding terminology) used in the reliability literature.

- *Output error*: The fault causes data corruption at benchmark output, register values, or memory state (although this category is expected to be empty, we include it in our study and check the processor state at simulation end to verify that functional correctness is preserved).
- *Performance error* (slowdown or speedup): The fault changes only the execution time of the benchmark.
- *Benign error*: The fault does not cause an output or performance error.

- *Crash*: The fault produces an exception or system crash.

4.3 Simulator and Microprocessor Model

Our statistical fault injection campaign runs on top of the PTLsim x86 architectural simulator [110], which is used widely for performance measurements. We have enhanced the x86-64 microprocessor model of PTLsim so it resembles as much as possible a modern design that incorporates all the major performance components of our study. Table 29 summarizes all parameters of the enhanced x86-64 microprocessor model we used in our experiments.

Table 29: Enhanced x8-64 model configuration.

Parameter	Setting
Fetch//Issue/Commit	4/4/4 instructions per cycle
Return address stack	16 entries
Branch target buffer (Cond./Uncond. direct branches)	4-way set associative, 1K sets
Branch target buffer (Indirect branches)	4-way set associative, 512 sets
Combined predictor	Bimodal, Two-Level predictors 16KB (65,536 entries, 2 bits per entry, 16 bits history) Meta-predictor table: 65,536 entries
Reorder buffer	128 entries
Functional units	4 clusters (ALUs: 2 INT, 2 FPU)
L1 instruction cache	32KB (64B cache line, 128 sets, 4 ways, 2 cycles latency, 8 MSHRs)
L1 data cache	16KB (64B cache line, 64 sets, 4 ways, 2 cycles latency, 32 MSHRs, max. MSHR entries for prefetch requests: 20)
L2 cache	256KB inclusive (64B cache line, 16 ways, 12 cycles latency, 40 MSHRs)
L3 cache	4MB inclusive (64B cache line, 32 ways, 40 cycles latency, 40 MSHRs)
Main memory	Infinite size (200 cycles latency)
Prefetch input queue (PIQ)	8 entries
Prefetch table	64 entries, 4-way set assoc., PC indexed
Confidence size	3 bits
Confidence threshold	3
Stride size	5 bits
Prefetch distance	1 (single step)
Prefetch request queue (PQR)	8 entries

First, we enhanced the branch prediction unit (Figure 31) with a new BTB model for conditional/direct branches (cBTB) with a misprediction penalty of 3 cycles. The cBTB is separate from the one PTLsim uses for indirect branches, which we label iBTB. Conditional and direct branches form the majority of instructions modifying control flow, so it is important to add the cBTB so the pipeline can maintain a steady stream of instruction supply to its execution units. In addition, simulating permanent errors to cBTB is more important than the iBTB since the cBTB predicts the targets of those branches.

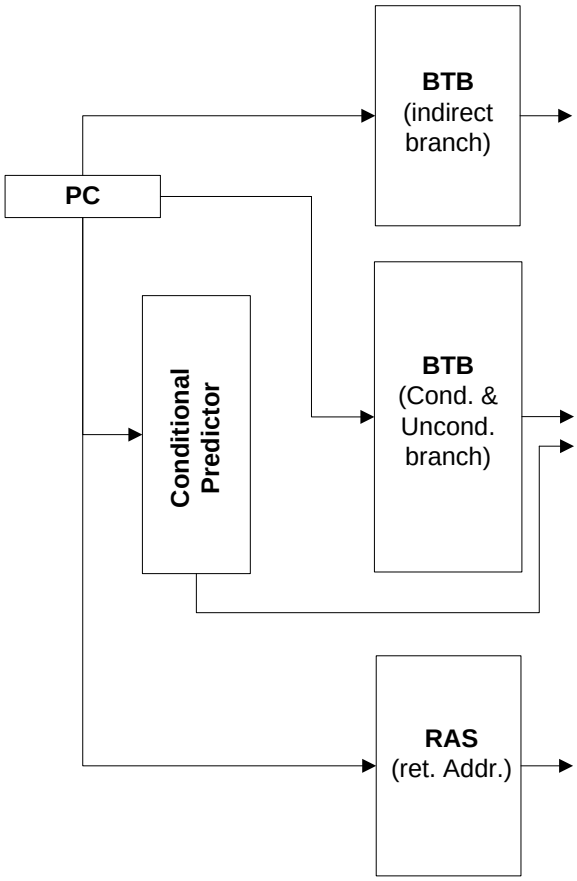


Figure 31: Block diagram of the branch prediction unit.

Second, we added a next-line instruction prefetcher (IP) to the instruction cache that shares the miss handling status registers (MHSR) with demand instruction cache miss requests from the processor (Figure 32). Instruction prefetching is an effective technique in hiding latency of the instruction cache misses that are non-overlapping in time because instruction cache is typically a blocking cache. The instruction prefetcher works as follows: On miss, a prefetch request is added on the prefetch queue with the next sequential cache line. Then, the request is issued to the L1 instruction cache; in case of miss an entry on the miss buffer is allocated and the request is propagated to the lower level of the cache hierarchy. Finally, request that already exists on the miss buffer are dropped.

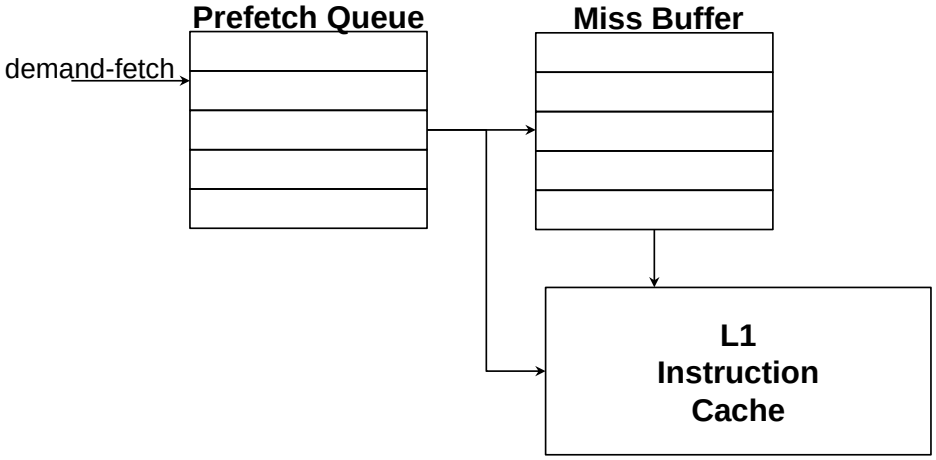


Figure 32: Block diagram of next-line instruction prefetcher.

Third, we added an L1 data cache stride data prefetcher (DP) that also shares the MSHRs with demand data cache miss requests (Figure 33). An address-training queue and a prefetch-request queue were also modelled to faithfully mimic the logic used to train the DP as well as the generation of prefetch requests before they are issued to the data cache (for hit/miss identification). The prefetch table consists of the following fields: (i) Tag: RIP of the x86 memory operation, (ii) Load Address: previous train address, (iii) Stride: most recently recorded stride, (iv) Confidence: *m* bit counter to indicate the occurrences of a particular stride, (v) Valid: indicates the existence of valid data; and (vi) LRU: replacement policy bits. The data prefetcher operates as follows: The load/store addresses produced by the address generation unit are buffered into the input queue to train the prefetcher. Repetitive memory operations with a linear stride generate prefetch requests with the prefetch distance shown on Table 29. The prefetch requests are then buffered in the request queue. Finally, null and off-range strides are dropped.

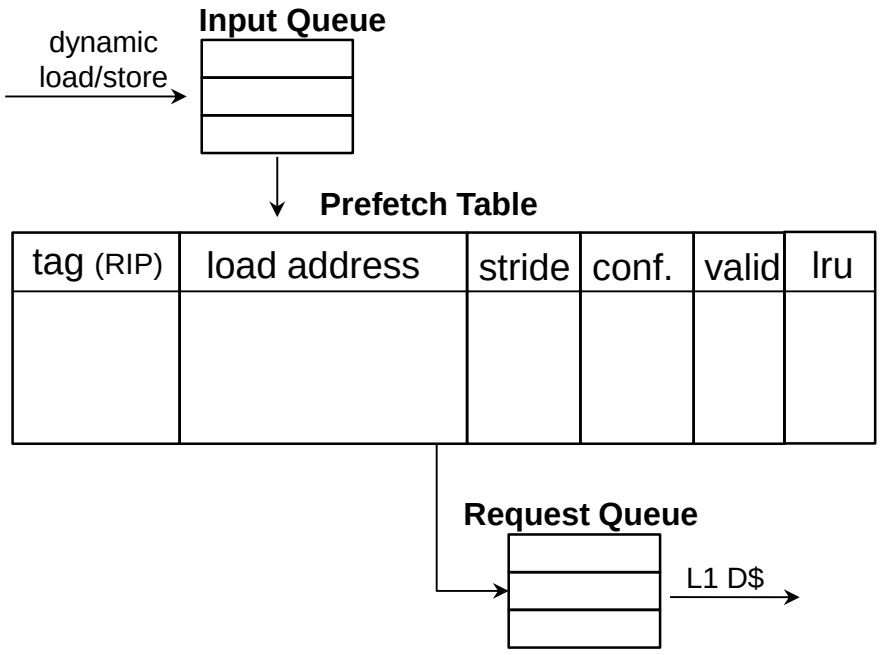


Figure 33: Block diagram of the L1 data cache stride prefetcher.

4.4 Statistical Fault Injection Framework

We have developed a statistical fault injection framework, on top of PTLsim architectural simulator, to evaluate the impact of permanent faults on the performance of modern microprocessors (Figure 34). It consists of three main elements: a fault mask database, golden and faulty models of the microprocessor, and a post-processing analysis tool. The fault mask database is populated with the set of fault masks injected in both the arrays and the control logic of the microprocessor components. Each fault mask has the following fields:

- Module_ID: The targeted microarchitectural array.
- Entry_ID: The line inside a structure where a permanent fault is injected (for the set-associative components, a pair of entry/way is generated to define an array entry exclusively).
- Position_ID: The bit location within an entry to inject a fault.
- Fault_type: Stuck-at-0, stuck-at-1.

The fault mask database for the arrays is populated based on the statistical sampling technique of [67] (The method has been originally proposed for soft error injection; it computes the number of injection experiments in an array of given size under confidence level[1] and error margin[2] requirements. There is no mandatory temporal parameter in the sampling of [67] and it can be adopted for both transient and permanent faults by taking or not time into consideration, respectively). More details regarding the selected confidence level and the error margin will be presented in the following sub-sections. Further, the Entry_ID, Position_ID, and Fault_type attributes are randomly selected for each fault mask based on a normal distribution. The control faults are modeled by modifying the semantics of the simulator's source code that models the control logic of each microarchitectural component. We injected a total of 155 permanent faults in the control logic of the DP and the BPU. The selection of the 155 control-logic faults was done using only one criterion: maximum correspondence of the injected fault in the architectural simulator model and an actual single stuck-at hardware fault at the RT level assuming a generic logic design implementation of each sub-component.

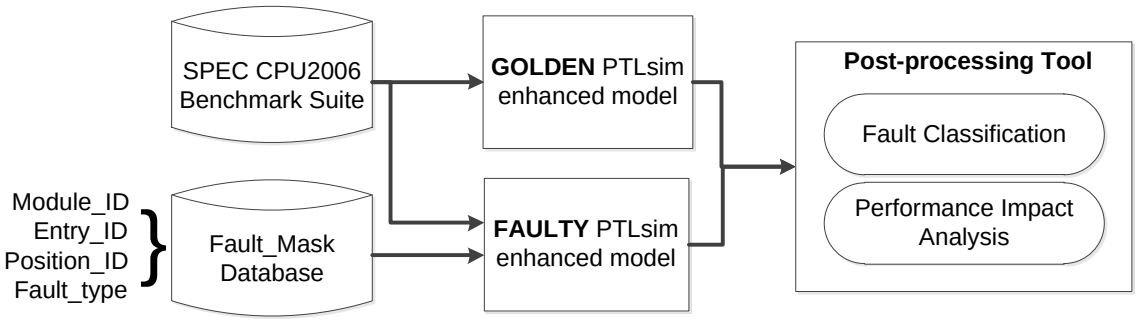


Figure 34: Statistical fault injection framework.

Each SPEC CPU2006 benchmark runs once on the golden (fault-free) model and once for each fault in the database on the faulty model (on each run, a single fault is injected). Figure 35 shows the timeline of a simulation run. At the beginning of the fault injection simulation, the framework reads the fault mask from the database and launches simulation. We warm up the microprocessor for an interval of 20 million committed x86 instructions (no checkpoint captured). At the end of the 20 million instructions, the fault is injected and executed for 80 million more committed instructions (i.e., each simulation lasts 100 million committed x86 instructions). At the end of each run, a checkpoint of the simulator state is extracted for further off-line analysis.

We present our experimental results in three parts:

- The first part classifies the behaviour of data and instruction prefetcher sub-components (arrays and control) in the presence of faults in the four categories (output error, performance error (slowdown or speedup), benign error, and crash) and then we measure the performance impact (IPC) due to them.

[1] Probability that the observed sample contains the measured attribute's real mean in the full population.
[2] Maximum expected difference between the population's mean value and a sample's mean value of the measured attribute.

- The second part classifies the behaviour of the branch prediction unit sub-components (arrays and control) in the presence of faults in the four categories (output error, performance error (slowdown or speedup), benign error, and crash) and quantify the performance impact (IPC) due to them.
- Third part presents a low-cost microarchitectural mechanism that detects and tolerates the performance impact of hardware faults in the branch prediction unit.

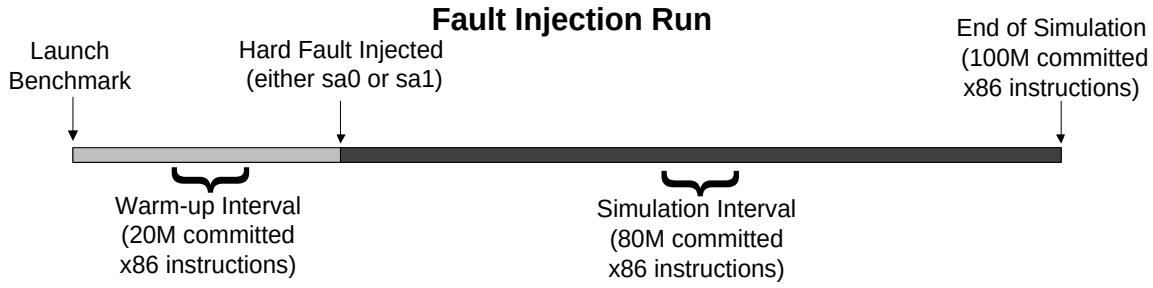


Figure 35: Fault injection simulation timeline.

4.5 Resiliency of data prefetcher

4.5.1 Classification of faults

In this sub-section, we present the fault classification of the prefetcher sub-components. For confidence (95%) and error margin (5%), 2,604 faults masks are sampled and injected into the arrays (Table 30).

Table 30: Distribution of the injected faults on the prefetcher.

Component	Field	#Injected faults
Data prefetcher array	Load address	571
	Tag	571
	Stride	216
	Confidence	149
	LRU	128
	Valid	58
	PRQ – Load Address	289
	PIQ – Load Address	289
Data prefetcher control	Prefetcher table index-generation logic	15
	Prefetcher table tag-search logic	4
	Prefetcher table replacement logic	4
	Stride calculation logic	5
	Confidence calculation logic	5
	Output logic to issue a prefetch request	2
	PRQ - CAM multi-hits	3
	PRQ - Head/Tail pointer	3
	PIQ - Head/Tail pointer	3
Instruction prefetcher	Fetch address	289
Total	**2,560 array faults + 44 control faults**	**2,604**

The following tables break down the fault classifications for each field of the prefetcher. We separately present results for the arrays and the control parts. No output error has been observed in our experiments and thus we have omitted columns for output errors (all are 0%); this is an expected result that matches intuition for the components we studied.

Table 31 shows that faults in the arrays of the data prefetcher either cause only performance errors or are benign. Faults in its arrays that are excited can lead to cache pollution, which causes performance fluctuations. For example, when a fault resides in the load address field of the data prefetcher and the corresponding entry is activated (trained, locked on a stride, and actively issuing prefetches), this will lead to modification of the number of issued prefetch requests and/or corruption of the load address of a prefetch request. Thus, in both cases, the cache will be polluted and performance will slow.

Table 31: Data prefetcher array fault classification (Average per component for all benchmarks and all injected faults).

Field	Slowdown error (%)	Speedup error (%)	Benign error (%)
Load Address	23.4	27.3	49.2
Tag	20.4	18.2	61.3
Stride	23.9	26.5	49.6
Confidence	16.1	29.5	54.4
LRU	27.0	16.1	56.9
Valid	22.2	8.7	69.1
PRQ – Load Address	44.7	3.8	51.5
PIQ – Load Address	66.3	24.0	9.7
Average	**30.5**	**19.3**	**50.2**

The high concentration of benign faults in our experiments is because faults occur in entries that are not activated. To verify this, we studied the behaviour of a few benign faults from the execution of several benchmarks (bzip2, games, zeusmp); our finding is that the number of prefetch requests remains stable (i.e., equal to the golden run) because all these faults are not excited throughout the simulation runs.

The different fields of the data prefetcher arrays behave as Table 31 shows: benign errors range roughly between 10% and 70% across fields, while the performance errors category takes from about 30% to about 90% of the fault population. On average, 49.8% of the faults in the arrays of the data prefetcher lead to performance errors (30.5% slowdowns and 19.3% speedups) and 50.2% are benign. Finally, there are no crashes or exceptions generated by faults in the DP because prefetch requests to invalid/illegal addresses are dropped in our system; thus we do not include a crash column in the Table 31.

Table 32 shows that faults in the control part of the data prefetcher lead to many more performance faults (greater than 76%) than faults in the arrays (Table 31). Slowdown errors occur much more frequently than speedups, and are more massive compared to the array. A fault in the output logic that issues prefetcher requests actually disables the data prefetcher.

Table 32: Data prefetcher control fault classification.

Field	Slowdown error (%)	Speedup error (%)	Benign error (%)
Prefetcher table index-generation logic	32.9	14.9	52.2
Prefetcher table tag-search logic	50.0	36.2	13.8
Prefetcher table replacement logic	48.3	34.5	17.2
Stride calculation logic	31.0	44.8	24.1
Confidence calculation logic	51.7	29.0	19.3
Output logic that decides whether to issue Prefetch or not	100.0	0.0	0.0
PRQ - CAM multi-hits	33.3	25.3	41.4
PRQ - Head/tail pointer	43.7	27.6	28.7
PIQ - Head/tail pointer	42.5	40.2	17.2
Average	**48.2**	**28.1**	**23.7**

Table 33 shows that more than 80% of the faults in the instruction prefetcher array (eight buffers holding addresses to be prefetched) are not activated. This is in line with the corresponding large percentage of benign faults in the arrays of the data prefetcher (Table 31). A next-line instruction prefetcher will not utilize more than a few entries because it generates a prefetch on an Icache miss.

Table 33: Instruction prefetcher array fault classification.

Field	Slowdown Error	Speedup Error	Benign Error
Instruction prefetcher array	17.5%	0.8%	81.7%

4.5.2 Benchmark profiling: prefetch-friendly and –neutral

We profile the full set of SPEC CPU2006 benchmarks to measure the IPC impact of a fault-free L1 cache-stride data prefetcher. Table 34 presents the IPC speedup for each benchmark due to the stride data prefetcher. On average, the data prefetcher boosts IPC by 6.85%. However, performance improvement varies and depends on the stream of memory access patterns generated by each benchmark. For that reason, we classify benchmarks into two major categories: prefetch-friendly, in which the IPC change is greater than the average speedup across all SPEC CPU2006 benchmarks, and prefetch-neutral, in which the change is less than the average speedup. Eleven benchmarks are classified as prefetch-friendly and 18 are classified as prefetch-neutral. The impact of faults on performance significantly differs between the two groups of benchmarks.

Table 34: Per benchmark IPC speedup provided by the L1 data prefetcher (*milc IPC is slowed).

Prefetch-friendly benchmarks	IPC (%) speedup	Prefetch-neutral benchmarks	IPC (%) speedup
bzip2	19.99	perlbench	2.58
bwaves	10.14	gcc	1.89
gamess	21.51	mcf	0.16
zeusmp	9.10	milc	–4.11*
leslie3d	7.55	gromacs	2.86
dealII	9.61	cactusADM	0.49
soplex	9.50	namd	0.34
GemsFDTD	19.66	gobmk	0.60
libquantum	17.20	povray	0.42
tonto	15.91	calculix	4.73
wrf	34.59	hmmer	0.62
average	**15.887**	sjeng	0.07
		h2564ref	0.96
		lbm	4.66
		omnetpp	0.07
		astar	3.01
		sphinx3	0.67
		xalancbmk	3.80
		average	**1.780**
Overall average (%)		**6.85**	

4.5.3 Performance impact of faults

In this section, we measure the performance impact of hard faults injected only into the data prefetcher. Figure 36 shows the average and maximum IPC slowdown (due to faults) when one, three, and five faults are injected in the prefetch table along with the standard deviation; the upper diagram shows prefetch-friendly benchmarks and the lower diagram shows prefetch-neutral benchmarks. Figure 36 presents the average performance loss across all benchmarks (i.e., the prefetch-friendly benchmarks show a combined 3.049% IPC loss if we average the maximum IPC loss over all single fault runs per benchmark, 5.759% IPC loss over all triple faults, and 9.271% IPC loss over all quintuple faults). Thus, an L1 cache-stride data prefetcher can severely degrade microprocessor performance, up to 9.271% on average for the prefetch-friendly and up to 0.733% on average for the prefetch-neutral benchmarks, when the prefetcher table's SRAM cells suffer multiple hard faults.

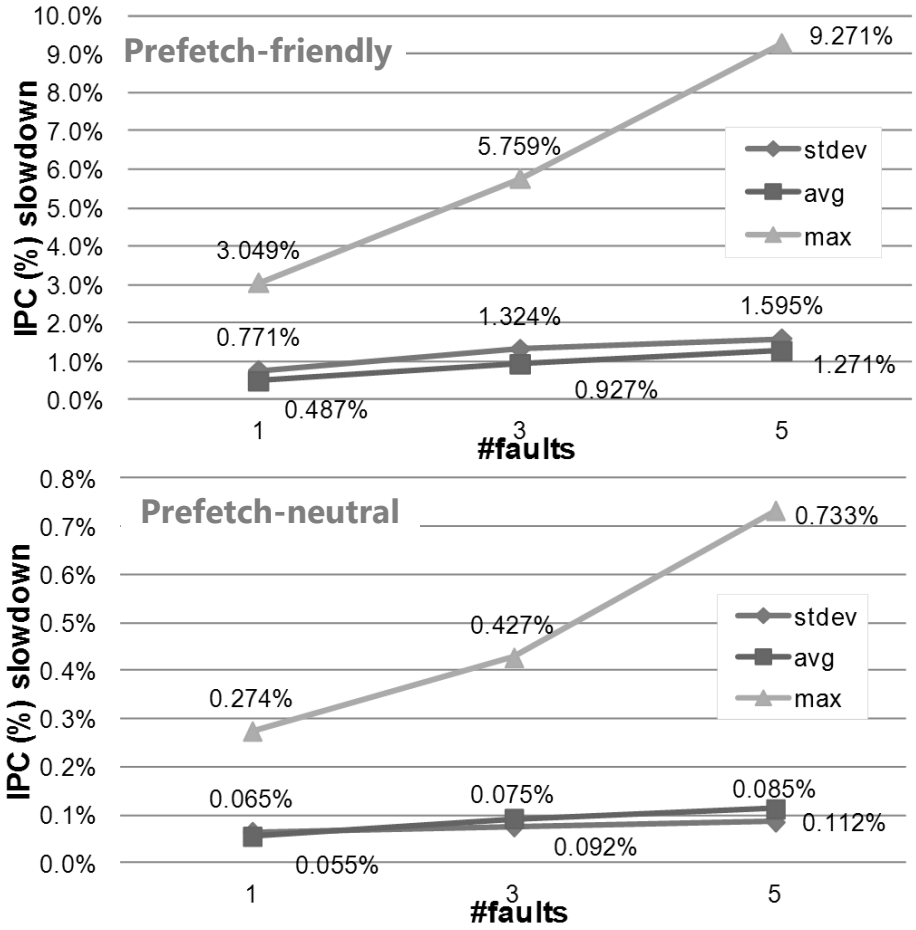


Figure 36: IPC loss for prefetch-friendly (upper graph) and –neutral (lower).

Table 35 shows the average (upper) and maximum (lower) normalized IPC slowdown for each SPEC CPU2006 benchmark when one, three, and five faults are injected. The benchmarks are grouped into prefetch-friendly (upper half) and -neutral (rt half), to identify any correlation between workloads and performance loss. The colours on each column depict the additional IPC loss relative to the fault-free model from the injection of one, three, and five faults. For example, on bzip2, the maximum IPC loss is 4.5% for a single injected fault, 4.8% for triple faults, and 17.1% for quintuple faults (i.e., the aggregation of single, triple, and quintuple IPC losses). As expected, the prefetch-friendly benchmarks show a greater IPC impact with the same number of faults compared to the prefetch-neutral. In particular, a fault-free prefetcher improves execution time of GemsFDTD by 20% and sphinx3 by 0.6% (Table 34). GemsFDTD suffers a maximum 17% IPC slowdown, while sphinx3 loses only 0.06% when quintuple faults are injected.

By further analysing the internal behaviour of the prefetcher, we found that the extent of the performance impact depends on the distribution of the training input addresses across the prefetch table entries. For example, Table 37 presents the activity of each prefetcher table entry for two benchmarks, bzip2 and gcc, and shows very different sensitivities to data prefetching (17% and 0.06%, respectively). gcc shows a much more uniform usage of the entries of the table, while bzip2 trains only seven entries (95% of training occurs on only three entries and the remaining four are trained only marginally). Thus, in gcc, the majority of the training addresses remain unaffected by the injected faults; even if they do access a faulty entry, the IPC impact is relatively small because of the lower average dynamic usage frequency. In bzip2, if the fault occurs in one of the

heavily used entries, the majority of training is affected, and so the IPC loss due to faults is much greater.

Table 35: Average, maximum, and standard deviation of IPC loss across all SPEC CPU2006 benchmarks when one, three, and five faults are injected into the prefetcher table. The 11 upper-most rows show the prefetch-friendly benchmarks, the next 18 show the prefetch-neutral benchmarks and the last row show the averages for the two categories.

	Benchmark	IPC (%) slowdown								
		1-fault			3-faults			5-faults		
		Avg	*Max*	*Stdev*	*Avg*	*Max*	*Stdev*	*Avg*	*Max*	*Stdev*
Prefetch friendly	bzip2	2.182	4.536	2.518	3.490	4.835	1.790	3.905	17.101	1.409
	Bwaves	0.261	0.087	0.405	0.278	0.934	0.491	0.294	0.974	0.499
	Gamess	0.009	0.506	0.005	0.668	7.728	3.447	1.035	18.533	4.246
	Zeusmp	0.003	1.350	0.060	0.019	1.424	0.060	0.083	1.512	0.060
	leslie3d	0.097	1.234	0.272	0.332	1.644	0.465	0.690	2.750	0.623
	dealII	0.033	0.168	0.011	0.035	3.291	0.020	0.041	4.446	0.028
	Soplex	0.442	0.572	0.001	0.688	0.981	0.242	0.696	2.543	0.350
	GemsFDTD	0.471	11.425	1.948	1.740	12.093	3.673	2.920	17.301	4.492
	libquantum	1.124	2.207	0.934	1.455	7.992	0.889	1.827	13.793	1.467
	tonto	0.615	7.176	2.012	0.865	15.515	2.312	1.160	15.524	2.613
	wrf	0.122	4.282	0.322	0.630	6.922	1.175	1.339	7.508	1.768
	Average	**0.487**	**11.425**	**0.771**	**0.927**	**15.515**	**1.324**	**1.271**	**18.533**	**1.595**
Prefetch neutral	perlbench	0.098	1.110	0.171	0.224	1.442	0.227	0.259	1.465	0.245
	gcc	0.009	0.033	0.006	0.016	0.047	0.013	0.021	0.065	0.018
	mcf	0.007	0.031	0.004	0.009	0.070	0.005	0.009	0.079	0.006
	milc	0.181	0.739	0.000	0.211	1.704	0.000	0.310	2.243	0.001
	gromacs	0.118	1.016	0.252	0.433	1.092	0.306	0.549	1.618	0.322
	cactusADM	0.161	0.204	0.124	0.241	0.396	0.127	0.268	0.723	0.150
	namd	0.008	0.019	0.007	0.009	0.084	0.006	0.009	0.148	0.007
	gobmk	0.010	0.018	0.009	0.011	0.021	0.008	0.012	0.028	0.008
	povray	0.007	0.132	0.006	0.011	0.146	0.008	0.013	0.488	0.008
	calculix	0.062	0.161	0.081	0.068	0.306	0.083	0.077	0.318	0.090
	hmmer	0.036	0.081	0.010	0.039	0.111	0.022	0.048	0.199	0.034
	sjeng	0.007	0.018	0.003	0.008	0.024	0.006	0.009	0.028	0.006
	h2564ref	0.007	0.028	0.007	0.008	0.028	0.007	0.009	0.156	0.007
	lbm	0.218	0.978	0.364	0.278	1.074	0.399	0.366	2.564	0.418
	omnetpp	0.025	0.088	0.027	0.031	0.286	0.021	0.036	0.681	0.019
	astar	0.013	0.219	0.028	0.026	0.261	0.042	0.047	0.273	0.069
	sphinx3	0.005	0.008	0.001	0.006	0.029	0.006	0.011	0.056	0.013
	xalancbmk	0.026	0.054	0.008	0.040	0.578	0.080	0.058	2.070	0.115
	Average	**0.055**	**1.110**	**0.065**	**0.092**	**1.704**	**0.075**	**0.112**	**2.564**	**0.085**
Overall average		**0.219**	**11.176**	**0.331**	**0.409**	**15.515**	**0.549**	**0.555**	**18.533**	**0.658**

To clarify the severity of the performance loss due to the faulty data prefetcher, Table 36 shows the actual IPC of the prefetch-friendly (upper half) and prefetch-neutral (lower half) SPEC CPU2006 benchmarks for the following CPU core configurations: (a) L1 cache stride data prefetcher disabled, (b) fault-free L1 cache stride data prefetcher enabled, and (c) faulty data prefetcher with 1, 3, and 5 faults injected into the prefetch table array (maximum IPC loss for each fault class). In particular, 8 out of 29 benchmarks (31%) lost the performance improvement gained from integrating the data prefetcher in the baseline CPU design. For example, on bzip2 the IPC without the data prefetcher was 1.074. When quintuple faults were injected into the prefetch table array, IPC was

reduced to 1.064 (similar behaviour was seen in: gamess, GemsFDTD, libquantum, tonto, cactusADM, povray, sjeng and omnetpp).

Table 36: IPC values for prefetch-friendly and –neutral benchmarks, without data prefetcher, with the data prefetcher enabled and with 1, 3 and 5 faults injected into the prefetch table array.

Benchmark		IPC				
		w/o data prefetcher	w/ data prefetcher	1-fault	3-faults	5-faults
Prefetch friendly	bzip2	1.074	1.284	1.226	1.221	1.064
	bwaves	0.648	0.714	0.713	0.707	0.707
	gamess	1.743	2.118	2.107	1.954	1.725
	zeusmp	0.971	1.059	1.045	1.044	1.043
	leslie3d	0.784	0.844	0.834	0.830	0.821
	dealII	0.988	1.083	1.081	1.047	1.035
	soplex	0.547	0.599	0.596	0.593	0.584
	GemsFDTD	0.528	0.634	0.561	0.558	0.525
	libquantum	0.395	0.463	0.453	0.426	0.395
	tonto	1.601	1.855	1.722	1.567	1.567
	wrf	0.793	1.068	1.022	0.994	0.988
Prefetch neutral	perlbench	1.663	1.705	1.686	1.680	1.680
	gcc	0.693	0.707	0.707	0.707	0.707
	mcf	0.208	0.209	0.209	0.209	0.209
	milc	0.757	0.726	0.721	0.714	0.710
	gromacs	0.944	0.971	0.961	0.960	0.955
	cactusADM	1.457	1.464	1.461	1.458	1.453
	namd	1.550	1.555	1.555	1.554	1.553
	gobmk	1.234	1.242	1.242	1.242	1.242
	povray	1.130	1.135	1.134	1.133	1.129
	calculix	1.127	1.181	1.179	1.177	1.177
	hmmer	1.169	1.176	1.175	1.175	1.174
	sjeng	1.180	1.181	1.181	1.181	1.180
	h2564ref	1.549	1.563	1.563	1.563	1.561
	lbm	0.686	0.718	0.711	0.710	0.700
	omnetpp	0.505	0.505	0.505	0.504	0.502
	astar	0.914	0.941	0.939	0.939	0.938
	sphinx3	1.365	1.374	1.374	1.374	1.373
	xalancbmk	1.102	1.144	1.143	1.137	1.120

By further analysing the internal behaviour of the prefetcher, we found that the extent of the performance impact that faults have, depends on the distribution of the training input addresses across the prefetch table entries (apart from the prefetch-friendliness of the workload).

Table 37: Training activity (X) of the prefetch table entries. Number of prefetch table entries that handle less than 50%, less than 75% and 100% of the memory traffic training the prefetcher.

Benchmark		Prefetch Table Entries Training Activity		
		X ≤ 50%	X ≤ 75%	X =100%
Prefetch friendly	bzip2	1	2	7
	bwaves	4	7	12
	gamess	17	17	17
	zeusmp	9	25	62
	leslie3d	9	17	47
	dealII	1	2	24
	soplex	8	18	63
	GemsFDTD	4	7	32
	libquantum	1	2	5
	tonto	3	6	40
	wrf	12	22	59
	Average	**6**	**11**	**33**
Prefetch neutral	perlbench	3	4	63
	gcc	7	20	64
	mcf	3	5	47
	milc	1	2	23
	gromacs	12	22	58
	cactusADM	4	6	11
	namd	2	8	57
	gobmk	3	9	64
	povray	6	14	64
	calculix	4	8	63
	hmmer	6	10	59
	sjeng	3	6	64
	h2564ref	8	20	64
	lbm	1	2	32
	omnetpp	1	2	31
	astar	4	6	47
	sphinx3	1	2	57
	xalancbmk	2	3	62
	Average	**4**	**8**	**51**

Table 37 presents the activity of each entry of the prefetcher table for the prefetch-friendly (upped half) and prefetch-neutral benchmarks (lower half). For example, in GemsFDTD, 4 entries are trained by 50%, 7 entries by 75%, and 32 entries by 100% of the load/store address traffic (GemsFDTD speedup is 20.07%, while slowdown is up to 17.30% when quintuple faults are injected). The observations from Table 37 are the following:

1. When a fault occurs in heavily used entries, the majority of training will be affected, and so the maximum IPC loss will be much greater. For example, in libquantum, a single entry is trained by 50%, 2 entries by 75%, and 5 entries by 100% of the load/store address traffic (libquantum speedup is 17.20%, while slowdown is up to 13.80% when quintuple faults are injected).

2. Benchmarks with uniform usage of the prefetch table entries have lower probability of massive IPC loss. For example, the training address stream generated by gcc is distributed across the entries of the prefetch table (7 entries are trained by 50% of the traffic while all 64 entries are trained by 100% of the memory traffic. Therefore, the probability of polluting the majority of the training addresses by a given number of injected faults is low leading to a very low max IPC loss (-0.065% with 5 faults; see Table 4)
3. Table 38 shows the number of issued prefetch requests per 1,000 committed instructions and the L1 cache miss rate (misses per 1,000 committed instructions, or MPKI) for the fault-free and faulty cases for each group of injected faults. As we can observe in Table 38, the faulty prefetcher is throttled because the faults reduce the number of training events. As a result, the number of issued prefetch requests drops for all benchmarks (on average, the number of issued prefetch requests drops from 22 to 20 per 1,000 committed instructions); therefore, performance gains due to prefetching are lower (the average L1 cache miss rate roughly increases from 26 to 27 MPKI in the quintuple injected fault scenario). The data in Table 6 also illustrate the greater performance sensitivity of the prefetch-friendly benchmarks to faults. Faults in the prefetch table change the prefetch addresses sent to memory, which in turn increases the L1 cache miss rate and hurts IPC.

We also looked at a variety of microarchitectural events that can be used to identify when faults in the data prefetcher lead to IPC loss. We found that the number of off-range strides is such an event and is triggered when the difference between the trained address and a new memory address is outside the legal stride limits. As a result, the incoming memory address is dropped and fails to train the stride data prefetcher.

Table 38: Average L1 cache miss rate and average prefetch issue rate with one, three, and five faults injected.

	Benchmark	fault-free		1-fault		3-faults		5-faults	
		Prefetch Issue Rate	Miss Rate	Prefetch Issue Rate	Miss Rate	Prefetch Issue Rate	Miss Rate	Prefetch Issue Rate	Miss Rate
Prefetch friendly	bzip2	3.81	16.15	3.80	16.42	3.65	16.44	3.56	16.45
	bwaves	22.88	121.27	22.20	121.65	21.50	121.84	17.65	121.98
	gamess	9.84	6.02	9.84	6.03	9.68	6.30	9.51	6.57
	zeusmp	45.54	15.12	44.15	15.33	42.50	15.41	40.49	15.51
	leslie3d	18.04	26.84	18.00	26.85	17.65	26.88	17.24	26.90
	dealII	6.43	4.57	6.41	4.57	6.38	4.58	6.37	4.59
	soplex	8.44	16.61	8.21	16.62	8.00	16.73	7.91	16.88
	GemsFDTD	40.65	43.01	38.91	42.34	37.76	42.63	37.21	42.95
	libquantum	25.22	14.28	24.84	14.31	24.10	14.36	23.37	14.40
	tonto	39.22	16.64	38.90	16.80	38.35	17.09	37.74	17.33
	wrf	32.96	7.64	32.70	7.70	32.12	7.86	31.44	8.05
	Average	**23.00**	**26.20**	**22.54**	**26.24**	**21.97**	**26.37**	**21.14**	**26.51**
Prefetch neutral	perlbench	14.70	2.82	14.58	2.89	14.31	2.89	14.17	2.92
	gcc	16.22	11.54	15.99	11.54	15.98	11.55	15.76	11.56
	mcf	8.05	219.37	7.88	219.37	7.80	219.39	7.39	219.39
	milc	35.15	38.96	34.39	39.37	33.64	40.04	32.40	40.74
	gromacs	4.33	31.49	4.30	31.67	4.21	31.68	4.12	31.70
	cactusADM	11.07	0.001	11.02	0.001	10.90	0.001	10.01	0.001
	namd	0.07	0.31	0.07	0.31	0.07	0.31	0.06	0.31
	gobmk	7.18	6.92	7.02	6.92	6.88	6.93	6.79	6.93
	povray	3.91	37.82	3.83	37.83	3.76	37.85	3.74	37.86
	calculix	45.74	9.45	45.18	9.45	43.84	9.45	43.27	9.46
	hmmer	19.08	6.29	18.84	6.36	18.54	6.43	17.97	6.59
	sjeng	100.20	4.03	100.02	4.05	99.86	4.06	99.19	4.06
	h2564ref	13.40	5.41	13.14	5.41	12.74	5.41	12.72	5.41
	lbm	17.69	42.00	17.44	42.04	17.03	41.90	16.60	41.98
	omnetpp	0.63	52.01	0.63	52.05	0.62	52.06	0.61	52.07
	astar	10.85	0.38	10.85	0.38	10.83	0.38	10.81	0.38
	sphinx3	3.01	0.83	2.93	0.83	2.93	0.83	2.90	0.83
	xalancbmk	57.92	23.82	55.30	23.98	55.30	24.06	54.31	24.18
	Average	**20.51**	**27.41**	**20.27**	**27.47**	**19.96**	**27.51**	**19.61**	**27.58**
Overall average		**21.76**	**26.95**	**21.41**	**27.00**	**20.96**	**27.08**	**20.37**	**27.17**

Table 39,Table 40 present the number of off-range stride events per 1K committed instructions for the fault-free and the faulty microprocessor models (i.e. 1, 3 and 5 faults injected into the prefetch table array). In particular, multiple permanent faults increase the amount of off-range stride events up to 25% for the prefetch-friendly benchmarks (off-range stride rate increased from 118.7 to 148.3 per 1K commits), up to 8% for the prefetch-neutral benchmarks (ranging from 231.3 to 249.3 per 1K commits) and across all SPEC CPU2006 benchmarks up to 14% (from 175.0 to 198.8 per 1K commits). The number of off-range strides increases for two reasons:

- Faults injected in tag field can result in unexpected table hits. Therefore, memory instructions with completely different access patterns are compared and produce out-of bound strides.
- Faults in the previous load address sub-field increase the amount of off-range stride occurrences.

Table 39: Number of off-range stride events for the prefetch-friendly benchmarks with a fault-free data prefetcher and with 1, 3, 5 faults injected in the prefetch table.

Off-range stride (per 1K commits)				
Prefetch-friendly benchmarks	**fault-free**	**1-fault**	**3-faults**	**5-faults**
bzip2	80.668	84.456	90.826	98.275
bwaves	199.938	215.250	217.930	232.224
gamess	0.619	8.729	14.790	21.116
zeusmp	58.051	62.352	69.640	78.678
leslie3d	162.893	164.042	173.872	183.859
dealII	161.653	167.454	185.109	197.482
soplex	185.109	197.482	253.957	256.547
GemsFDTD	231.682	238.504	249.156	257.502
libquantum	199.884	210.384	226.992	240.348
tonto	7.225	12.792	22.158	34.330
wrf	19.070	22.450	26.061	31.600
Average	**118.799**	**125.808**	**139.135**	**148.360**

Table 40: Number of off-range stride events for the prefetch-friendly benchmarks with a fault-free data prefetcher and with 1, 3, 5 faults injected in the prefetch table.

Off-range stride (per 1K commits)				
Prefetch-neutral benchmarks	**fault-free**	**1-fault**	**3-faults**	**5-faults**
perlbench	175.905	179.268	184.975	190.678
gcc	451.466	454.177	463.396	474.071
mcf	29.761	33.629	47.334	60.829
milc	126.264	131.267	139.166	149.406
gromacs	305.408	305.732	311.365	317.117
cactusADM	154.198	155.932	161.514	170.552
namd	345.105	346.445	348.692	352.429
gobmk	322.921	325.778	331.724	336.996
povray	344.224	346.796	350.445	357.719
calculix	56.477	57.814	62.998	68.642
hmmer	219.894	226.234	239.336	250.625
sjeng	113.718	128.047	141.591	151.138
h2564ref	181.039	181.074	188.750	195.844
lbm	413.886	414.130	417.924	421.734
omnetpp	314.216	315.534	321.305	327.514
astar	106.356	120.792	125.509	133.075
sphinx3	309.220	310.852	315.971	321.027
xalancbmk	194.846	201.784	207.318	208.105
Average	**231.383**	**235.293**	**242.184**	**249.305**

The severe impact of faults on the performance of SPEC CPU2006 benchmarks indicates the need to integrate fault detection schemes for stride prefetchers. Figure 37 presents the average percentage of increase on the off-range stride occurrences (y-axis) to the amount of injected faults (x-axis). It is evident that the number of off-range stride events linearly correlates to the number of faults in the prefetcher and therefore to IPC loss. Monitoring the behaviour of off-range

occurrences throughout workload execution, can steer the implementation of permanent error detection mechanisms.

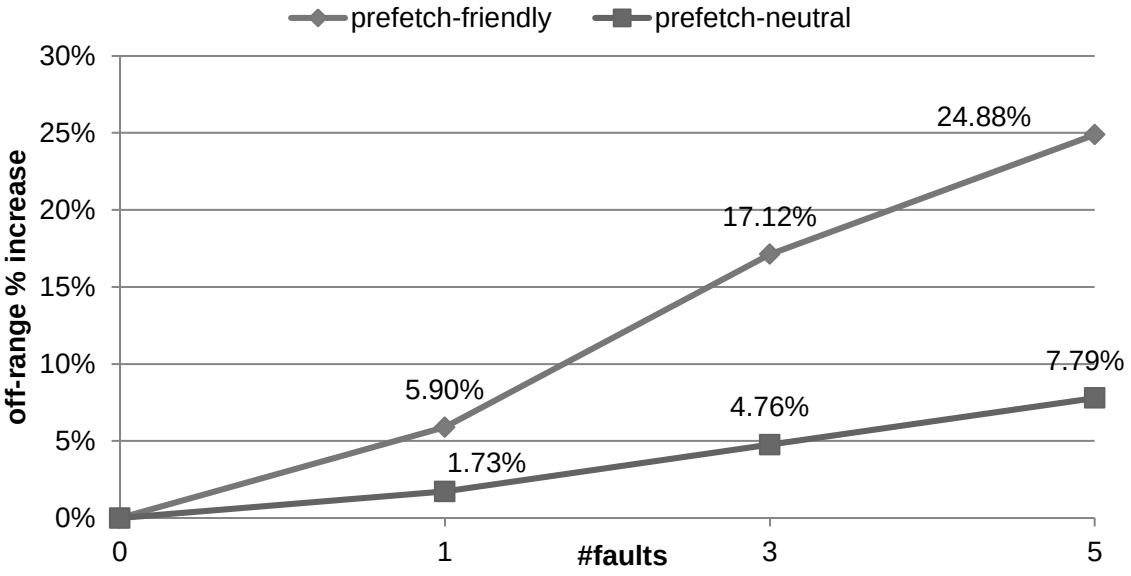
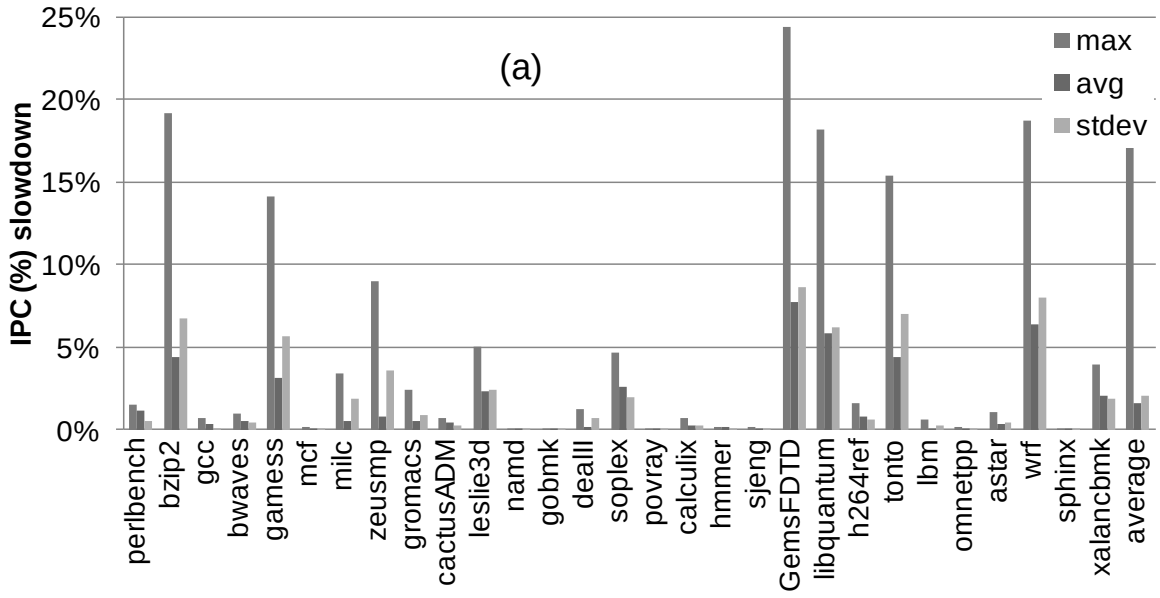


Figure 37: Correlation of off-range (%) increment with 1, 3 and 5 faults injected.

Apart from the prefetch table fault injections and analysis, we performed a similar injection campaign in the input and request queues of the data prefetcher (PIQ and PRQ). Due to the small size of PIQ and PRQ (8 entries each), we injected only single faults in them. This was sufficient to demonstrate the severe impact on performance that hard faults on these queues can have on IPC. Figure 38 shows the maximum and average IPC slowdowns and the standard deviation for single faults injected into the PIQ and PRQ per benchmark.



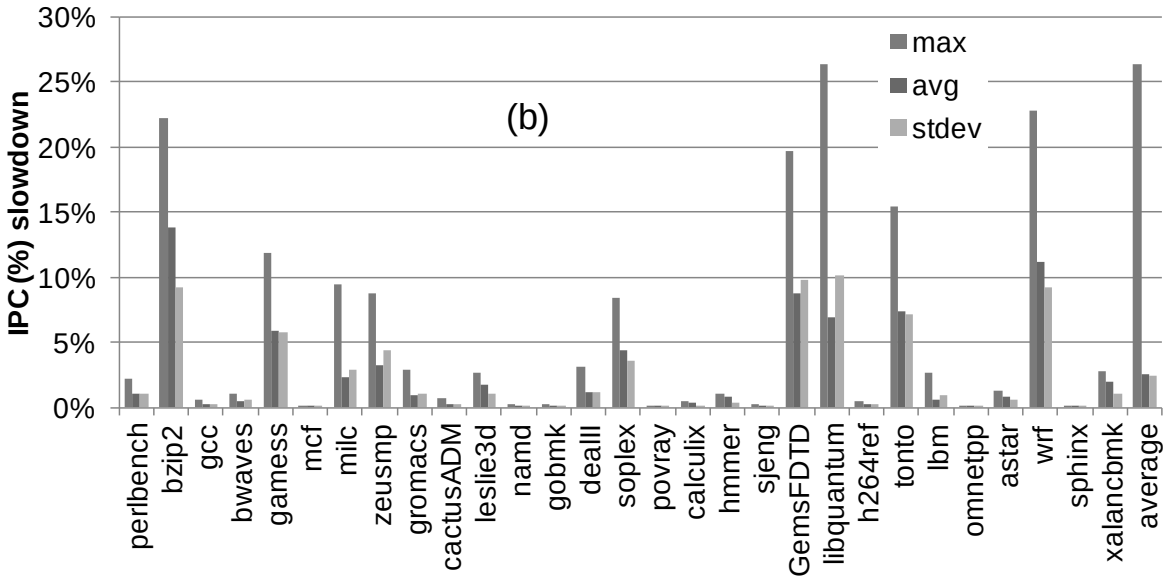


Figure 38: Average and maximum IPC slowdowns and standard deviations for the fault-free and faulty (a) PIQ and (b) PRQ when single faults are injected.

Across all 29 benchmarks, the average IPC loss (1.5% and 2.5% for PIQ and PRQ, respectively) and maximum IPC loss (24.3% and 26.3% for PIQ and PRQ, respectively) are significantly higher than that of the prefetch table because a large number of training addresses (buffered in PIQ) and prefetch requests (queued in PRQ) are likely to be polluted by a single hard fault. Table 41 shows the IPC change for the prefetch-friendly (upped half) and prefetch-neutral benchmarks (lower half). It is evident that having a faulty PRQ or PIQ severely slows-down performance (13 benchmarks out of the 29 lost the speedup gained by the data prefetcher due to a faulty PRQ and 11 due to a faulty PIQ).

Table 41: Average IPC for prefetch-friendly and –neutral benchmarks, without the data prefetcher, with a fault-free data prefetcher and with single faults injected into the prefetch input and request queue.

	Benchmark	Average IPC			
		w/o data prefetcher	*w/ data prefetcher*	*1-fault PIQ*	*1-fault PRQ*
Prefetch friendly	bzip2	1.074	1.289	1.041	1.000
	bwaves	0.648	0.714	0.707	0.706
	gamess	1.743	2.118	1.819	1.866
	zeusmp	0.971	1.059	0.964	0.965
	leslie3d	0.784	0.844	0.802	0.821
	dealII	0.988	1.083	1.069	1.048
	soplex	0.547	0.599	0.571	0.547
	GemsFDTD	0.528	0.632	0.477	0.507
	libquantum	0.395	0.463	0.378	0.341
	tonto	1.601	1.855	1.569	1.567
	wrf	0.793	1.068	0.868	0.824
Prefetch neutral	perlbench	1.663	1.705	1.679	1.666
	gcc	0.693	0.707	0.702	0.702
	mcf	0.208	0.209	0.208	0.208
	milc	0.757	0.726	0.701	0.657
	gromacs	0.944	0.971	0.947	0.943
	cactusADM	1.457	1.464	1.454	1.453
	namd	1.550	1.555	1.554	1.551
	gobmk	1.234	1.242	1.240	1.239
	povray	1.130	1.135	1.133	1.133
	calculix	1.127	1.181	1.179	1.174
	hmmer	1.169	1.176	1.174	1.163
	sjeng	1.180	1.181	1.179	1.178
	h2564ref	1.549	1.563	1.538	1.556
	lbm	0.686	0.718	0.714	0.698
	omnetpp	0.505	0.505	0.504	0.504
	astar	0.914	0.941	0.930	0.928
	sphinx3	1.365	1.374	1.372	1.372
	xalancbmk	1.102	1.144	1.098	1.113

The fault location determines the extent of the performance impact. Figure 39 shows the average utilization of each entry of the PRQ and PIQ (% of times a given entry of the queue is utilized). In particular, the PIQ entries are utilized uniformly across all benchmarks with the exception of the top 2 entries. On the contrary, the top three entries in PRQ are utilized 95% of the time across all benchmarks. Therefore, faults that reside in the rear entries of the queue have minimal impact on performance. A different entry allocation scheme (e.g. advance a write pointer every time we insert a new prefetch request in the PRQ) can distribute the payload uniformly for each of the queues and reduce the IPC loss. Detecting faults in the PIQ could be done by monitoring off-range strides (they should increase) while detecting faults in the PRQ can be accomplished by monitoring addresses that cross the 4k page boundary.

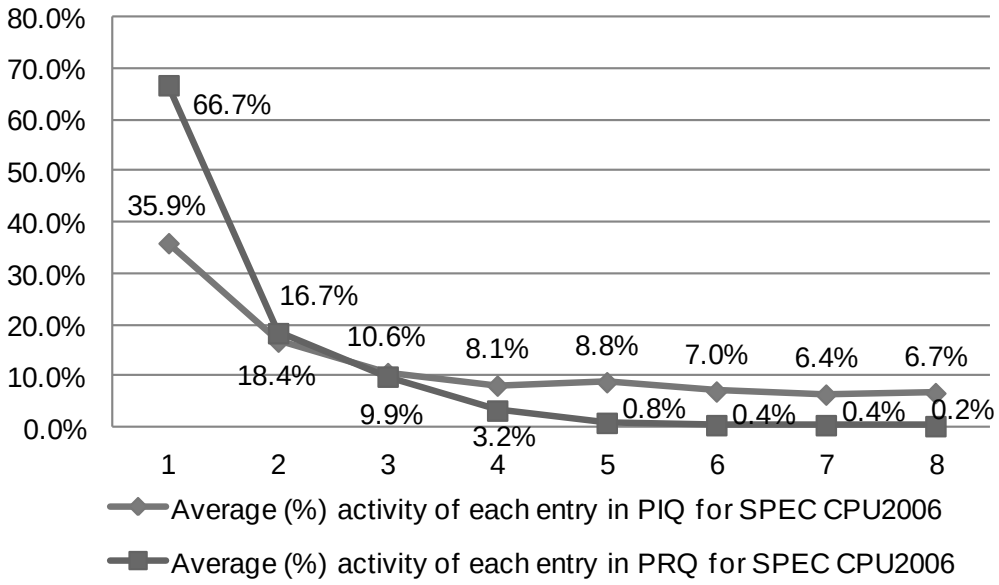


Figure 39: Utilization of the PIQ and PRQ entries across all SPEC CPU2006 benchmarks.

4.5.4 Performance Variability

In this section, we examine the performance variability across identical CPU cores in the presence of multiple faults in their data prefetchers. Our findings (resulting from the performance impact analysis and the detailed results of the previous subsection) are the following:

- There is a large variation of IPC loss even under the presence of different single faults (up to 7% for single faults in the prefetch table, 24% for the PIQ, and 26% for the PRQ) in the data prefetcher. This finding holds when all cores are affected by the same number of faults.
- The difference in IPC slowdown for different numbers of faults (1 to 5) across the cores ranges between 0.005% and 17% compared to the fault-free IPC, when considering only faults in the prefetch table. The same range is 0.01 to 24% for PIQ faults and 0.02% to 26% for PRQ faults.
- The difference between the best- and worst-case performance for single and multiple faults is not due to outlier behaviour. The standard deviation of the IPC loss on the prefetch-friendly benchmarks due to faults in the prefetcher table is 0.7%, 1.3%, and 1.6% for single, triple, and quintuple faults, respectively.

As Table 35 shows, certain benchmarks have even larger stdev values: for example the stdev of bzip2 benchmark with single fault injected is 2.5%, while the gamess benchmark stdev is 3.4% and 4.2% for triple and quintuple faults, respectively.

The standard deviation of IPC loss for the prefetch-neutral benchmarks is 0.06%, 0.07%, and 0.08% (Table 35). The standard deviation of the IPC drop for all single faults injected into PIQ and PRQ is 2.0% and 2.4%, respectively. The key message here is that hard faults in data prefetchers significantly increase inter-core performance variability.

4.6 Resiliency of branch prediction unit

4.6.1 Classification of faults

In this sub-section, we present the fault classification in the branch prediction unit component (we pick a representative component from each structure of predictors integrated into the BPU). For confidence (95%) and error margin (5%), 2,604 faults masks are sampled and injected into the arrays (Table 42).

Table 42: Distribution of the injected faults on the branch prediction unit.

Component	Field	#Injected faults
Branch predictor array	Bi-modal predictor	660
	cBTB – Branch address	662
	cBTB – Tag	660
	RAS	403
Branch predictor control	Bi-modal/Two-level/Meta-predictor index-generation logic	64
	Bi-modal/Two-level direction counter adder	2
	Global history register shift operation to append new values	2
	Meta-predictor selection logic	2
	Meta-predictor update logic	2
	RAS head/tail pointer value	10
	RAS head/tail pointer update logic	5
	RAS annulment bit	2
	RAS overflow bit	2
	RAS underflow bit	2
	cBTB index-generation logic	10
	cBTB tag-search logic	2
	cBTB replacement logic	2
	Global BPU logic	4
Total	**2,385 array faults + 111 control faults**	**2,495**

The following tables break down the fault classifications for the most representative sub-components of the branch prediction unit (Bimodal Predictor, cBTB, RAS). We separately present results for the arrays and the control parts. No output error has been observed in our experiments and thus we have omitted columns for output errors (all are 0%); this is an expected result that matches intuition for the components we studied. Table 43 shows that more than 55% of the faults in the BPU arrays are performance faults; about 43% are benign, while the RAS exhibits the only case in which crashes are reported (6.1%) due to the generation of illegal addresses. A similar behaviour was expected for the faults in the cBTB array (tag and address) for the same reason (invalid address generation). In our experiments, we have not observed such cases in the cBTB, most likely because the BTB array is significantly larger than the RAS array. It is therefore probable that the injected faults leading to invalid addresses in the BTB array are not excited by the benchmarks.

Table 43: Branch predictor unit array fault classification.

Field	Slowdown Error (%)	Speedup Error (%)	Benign Error	Crash
Bi-modal Predictor	40.3	34.4	25.3	0.0
cBTB – Branch Address	9.0	43.3	47.7	0.0
cBTB – Tag	19.8	31.0	49.2	0.0
RAS	36.0	8.5	49.4	6.1
Average	**26.3**	**29.3**	**42.9**	**1.5**

Table 44 shows that faults in the control part of the BPU, which lead to more performance faults (greater than 60%) than faults in the arrays (Table 43). Slowdown errors occur much more frequently than speedups, and are more massive compared to the array. Furthermore, RAS-related control logic can lead to simulator crashes because they generate invalid addresses, which is also the case in the RAS array (Table 43).

Table 44 Branch Predictor Unit Control Fault Classification.

Field	Slowdown Error (%)	Speedup Error (%)	Benign Error (%)	Crash (%)
Bi-modal/Two-level/Meta-predictor index generation logic	29.3	16.0	54.7	0.0
Bi-modal/Two-level direction counter adder	100.0	0.0	0.0	0.0
Global history register shift operation to append new values	98.3	1.7	0.0	0.0
Meta-predictor selection logic	84.5	10.3	5.2	0.0
Meta-predictor update logic	27.6	20.7	51.7	0.0
RAS head/tail pointer value	76.6	7.6	13.8	2.1
RAS head/tail pointer increment/ decrement logic	2.1	2.1	29.7	66.2 %
RAS annulment bit	7.4	88.9	3.7	0.0
RAS overflow bit	22.4	6.9	70.7	0.0
RAS underflow bit	6.9	89.7	3.4	0.0
cBTB index-generation logic	71.6	0.9	27.6	0.0
cBTB tag-search logic	0.0	0.0	0.0	100.0
cBTB replacement logic	1.7	0.0	98.3	0.0
Global BPU logic	46.9	53.1	0.0	0.0
Average	**41.1**	**21.3**	**25.6**	**12.0**

4.6.2 Performance Impact of faults

In this section we measure the performance loss and variability due to hard faults. According to the analysis of Section 4.2.2, we inject multiple faults of different group sizes (1 and 5 to 25 in steps of 5 faults) in the five large arrays (Bimodal, Two-level, Meta predictors, cBTB and iBTB) while for the two smaller structures (RAS and BHR) we inject single faults.

Figure 41 shows the AVG and max IPC slowdown for the five arrays. Each graph shows the average across all 11 benchmarks with up to 25 faults injected. Using [67] we compute fault populations for 99% confidence and 3% error margin. The calculations lead to a total of 1176 different single faults which are used for the injections (110 faults for each of the Bimodal, Two-level, Meta; 490 for the cBTB; 290 for the iBTB ; 50 for the RAS; and 16 for the BHR). In multiple fault injections we apply sets of randomly selected faults to each component with the exception of RAS and BHR where we inject single faults because of their small size. Each group is constructed by randomly selecting (uniform distribution) from the set of 1176 single faults. For each multiple fault group size (5, …, 25) we run 1200 different injection experiments for a total of 6000 multiple fault injections. All faults are injected in different array entries because our mechanisms can protect from any number of faults per entry.

Table 45: Mispredictions per 1K instructions in fault-free BP and BTBs (* = very few misses).

Benchmark	Branch predictor	cBTB	iBTB
perlbench	1.39	0.11	0.02
bzip2	6.49	0.00*	0.00*
gcc	12.98	0.14	0.00*
mcf	30.39	0.16	0.00*
gobmk	20.00	0.07	0.00*
dealII	1.71	0.00*	0.00*
soplex	14.02	0.00*	0.00*
povray	5.86	0.06	0.00*
lbm	0.05	0.00*	0.00*
omnetpp	5.85	0.03	0.00*
xalancbmk	3.53	0.10	0.00*
Average	9.30	0.06	0.00*

To better interpret the multiple fault injection graphs of Figure 41, Table 45 shows the number of mispredictions per 1K instructions for BP, iBTB and cBTB while Table 46 shows the branch dynamic frequencies.

Table 46: Dynamic branch instructions per 1K instructions that use the Bimodal predictor, Two-level predictor, cBTB and iBTB (*= very low activity).

benchmark	Bimodal	Two-level	cBTB	iBTB
perlbench	179.79	28.31	245.5	12.81
bzip2	136.35	34.46	181.9	0.00*
gcc	223.39	78.82	364.7	7.65
mcf	367.29	193.46	625.4	0.00*
gobmk	210.38	95.21	350.8	0.00*
dealII	36.40	35.52	96.42	0.32
soplex	271.86	43.52	326.7	0.00*
povray	85.49	59.82	195.9	8.50
lbm	8.44	2.89	63.95	0.00*
omnetpp	154.15	63.68	264.8	7.08
xalancbmk	193.75	43.88	269.7	20.97
Average	**169.75**	**61.78**	**271.4**	**5.21**

Among the three components of the branch predictor (Bimodal, Two-level, Meta) multiple faults in the Bimodal and Meta lead to severe IPC loss (first and third

diagrams of Figure 41). For the Bimodal, the IPC impact is high because it drives the prediction decision 78% of the time. The Two-level predictor is less frequently used and thus faults in it have smaller impact on IPC (second diagram). Faults in the Meta predictor, consistently have a large IPC impact because it is always used. As Table 45 shows, the frequency of fault-free mispredictions in the two BTBs is very small so the impact of faults is negligible too.

Figure 42 shows the extra mispredictions due to faults over the fault-free case in the predictors (an n % point means that if the fault-free case has a k % misprediction ratio, the faulty has k+n %). Multiple faults in the Bimodal and the Meta predictor lead to significantly higher misprediction ratios and IPC loss as shown in Figure 41. Multiple faults in the two-level predictor lead to very limited IPC loss because it is used infrequently by the Meta predictor.

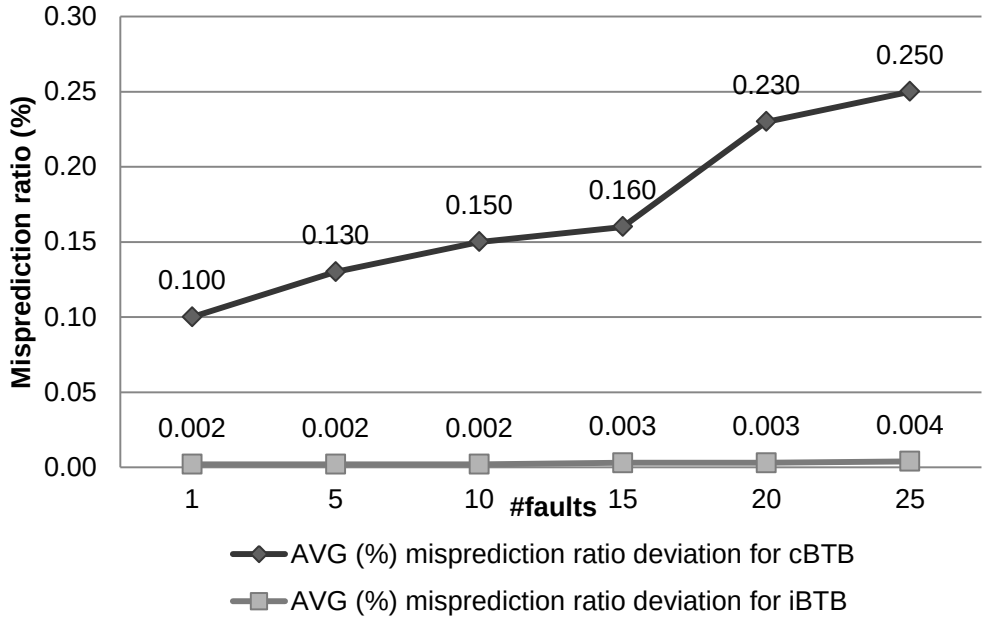
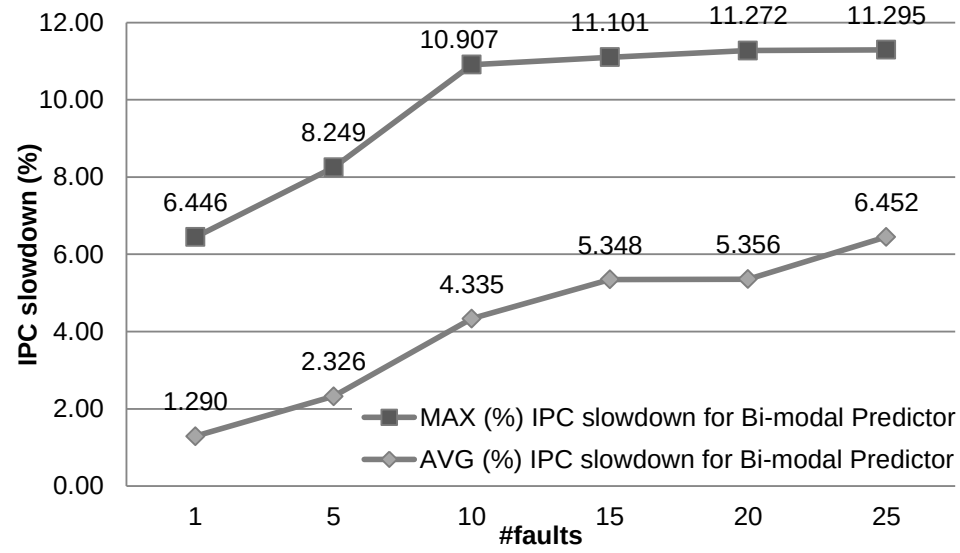
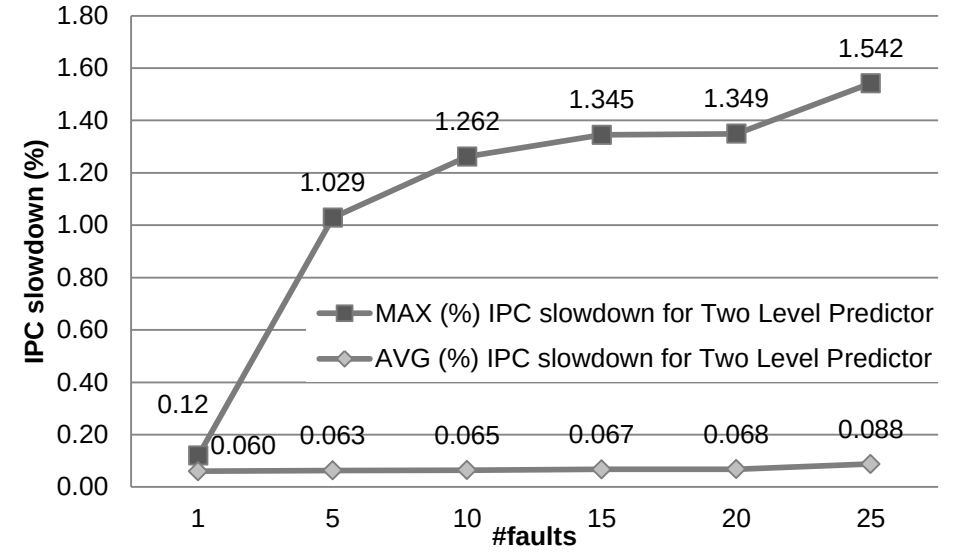


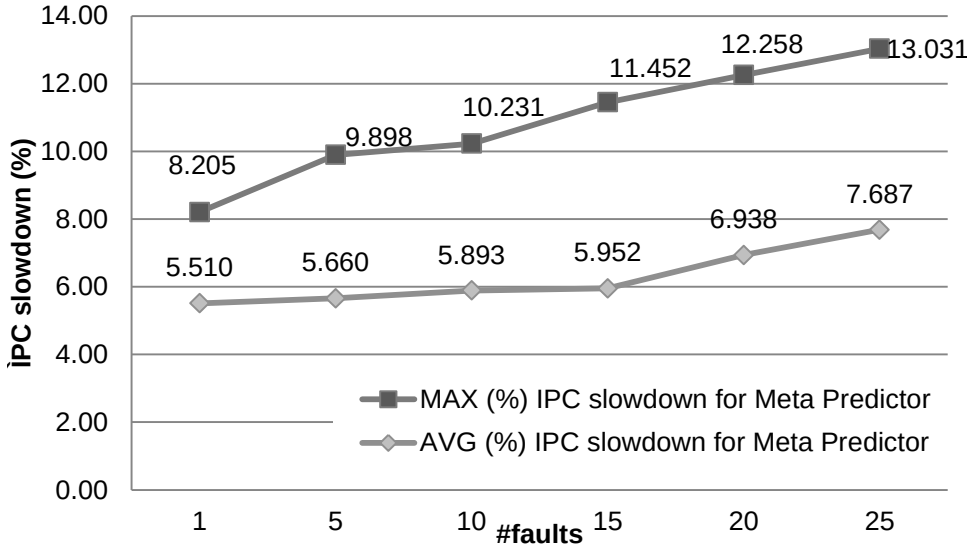
Figure 40: Extra misprediction ratio (%) (on top of fault-free case) caused by multiple faults in BTBs.

The BTBs are minimally affected by faults, as the last graphs of Figure 41 show, due to the low number of extra mispredictions (Figure 40). The iBTB's lack of IPC sensitivity is due to low frequency of indirect branches (Table 46) and fault masking in the high order bits of the target address which does not change often. Although the cBTB is much more frequently used (Table 46), IPC is less sensitive to cBTB mispredictions because (a) only a small number of cBTB entries are utilized (Table 47), (b) as in iBTB, a lot of faults are masked due to infrequently changing high address bits and (c) the cBTB misprediction penalty is much lower than that of the BP because the targets of conditional and unconditional-direct branches are verified in the front end of the processor.

Despite the small impact of faults in the Two-level predictor and the two BTBs, we include them in our evaluation because in workloads with a larger code footprint, the impact of multiple faults will be higher. Table 48 shows the max and average IPC slowdown for single faults in the RAS and BHR. The results clearly justify the need for protection. A single faulty cell in the RAS seriously degrades its prediction accuracy; IPC slowdown can be more than 16% when the call stack depth is low because of high reuse of the same faulty entry. Similarly, a single faulty bit in the BHR degrades performance significantly by affecting the accuracy of branch predictors.


12.00
10.00
8.00
6.00
4.00
2.00
0.00
IPC slowdown (%)
6.446
8.249
10.907
11.101
11.272
11.295
1.290
2.326
4.335
5.348
5.356
6.452
MAX (%) IPC slowdown for Bi-modal Predictor
AVG (%) IPC slowdown for Bi-modal Predictor
1
5
10
15
20
25
#faults


1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
IPC slowdown (%)
0.12
1.029
1.262
1.345
1.349
1.542
0.060
0.063
0.065
0.067
0.068
0.088
MAX (%) IPC slowdown for Two Level Predictor
AVG (%) IPC slowdown for Two Level Predictor
1
5
10
15
20
25
#faults


14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
ÌPC slowdown (%)
8.205
9.898
10.231
11.452
12.258
13.031
5.510
5.660
5.893
5.952
6.938
7.687
MAX (%) IPC slowdown for Meta Predictor
AVG (%) IPC slowdown for Meta Predictor
1
5
10
15
20
25
#faults

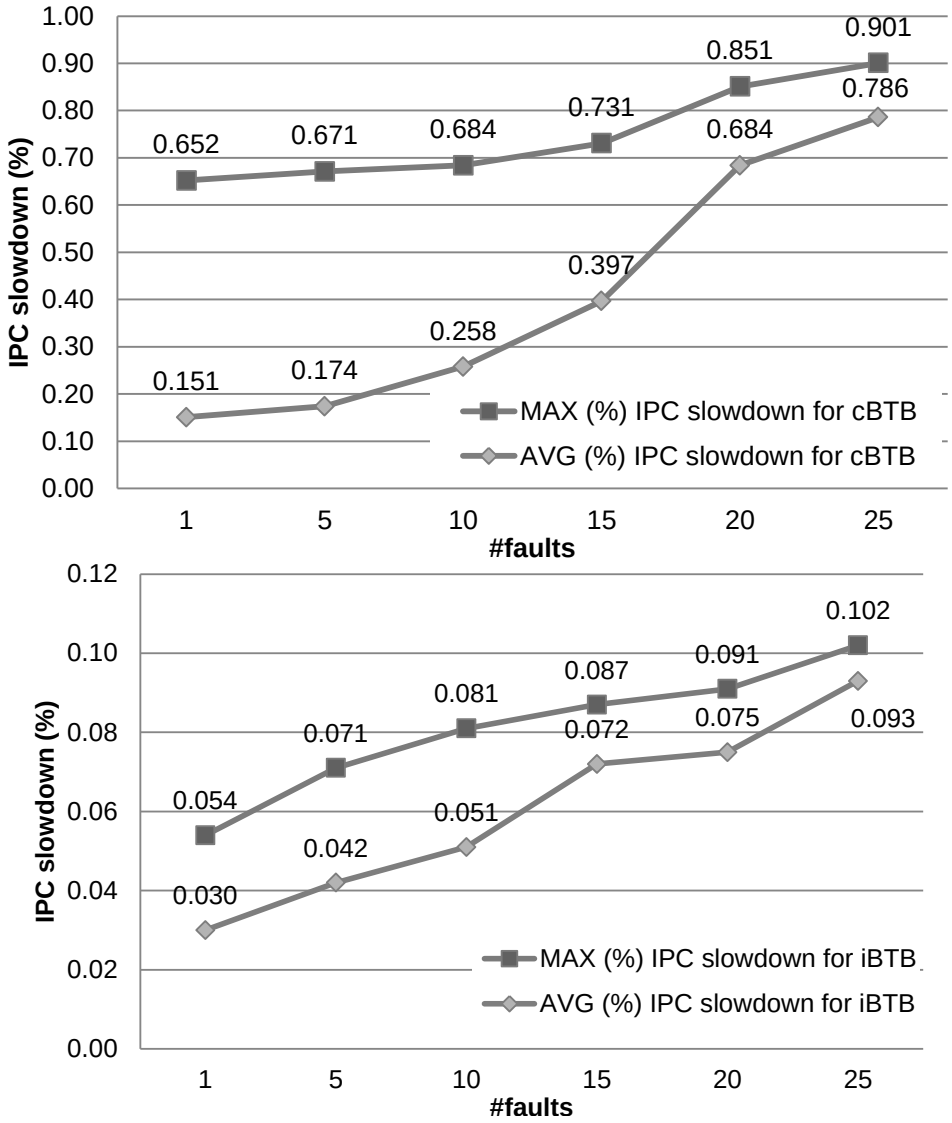


Figure 41: Average and maximum % IPC loss (over the fault-free case) with 1…25 faults per component[3].

[3] The 6.4% IPC slowdown for single fault in the Bimodal predictor has been measured for gcc benchmark which heavily accesses the single faulty entry for the SimPoint sample with the largest weight. Similarly high IPC loss has been observed even for single faults in gobmk and omnetpp.

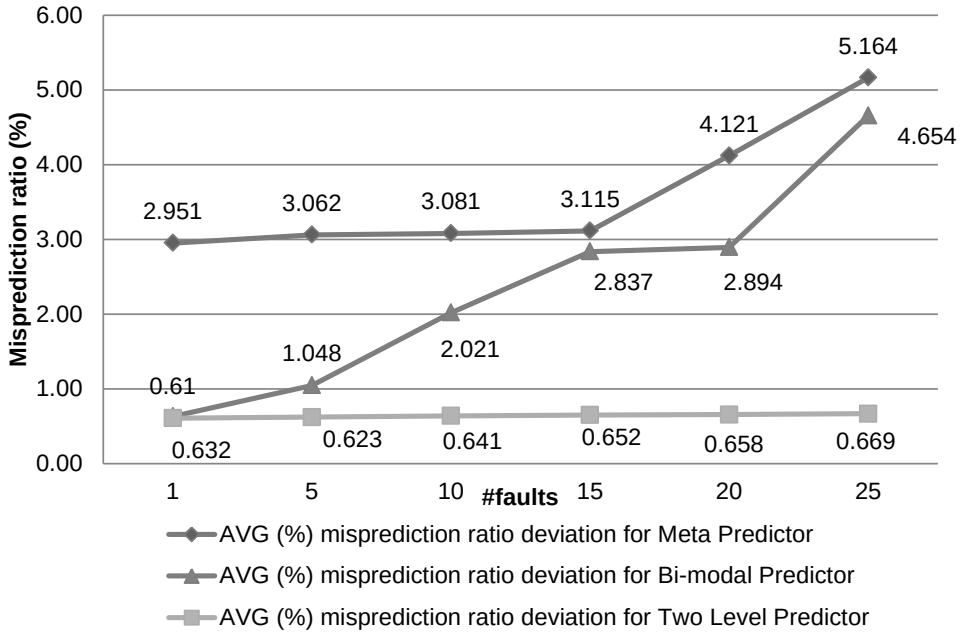


Figure 42: Extra misprediction ratio (%) (on top of the fault-free case) caused by multiple faults in BP.

Table 47: cBTB entry utilization per benchmark.

Benchmark	cBTB % utilization
perlbench	44.82
bzip2	0.53
gcc	2.24
mcf	0.12
gobmk	6.68
dealII	1.17
soplex	9.81
povray	3.36
lbm	0.00
omnetpp	1.39
xalancbmk	17.92
Average	**8.00**

Table 48: Max and average IPC slowdown (%) for single faults in the RAS and the BHR.

Component	Avg IPC slowdown (%)	Max IPC slowdown (%)
RAS	1.03	16.12
BHR	2.26	8.27

4.6.3 Performance variability

Our simulations show *large performance variability* across cores in the presence of faults. Our findings are the following:

- The max IPC slowdown even for *single faults* (Table 48, Figure 41) can be up to 6.45% for the Bimodal, 8.21% for the Meta, and 16.12% for the RAS, there is a large variation of IPC loss. This is observed when all cores are affected by the same number of faults.
- The difference in IPC slowdown for *different number of faults* (1…25 faults; Figure 41) ranges for the Bimodal between 1.3% and 6.5% over the fault-free IPC on the average (6.5% to 11.3% for worst case) and for

the Meta predictor between 5.5% and 7.7% over fault-free IPC on average (8.2% to 13.0% for the worst case). This is observed when the cores are affected by different number of faults.

- The difference between the best and worst case performance for single and multiple faults is not due to outlier behavior. The standard deviation (stdev) of the IPC change for single faults in Bimodal, Meta and RAS is shown in Table 49. Similarly, Table 50 summarizes the stdev of IPC drop for all multiple fault group sizes (5 to 25) in the Bimodal predictor.

Together, the data of Table 49 and Table 50 demonstrate the large variability of the IPC impact for all fault group sizes. In 6 out of the 11 benchmarks the stdev of IPC loss over the fault-free case ranges between 3.5% and 9.6%, and the average stdev across all benchmarks is more than 3% for all fault group sizes.

Table 49: Stdev of IPC change over the fault free case for single fault runs in Bimodal, Meta and RAS.

Stdev of IPC change (%) for single fault runs			
benchmark	**Meta**	**Bimodal**	**RAS**
perlbench	1.648	0.181	1.079
bzip2	8.261	0.643	0.475
gcc	6.510	3.211	1.236
mcf	0.464	0.045	1.014
gobmk	3.457	3.483	0.581
dealII	0.612	0.055	0.516
soplex	0.308	0.662	0.062
povray	4.969	0.464	1.268
lbm	0.050	0.513	0.578
omnetpp	4.184	4.179	2.785
xalancbmk	3.223	3.073	1.885
Average	**3.062**	**1.500**	**1.043**

Table 50: Stdev of IPC drop over the fault free case for multiple faults in the Meta predictor.

	Stdev IPC drop (%) [#faults in Meta]				
benchmark	**[5]**	**[10]**	**[15]**	**[20]**	**[25]**
perlbench	1.843	1.901	2.891	2.755	2.863
bzip2	8.851	8.987	9.644	9.344	9.399
gcc	6.756	6.612	6.891	6.954	6.901
mcf	0.672	0.681	0.788	0.712	0.764
gobmk	3.912	3.935	4.576	4.398	4.411
dealII	0.759	0.814	0.988	0.911	0.932
soplex	0.390	0.400	0.691	0.600	0.609
povray	5.112	5.231	5.958	5.921	5.932
lbm	0.050	0.051	0.077	0.074	0.076
omnetpp	4.352	4.472	4.989	4.981	4.984
xalancbmk	3.499	3.618	4.273	4.247	4.251
Average	**3.290**	**3.336**	**3.796**	**3.717**	**3.738**

4.7 Mechanisms to detect and tolerate hard faults

We propose low-cost hardware-based mechanisms that can detect and diagnose any number of hard faults in front end speculation components and recover the performance loss. The basic detection mechanism is based on enhancing the inherent self-verifying property of the predictors with a write-read-compare (WRC)

flow, while the performance recovery techniques exploit spatial redundancy to isolate, replace, or remap the faulty entries.

4.7.1 Fault detection and diagnosis

All front end predictors generate on-the-fly a response during the self-verification stage. We exploit this property to trigger fault detection and diagnosis. Even though our study focuses on control flow predictors (assuming that no parity or ECC protection exists) the proposed techniques can easily be ported to other self-verified speculation structures such as load address, value, memory dependence predictors, etc.

The top part of Figure 43 shows a traditional self-verifying flow of a speculation component while the bottom part shows the same flow enhanced with the WRC mechanism. During the normal self-verifying flow, the array is updated upon a misprediction (in some cases the predictor is always updated, e.g. in BP). Our enhanced self-verifying flow is activated only on a misprediction and operates as follows:

1. *Write*: This is the normal update step to the predictor.
2. *Read*: The newly written value is read from the prediction array in the next cycle.
3. *Compare*: The newly read data from the prediction array is compared against the ones used to update it (held in a register). In case of mismatch, a permanent (or intermittent) fault is detected (it can't be transient/soft since the entry was just updated). On a match, we assume a fault-free entry.

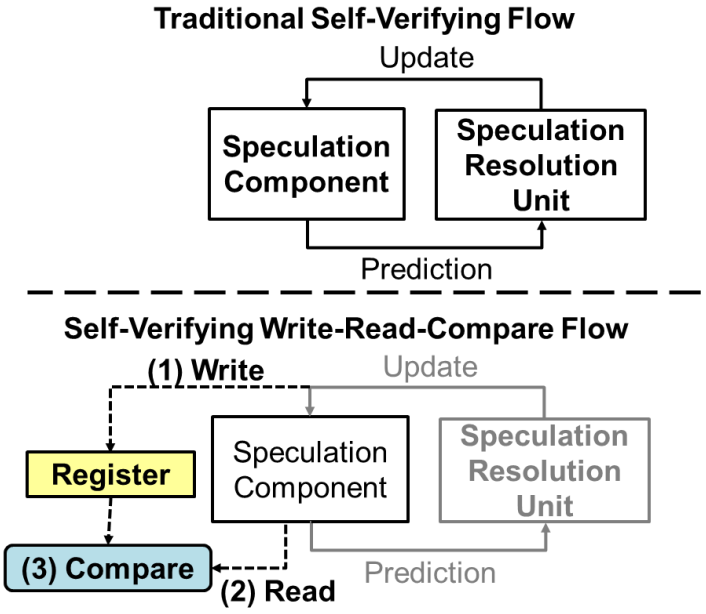


Figure 43: Self-verifying flow enhanced with WRC flow.

The WRC flow requires a comparator to detect the existence of a fault (if the SRAM array supports MBIST, the comparator of MBIST logic can be reused) and extra control logic. Furthermore, array entries are extended with a faulty bit that is set when the entry is diagnosed as faulty. The "faulty" bit is later used by the recovery mechanisms.

The WRC flow will stall fetch during the Read step cycle, only if the front end needs to read the same SRAM bank for a new prediction in that cycle (we model

these stalls assuming an 8KB bank for the BPs and BTBs). We experimentally show that WRC flow overhead is very small.

In Table 51, we show the IPC overhead in fault-free runs when fault detection via the WRC flow is enabled for all predictors. The IPC loss is due to the WRC overhead. Compared to the IPC loss we observe in the runs with multiple faults, the WRC overhead is very small. The WRC overhead can be further minimized if it is enabled only when the CPU core operates in low voltage (i.e. more susceptible to errors). Follows a detailed analysis of the protection mechanisms for each component.

Table 51: IPC (%) loss per benchmark due to fetch stalls in fault-free processor when predictor WRC is enabled.

Benchmark	IPC loss
perlbench	0.164
bzip2	0.209
Gcc	0.819
Mcf	0.327
gobmk	1.227
dealII	0.118
soplex	0.459
povray	0.331
Lbm	0.000
omnetpp	0.097
xalancbmk	0.074
Average	**0.348**

BTB detection/diagnosis: For every cBTB or iBTB read access (target address prediction) the pipeline verifies the predicted target. If the prediction is correct then the BTB entry, used to make the prediction, is fault-free. If the target is mispredicted, then the misprediction happens either due to aliasing or due to a fault. In either case, we trigger the WRC flow and detect if a permanent or intermittent fault has occurred, thus protecting BTB write accesses. Figure 44 illustrates the concept.

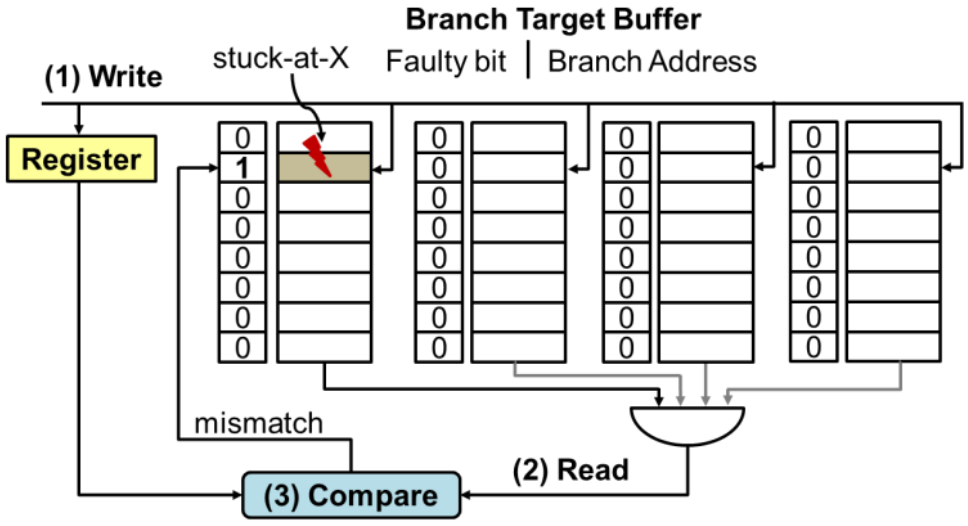


Figure 44: BTB with WRC flow support.

BP detection/diagnosis: A correct conditional branch execution verifies that the 2-bit counter used in the prediction is fault-free. A misprediction could be due to a

fault, aliasing, Meta predictor wrong decision or due to random branch behaviour. Even though, unlike BTBs, BP arrays are always updated, we still trigger a WRC flow only when the predictor mispredicts. The WRC flow for the BP predictors is shown in Figure 45.

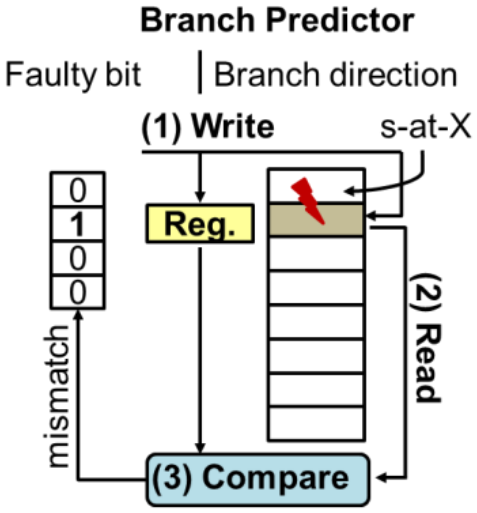


Figure 45: Branch predictor with WRC flow support.

RAS detection/diagnosis: Typically, a RAS is written on every push and read on every pop operation. It is also written when we resolve a misprediction and restore the contents of one of its entries with the correct branch target. Protecting the RAS on read operations can be done when the return instruction is executed using the default verification mechanism. If no misprediction is found then we safely assume that the RAS entry used to make the prediction is fault-free. If a misprediction occurs then we trigger a WRC flow when restoring the RAS entry contents. We also trigger a WRC flow when we push into the RAS after fetching a call instruction. On either occasion, we store the target address to both the RAS and a register. Then, we read the same RAS entry (using a secondary write pointer) and compare the value with the register. If no mismatch is found then there is no fault. If a mismatch is found then a permanent/intermittent fault has been detected (Figure 46). Small structures such as the RAS are typically implemented with flip-flops instead of SRAM but our protection techniques apply to both cases.

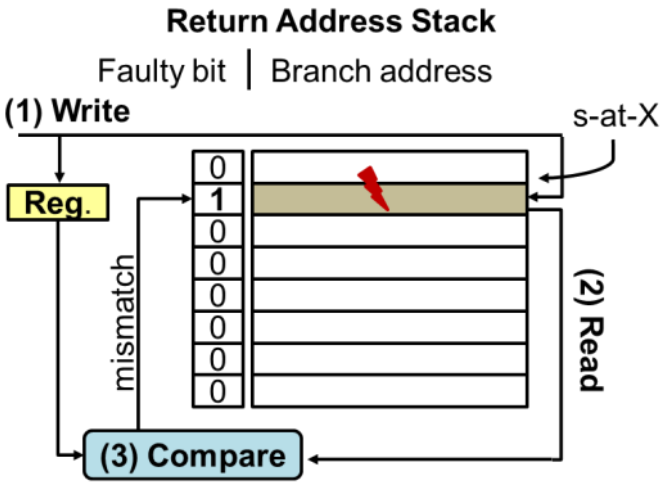


Figure 46: RAS with WRC flow support.

BHR detection/diagnosis: The fault detection mechanism for a N-bit Branch History Register is based on adding a log2(N)-bit up/down, saturating counter, which operates as follows: the counter is incremented every time a '1' is shifted into the BHR and decremented every time a '1' is shifted out. So at any point in time, the counter tracks the number of 1's in the BHR. If the counter underflows, then at least one stuck-at-1 fault has been detected. A stuck-at-1 fault at slot X of an N-bit BHR will propagate to the remaining slots X+1, …, N since the BHR is a shifting structure. That will eventually cause the counter to underflow because we will decrement (shift out 1) more often than we would increment (shift in 1). In the event of constantly shifting 1's (for example when a BHR encounters a stream of taken conditional branch instances inside a loop) then it is possible that no fault will be detected for a period of time. This is not an issue though because the fault is essentially masked. On the contrary, the counter overflows, when the BHR experiences at least one stuck-at-0 fault. A stuck-at-0 fault at slot X will propagate to the remaining slots X+1, …, N and will eventually cause the counter to overflow because it will increment (occasionally shift in 1s) but never decrement (shift out only 0s). Upon a BP misprediction, the BHR contents could be restored to a previous value in order to improve prediction accuracy. In this case, the counter is set to be equal to the number of 1s in the restored BHR value. Figure 47 illustrates the hardware support for detecting hard faults in the BHR. Once a fault is detected, diagnosis of the fault happens without stalling instruction fetch as follows (the predictor keeps using the BHR contents):

- *Stuck-at 1 (0)*: We load BHR with 0s (1s), set the counter to *N* (0) and gradually decrement. This mode ends when the counter underflows (overflows) or when we see the first '1' ('0') at the BHR output before the counter underflows (overflows). If we detect a '1' ('0') then we have verified that the stuck-at-1 (stuck-at-0) fault is hard and the faulty location is at the (*N-M*)*-th slot* of the BHR where *M* is the value of the counter at detection time and *N* is the BHR's length (*M*<*N*). If we don't detect a '1' ('0') at the BHR output by the time the counter underflows (overflows) then no fault is detected and normal operation proceeds by resetting the BHR and the counter.

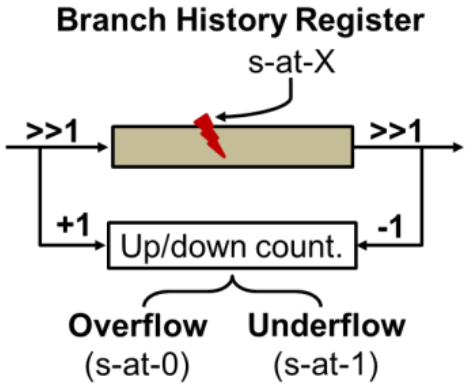


Figure 47: BHR fault detection mechanism.

4.7.2 Performance recovery alternatives

The low-cost fault detection/diagnosis mechanisms discussed in the previous subsection are based on the inherent self-verifying operation of the speculation components and are the major reason towards minimizing performance impact. Once we detect the faulty entries we can minimize the performance impact by avoiding accessing them. Different schemes can be employed towards this goal.

We describe here some indicative mechanisms to recover performance in the BP, RAS and BHR. Similar techniques can be applied to the BTBs but we do not include them for space reasons.

Potential recovery techniques for the BP arrays include:

1. *Single-bit counter*: A faulty 2-bit direction counter (Figure 48) can be converted to a 1-bit counter when a single stuck-at fault is detected (instead of disabling it altogether). We propose adding a small associative table (which we call fault map) that holds the array index of faulty entries, and a 2-bit mask marking the location of the fault. We then use the mask to make a prediction using the fault-free bit or update the counter's fault-free bit. The size of the fault map is dictated by the maximum, estimated number of faults that can occur in a SRAM array under a given technology node and operating voltage.

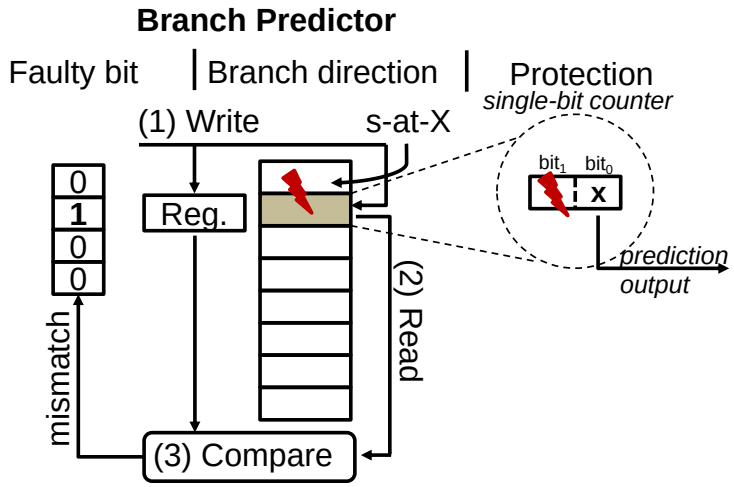


Figure 48: Single-bit counter protection scheme.

2. *Static prediction outcome*: detect faulty counters with a fault map and predict a branch that accesses a faulty counter as always not-taken or always taken (Figure 49). We experimented with a default prediction of not-taken. The results show that performance does not improve and in some cases IPC was further decreased. This happens because the faulty design can provide higher prediction accuracy, since there is a high probability that a fault is masked.

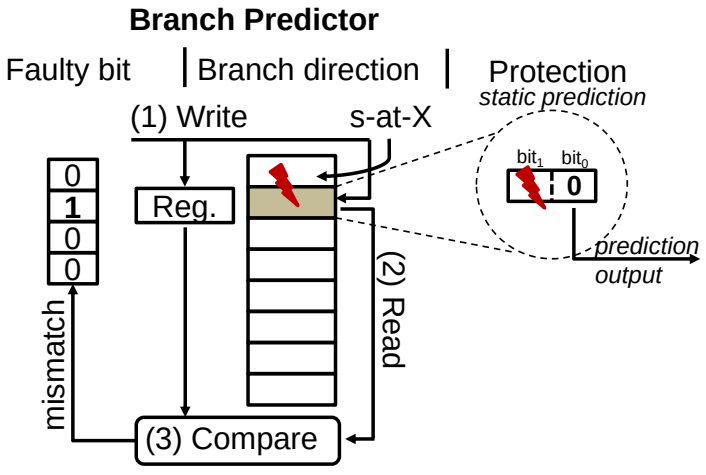


Figure 49: Static prediction outcome protection scheme.

3. *Spare cells*: we re-map faulty entries to fault-free ones (Figure 50). The spare cells can fully recover the performance loss induced from hard faults, as long as the amount of concurrent active hard faults doesn't exceed the amount of the available spares. However, the logic to provide online repair for spare cells lies on the read and write critical paths of the SRAM array and can increase access time [43].

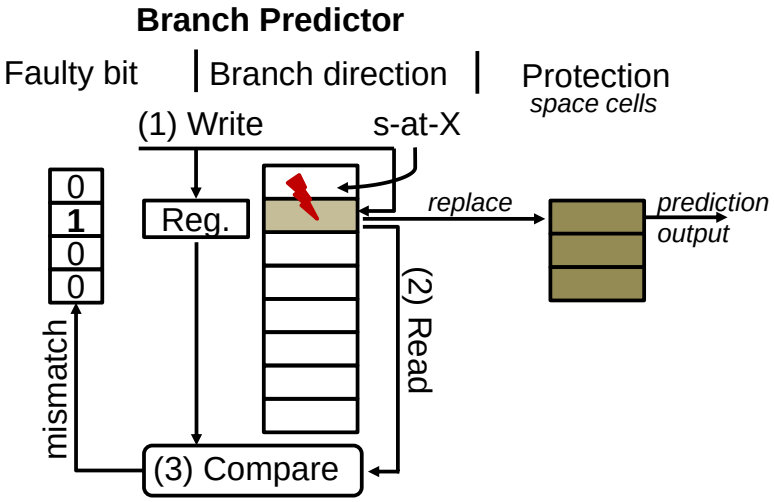


Figure 50: Space cells protection scheme.

A potential recovery mechanism for the RAS would be to disable the faulty entries: the top-of-stack pointer logic arithmetic can be modified to bypass all entries with the faulty bit being set.

Finally, a potential recovery mechanism for the BHR would be to use bypass multiplexers: Once a fault is located, through the up/down saturating counter, the contents of the faulty BHR slot are masked in the index generation logic. Further, a bypass mechanism avoids fault propagation to the rest of the BHR. The BHR slot bypass logic is implemented through the addition of a 2:1 mux between any 2 successive BHR flops. Multiplexer inputs come from the preceding flop of the BHR and the preceding multiplexer to implement the bypass path (Figure 51). The diagnosis mechanism feeds the multiplexers control signals. The faulty BHR slot is bypassed based on the control signals driving the multiplexers that change only when detecting a fault during BHR update.

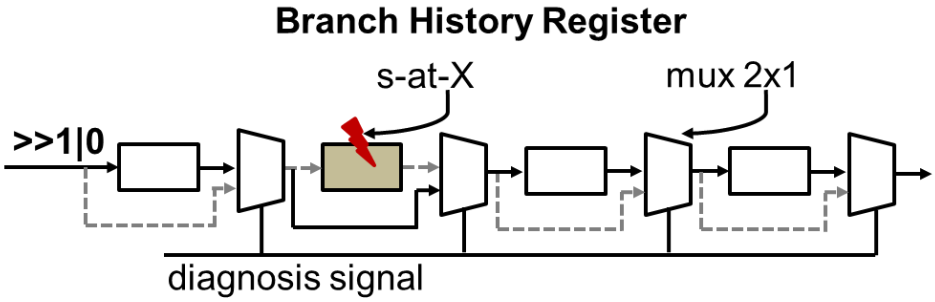


Figure 51: BHR performance recovery mechanism.

4.7.3 Timing implications of the protection mechanisms

There is no timing impact for the WRC flow since we the Read and Compare step are done in a separate cycle from that of the Write step. Our performance recovery solutions for the cBTB/iBTB do not lie on the critical prediction path (they

are triggered on the update flow of the structure) and thus have no effect on the cycle time.

BHR protection can have an impact in the case of multiple, successive, faulty flip-flops but the probability of that happening is extremely small due to the size of the BHR. Alternatively, we can assume that the position and number of multiplexers is such that no cycle violations occur when all bypass paths are activated (e.g. protect every other flip flop in the BHR). The protection mechanism for the RAS can be on the critical path but there is plenty of slack when updating the Read/Write pointers to transition to the next, fault-free RAS entry.

BP fault protection lies on the critical path of making a branch outcome prediction and could put pressure on the clock cycle. In order to alleviate this pressure, control flow can be redirected, one cycle after making a prediction with a faulty counter, once we identify the fault and its position and only if the alternative prediction differs. The redirection leads to a 1 cycle fetch stall.

4.7.4 Existing repair techniques

SRAM arrays can be protected against faults circuit level techniques such as wordline boosting [78] which reduces the effect of process variations and failures; however, such techniques add complexity and area to SRAM array design and only address write failures, which are a subset of all hard faults.

ECC is an architectural-level technique for transient and permanent error detection and correction, typically employed in SRAM arrays which affect functionality. ECC adds area and design overhead proportional to the SRAM array size and number of faults that we want to detect and correct (in the experimental section we compare our solutions to ECC).

Hardware redundancy techniques such as spare SRAM rows/columns have a significant area overhead and can increase access latency [43]. Such techniques won't scale (in terms of circuit complexity and size) in more defective technologies where large numbers of spare rows and columns are required.

Existing SRAM protection/repair techniques are costly and are best used to protect architectural arrays since the cost is justified. For control flow predictors, which do not affect correctness (and thus the cost of expensive techniques is not justified), we propose adopting low cost detection and protection techniques which can recover the performance loss due to the extra mispeculations that hard faults cause.

4.7.5 Comparing with ECC/parity-based protection

SRAM arrays can be protected by parity or ECC. Parity-based protection provides fault detection but no fault diagnosis (e.g., soft vs. hard determination) or performance recovery, masking or mitigation of the fault effect.

ECC provides fault detection and correction but no diagnosis which means that ECC bits can be used to protect against soft errors whose performance impact is negligible. Furthermore, an ECC correction flow lies on the critical path for array reads and will impose a 1-cycle fetch stall overhead just like our proposed solutions for the BP arrays. The ECC generation flow for array writes has the same time overhead per write operation as our WRC flow since it requires a Read-Modify-Write (RMW) flow. The total number of times the ECC RMW flow will be triggered for the BP predictor (and thus the potential for fetch bandwidth loss) will be higher than in our WRC flow solution because ECC is updated for

every write operation, not just in case of a misprediction (as with WRC). Table 52 shows the IPC overhead in fault-free runs of each benchmark when using ECC in the BP arrays.

Table 52: IPC (%) loss per benchmark due to fetch stalls in fault-free processor when predictor ECC is enabled.

benchmark	IPC loss
perlbench	12.806
bzip2	4.357
gcc	10.762
mcf	2.302
gobmk	9.047
dealII	3.483
soplex	3.411
povray	5.462
lbm	0.346
omnetpp	3.671
xalancbmk	12.290
Average	**6.176**

Compared to the WRC IPC loss (Table 51), the ECC IPC loss is almost 6% larger on average (and can be more than 12% in some cases). ECC overhead on writes for the RAS, cBTB and iBTB is similar to the proposed mechanisms. Table 53 compares the area overhead of parity, ECC [69] and our 1-bit counter BP protection solution, assuming 32 bits (16x2-bit counters) per BP entry.

Table 53: BP area overhead of error detection and correction techniques [69].

Protection Technique	Area Overhead
Parity	3.12%
SEC-DED	21.9%
DEC-TED	40.6%
WRC+1-bit (proposed)	0.68%

The extra silicon estate required by our proposed BP error protection solution is a fault map (Figure 45) to identify the faulty entries. We reuse the comparator and register for the WRC flow from the BP Memory BIST logic. We assume a 25-entry fault map to deal with the max number of faults across all 4 technology nodes (Figure 30). Each fault map entry consists of a 16-bit index and 2-bit fault mask for a total of 450 bits of storage (0.68% area overhead) which clearly provides the best trade-off among all solutions.

4.7.6 Performance recovery results

In this section, we provide results for the components that suffer the largest IPC slowdown: Bimodal predictor, Meta predictor, RAS, BHR. We have also implemented and evaluated the protection mechanisms for the Two-level predictor and the two BTBs and results are in line with the ones presented here.

Figure 52 shows the average IPC loss, over all benchmarks, when a varying number of faults are injected in the Bimodal and the Meta predictors. The “unprotected” lines show the IPC loss for the unprotected, faulty core, and the “protected” lines show the IPC loss when the 1-bit counter protection mechanism is used.

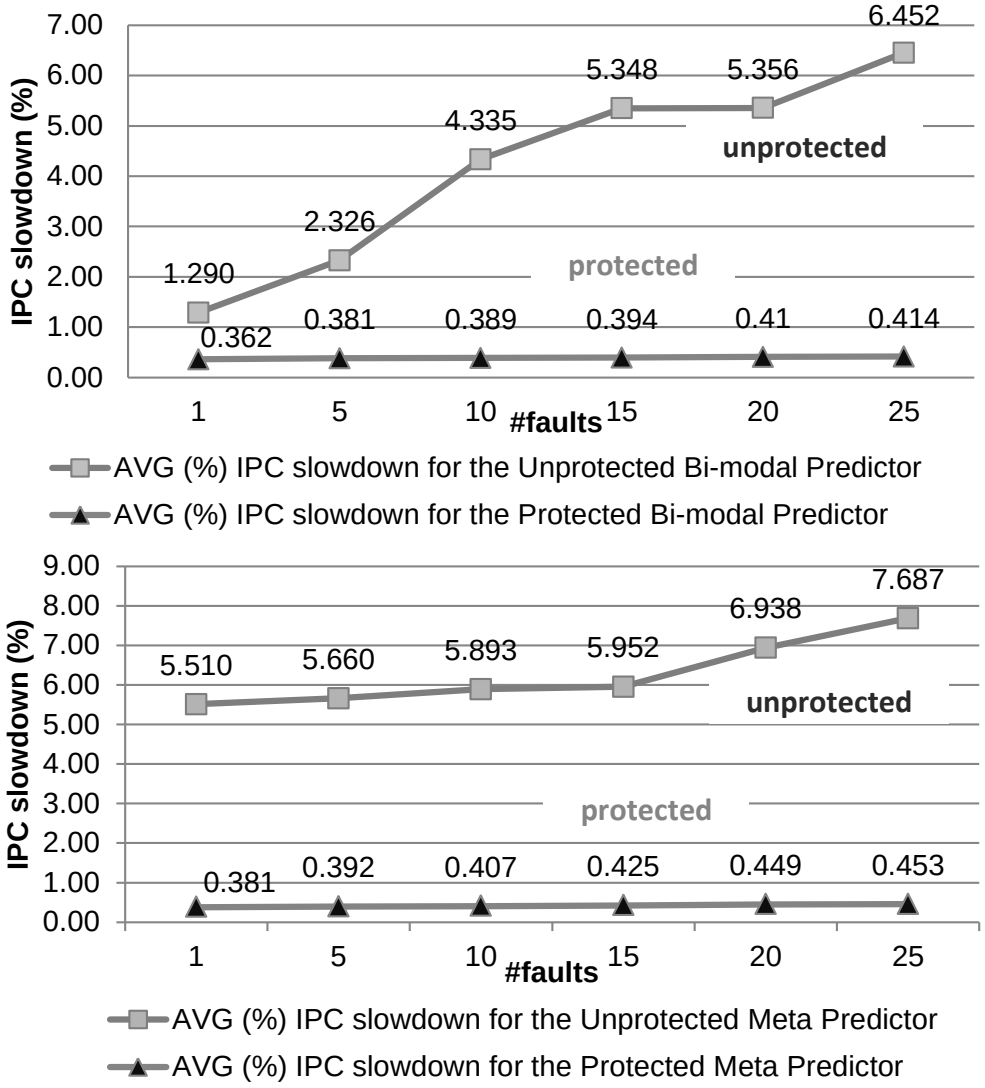


Figure 52: Unprotected vs. protected (1-bit counter) IPC slowdown (%) for Bimodal and Meta predictors.

We recover 94.1% of the IPC loss in the Meta predictor (IPC loss of 0.453% with protection compared to 7.687% IPC loss without protection) and 93.6% of the IPC loss in the Bimodal predictor (IPC loss of 0.414% with protection compared to 6.452% without protection). Our mechanisms also virtually eliminate performance variability due to faults. The unrecovered IPC loss is due to the WRC flow overhead and the use of 1-bit vs. 2-bit counters. The protected cores achieve performance that is almost the same as the fault-free core despite the large number of permanent faults.

Figure 53 shows the performance recovery for single faults in the RAS and BHR; we compare the unprotected and protected structures. More than 94% of IPC loss due to a single fault in the RAS and more than 92% in the BHR are recovered. The remaining IPC loss over the fault-free case (0.057% for RAS, 0.169% for BHR) is due to the reduced sizes of the faulty, protected structures.

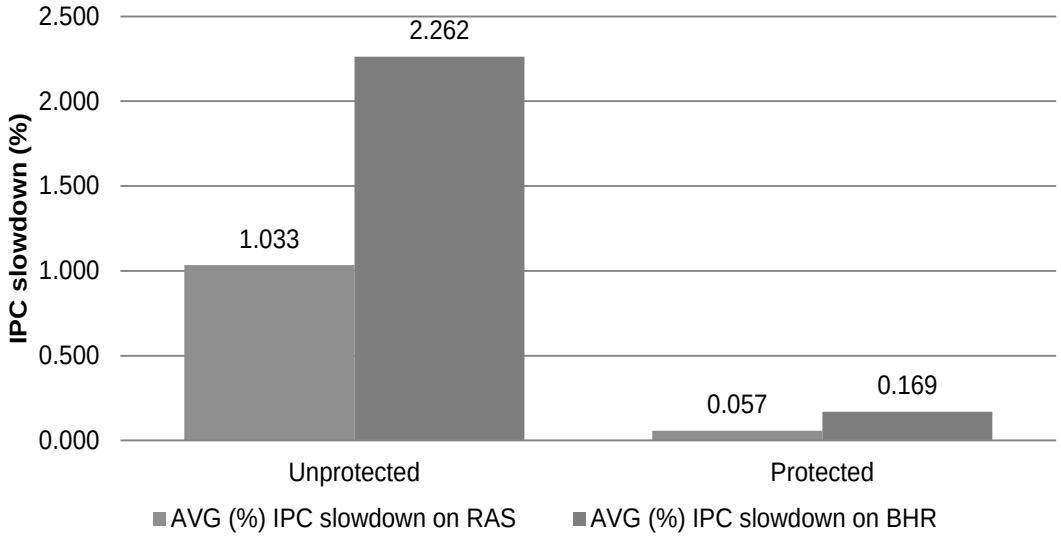


Figure 53: Unprotected vs. protected RAS and BHR average IPC slowdown (%) due to single faults.

4.7.7 Variability recovery and TCO improvement

In this section we discuss the impact on performance variability due to hard faults in front end predictors by measuring the Total Cost of Ownership (TCO). Let us assume a data centre installation with 1000 cores in 16nm. Using the SRAM failure probabilities of Table 1 and an SRAM array of 150K bits (approximately the size of each of the three predictor arrays in our experiments), Figure 54 shows the number of cores that are expected to contain k hard faults in the SRAM array (k is on the x-axis). The number of faults k ranges from 0 to 25 faults.

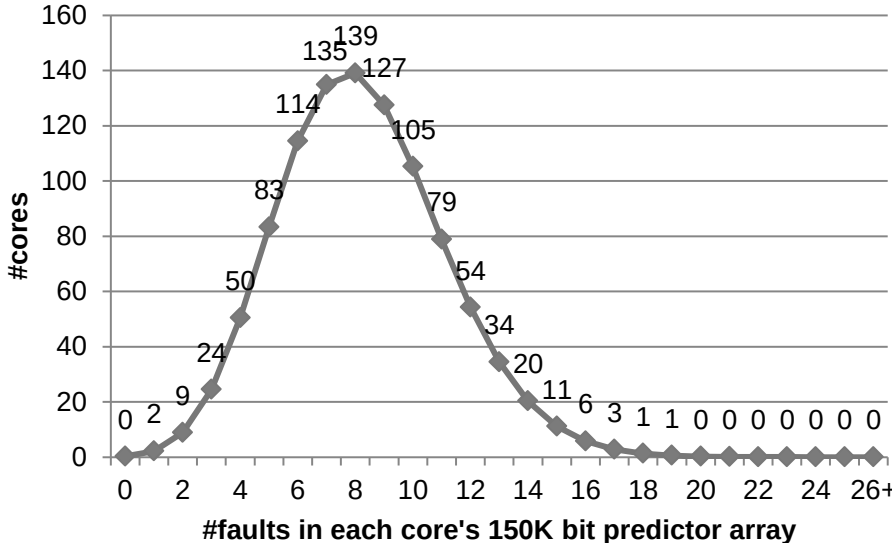


Figure 54: Number of 16nm cores in a 1000-core data centre containing k hard faults in a 150Kbit predictor.

Figure 54 clearly demonstrates that in this 1000-core configuration the numbers of faults in the predictor array varies significantly. For example, 210 of the 1000 cores are expected to contain more than 10 faults, while 11 of the 1000 cores are expected to contain more than 15 faults. Assuming 150Kbits in the Meta predictor array, 210 of the 1000 cores (those with more than 10 hard faults) will suffer (if left unprotected) an average IPC loss of more than 5.893% (max IPC loss 10.231%); as the third diagram of Figure 4 shows. Similarly, 11 of the 1000 cores (those with more than 15 hard faults) will suffer an average IPC loss of more than

5.952% (max IPC loss 11.452%). IPC loss translates to an effective lower clock frequency. For example, a 3.0 GHz core suffering a 6% IPC loss due to faults in a predictor array operates at the same performance level as a 2.82 GHz core due to a 180 MHz frequency degradation.

If the aforementioned IPC loss for the 210 cores, with >10 faults, or the 11 cores, with >15 faults, renders these cores unusable (given an expected minimum performance requirement) then the system must provide spare cores to recover the throughput loss; this increases the overall TCO.

Table 54 shows the TCO increase due to the degraded performance (on average 5.9% and 6.0% IPC loss based on Figure 13; or PVF – performance vulnerability factor – according to [44] [45]) in cores with more than 10 and 15 hard faults on the Meta predictor array. When our protection mechanisms are employed to recover the performance loss (on average the un-recovered IPC slowdown is only 0.4% both for >10 and >15 faults), the TCO of the system remains almost unaffected. We apply the TCO calculations on the original ISPASS paper [45] data centre configuration: the HPE (high performance) server configuration provided in the tool of [111] with 1000 cores, and for three different utilization levels: (a) low (20%), (b) medium (50%) and (c) high utilization (90%).

Table 54: TCO with fault-free, unprotected faulty Meta predictor, and faulty protected cores.

	(TCO=$/Month)		
Average Server Utilization	0.2 (low)	0.5 (medium)	0.9 (high)
Initial TCO	7.848M	8.018M	8.246M
TCO with PVF[4]=5.9%	8.373M (+6.7%)	8.555M (+6.7%)	8.798M (+6.7%)
TCO with PVF=6.0%	8.384M (+6.8%)	8.566M (+6.8%)	8.808M (+6.8%)
TCO with our protection mechanisms (PVF=0.4%)	**7.880M** (+0.4%)	**8.051M** (+0.4%)	**8.297M** (+0.6%)

The TCO increase due to performance loss from a faulty Meta predictor ranges from 6.7% to 6.8% for three different utilization levels of the server farm. When our protection techniques are employed, TCO is marginally affected by only 0.4% to 0.6%.

4.8 Related work

There is no prior work in the literature that: (1) classifies and measures the performance overhead and variability of hard fault in speculative structures; and (2) proposes low-cost fault detection and performance recovery for the front-end predictors.

Speculative arrays: Hardy et al. [44] propose an analytical model called performance vulnerability factor to predict IPC loss due to faults in non-architectural arrays. The method's accuracy depends on having a fixed misprediction penalty in each structure, which is not always true in modern

[4] The PVF (Performance Vulnerability Factor) parameter of the TCO tool corresponds to the performance (IPC) loss.

microarchitectures. Karimi et al. [63] evaluated the impact of faults in branch predictors but the study has limited scope and does not make broad conclusions. Finally, Hsieh et al. [49] discussed the importance of detecting hard faults that only lead to performance degradation to improve microprocessor yield.

A few works address the detection (but no protection) of hard faults in speculative structures. These include Almukhaizim et al. [7], Hatzimihail et al. [46], and Bhattachatya et al. [17] (branch predictor). None of these works evaluates the IPC recovery and cost of the techniques.

Architectural SRAM arrays: Several works examine the impact of and propose remediation techniques for hard faults in caches [5] [8] [66] [89] [100]. Caches affect functionality and their error protection solutions are entirely complementary to our work; we expect cores to need protection in all SRAM arrays, including caches.

Pipeline/Core logic: Using the inherent microarchitectural and/or architectural redundancy to detect and/or repair hard faults has been studied extensively [23] [84] [94] [71] [90] [95] [41] [75]. The focus is to gracefully salvage single or multicore processor chips which contain hard faults in the processor pipeline.

4.9 Findings summary

We presented a detailed classification and quantification of the impact of permanent faults in the performance components of modern microprocessors. The analysis relies on a comprehensive statistical fault injection framework built on top of a cycle-accurate x86-64 based architectural simulator and the latest SPEC CPU2006 CPU benchmarks. Our analysis verifies that performance components with permanent faults do not lead to functional errors and that most faults (44% to 96%) cause only performance fluctuation. Across the different components, performance slowdown ranges from 2% to 20% (faults in control components have more severe performance impacts).

Furthermore, the existence of hard faults in a stride data prefetcher can affect microprocessor performance significantly and increase inter-core performance variability. Our detailed experimental analysis demonstrates that IPC loss due to hard faults in the prefetch table can be up to 17%, and up to 24% and 26% for the prefetch input queue and prefetch request queue, respectively. Also, performance variability across cores is increased: the standard deviation of IPC loss between benchmarks can be more than 4.5%. Similar behaviour was measured in microprocessor front end predictors. The experimental resuts show that: (a) IPC loss due to faults can be more than 16% over the fault-free IPC; and (b) performance variability across faulty cores with the same number of faults each can lead to more than 8% of IPC difference over the fault-free case. The IPC loss stdev for different number of faults per benchmark, ranges between 3.5% and 9.6% for most of the benchmarks, while the stdev of IPC loss across benchmarks is almost 4%.

To tolerate the large performance drop and variability we have proposed a coherent suite of low-cost hardware-based solutions for the detection, diagnosis and performance recovery of multiple hard faults in all front-end speculation components (BPs, BTBs, RAS, BHR). Our evaluation shows that almost the entire IPC loss due to faults is recovered by our solutions. The unrecovered IPC in all cases is within 0.5% of the fault-free IPC while the difference in IPC between cores with significantly different numbers of faults is virtually eliminated: always less than 0.07% of the fault-free IPC. Finally, we estimated the TCO

overhead due to hard faults in one front end predictor, assuming a server farm of 1000 cores and an available TCO estimation model. Our findings show an almost 7% TCO overhead that gets virtually eliminated if our protection mechanisms are used.

5. CONCLUSION AND FUTURE WORK

Today, the pervasiveness of microprocessors, the most complex and immensely powerful product of electronics, in our society goes far beyond the wildest imagination. The same path that is leading technologies toward these remarkable achievements is also making them increasingly unreliable posing a threat to our society. Silicon technology process scaling trends, modern architecture complexity and the compelling requirement to diminish the Time-to-Market threaten to create a "validation wall". As a result, semiconductor industry and academic researchers must explore radical solution and develop innovative techniques to address the dependability challenges of the current and the forthcoming microprocessors. This thesis introduced novel methodologies to address the validation challenges posed throughout different stages of the life-cycle of a microprocessor.

Microprocessor validation is grouped into three categories, based on where they intervene in a microprocessor's lifecycle: (a) *silicon debug*: the first hardware prototypes are exhaustively validated, (b) *manufacturing testing*: the final quality control during massive production, and (c) *in-field verification*: runtime error detection techniques to guarantee correct operation. This thesis introduces various techniques to tackle the challenges of microprocessor validation targeting to: (a) make the microprocessor's verification process more efficient; and (b) be easily applicable to the existing industrial flow. The contributions of this thesis are as follows:

- **Silicon debug**: The share of silicon debug in the overall microprocessor chips development cycle is rapidly expanding due to the ever growing design complexity and the limited throughput of pre-silicon verification methods. Massive application of short random test programs on the prototype microprocessor chips is one of the most effective parts of silicon debug. Despite its bug detection capability, it is constrained by extreme computing needs for random test programs simulation to extract the bug-free memory image. Another major bottleneck and source of "noise" in this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behaviour adds long delays in the debug flow since each failing random test program must be separately examined, although it does not usually bring new debug information. We proposed the employment of self-checking random test programs along with a deconfigurable microprocessor architecture to avoid the time-consuming simulation step, triage the redundant debug sessions and thus accelerate silicon debug. To do so, we exploited the inherent diversity found in all popular Instruction Set Architectures (ISAs) and the ability to deconfigure hardware modules without affecting the functional completeness of a design. Detailed evaluation of the method on an x86 microprocessor model demonstrated its effectiveness in accelerating silicon debug.
- **Manufacturing testing**: We presented an efficient multithreaded (MT) SBST methodology that optimizes self-test time taking maximum advantage of thread-level parallelism while at the same time enhances the self-test program error detection capability on the thread-specific control logic of the processor. The methodology contributed to the effective application of SBST in manufacturing testing. Our experiments on OpenSPARC T1 revealed that the proposed methodology improved

significant test execution time at both the core level (3.6 times) and the processor level (6.0 times) against single-threaded execution, while at the same time it improves fault coverage. Compared with a straightforward multithreading approach, it reduces the self-test time at both the core level and the processor level by 33% and 20%, respectively. Overall, our methodology guarantees high stuck-at fault coverage (88% for the entire processor, more than 1.5M logic gates), which is the highest coverage ever reported in the literature by a software-based functional test methodology in such a complex industrial microprocessor.

- **In-field verification**: Aggressive technology scaling along with low voltage operation exacerbates the likelihood and rate of hard faults not only in large SRAM arrays (such as cache memories), but also in non-SRAM microprocessor structures. Some of the largest non-cache SRAM structures support speculation such as the branch predictor tables, the branch target buffers, and the data prefetcher. Faults in these structures will not affect correctness, but can cause severe performance degradation and variability among otherwise identical cores. We accurately classified and quantified the performance impact of hard faults in non-SRAM structures over a set of CPU benchmarks. To do so, we applied a statistically safe fault injection campaign for single and multiple faults a modified version of the cycle-accurate x86 architectural simulator PTLsim running the SPEC CPU2006 suite. Our evaluation revealed significant differences in the effect of faults and their performance impacts across the components as well as within each component. In particular, we demonstrated that a very large fraction (44% to 96%) of hard faults in these components leads to performance fluctuation, Furthermore, faults in the data prefetcher degrade IPC by up to 26%, compared to fault-free operation, while faults on the branch prediction unit reduce IPC by more than 16%, respectively. Moreover, we found that faults in these components can substantially increase the performance variability across identical cores. Finally, we proposed low-cost microarchitectural techniques to diagnose predictor faults and recover the performance loss. Our techniques exploited the self-verification property of predictors to achieve performance recovery at lower cost than comparable techniques. We found that our solutions can recover almost all performance loss and virtually eliminate performance variability among cores.

The research outcomes of this thesis open the door to several future directions. Future systems architectures must be designed to facilitate hardware validation. In particular, future solutions should have adhere to the following guideline principles: (a) low-power, (b) negligible area overhead, (c) scale with design complexity; and (d) highly automated. In the silicon debug domain, future research should focus on the automation and standardization of the design bug detection and root-cause analysis process. Furthermore, this thesis demonstrated the effectiveness of software-based techniques in accelerating manufacturing testing and guaranteeing a high level of fault coverage. This may be an indication that future microprocessors should devote valuable silicon estate in hardware hooks that enable the at-speed, low-cost testing. The growing demand for high-performance computer systems push computer architects to integrate numerous performance mechanisms in the microprocessor designs. However, functional

correctness is prioritized over performance correctness. This work revealed that faults in performance components can lead to noticeable performance loss and variability in otherwise identical cores. Therefore, future designs must integrate mechanisms to continuously monitor the system performance health and applying contingency actions. Finally, a vital future research direction is to bridge the gap between silicon debug, manufacturing testing and in-field verification techniques through the development of cross-cutting solution that will operate throughout the entire life-cycle of a microprocessor.

The vital challenge of future technologies is to build dependable systems. This thesis proposed various novel techniques to make the validation process, throughout microprocessor life-cycle, more effective in terms of bug/error detection efficiency, resource- and time-budget. We hope, that the contributions presented in this thesis will advance the research in manufacturing dependable microprocessor architectures and will find applicability in future commercial microprocessor products.

ACRONYMS

ATPG	Automatic Test Pattern Generation
BIST	Built-in Self-Test
BTB	Branch Target Buffer
BP	Branch Predictor
cBTB	Conditional/Unconditional direct Branch Target Buffer
CMP	Chip-Multiprocessors
CMT	Chip-Multithreading
DFT	Design for Testability
DLP	Data-Level Parallelism
DP	Data Prefetcher
DPPM	Defective Parts Per Million
ERIT	Equivalent Random Instruction Test
HDL	Hardware Description Language
HVM	High-Volume Manufacturing
iBTB	Unconditional indirect Branch Target Buffer
ICT	Information and Communication Technology
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
IP	Instruction Prefetcher
MHSR	Miss Handling Status Registers
MT-SBST	Multithreading Software-Based Self-Test
OS	Operating System
RAS	Return Address Stack
RIT	Random Instruction Test
RTL	Register-Transfer Level
SBST	Software-based Self-Testing
SEU	Single Event Upset
TLP	Thread-Level Parallelism
TTM	Time-to-Market

REFERENCES

[1] J.Abella, J.Carretero, P.Chaparro, X.Vera, A.Gonzalez, "Low Vccmin Fault-Tolerant Cache with Highly Predictable Performance", In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2009.

[2] M.Abramovici, P.Bradley, K.Dwarakanath, P.Levin, G.Memmi, D.Miller. "A reconfigurable Design-for-Debug Infrastructure for SoCs", In ACM/IEEE Design Automation Conference (DAC), 2006.

[3] A.Adir, E.Almog, L.Fournier, E.Marcus, M.Rimon, M.Vinov and A.Ziv, "Genesys-pro: Innovations in Test Program Generation for Functional Processor Verification", In IEEE Design & Test of Computers (D&T), 21(2):84-93, 2004.

[4] A.Adir, S.Copty, S.Landa, A.Nahir, G.Shurek, A.Ziv, C.Meissner and J.Schumann. A Unified Methodology for Pre-silicon Verification and Post-silicon Validation. In ACM/IEEE Design, Automation & Test in Europe Conference (DATE), 2011.

[5] A.Agarwal, B.C.Paul, H.Mahmoodi, A.Datta, K.Roy, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies", In IEEE Trans. on VLSI Systems (TVLSI), vol. 13, no. 1, pp. 27-38, January 2005.

[6] A.Agarwal, P.Ranganathan, N.Jouppi, J.Smith, "Configurable Isolation: Building high availability systems with commodity multi-core processors", In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2007.

[7] S.Almukhaizim, T.Verdel, Y.Makris, "Cost-effective graceful degradation speculative processor subsystems: the branch prediction case", In IEEE International Conference on Computer Design (ICCD), 2003.

[8] A.Ansari, S.Gupta, S.Feng, S.Mahlke, "Zerehcache: Armoring Cache Architectures in High Defect Density Technologies", In ACM/IEE International Symposium on Microarchitecture (MICRO), 2009.

[9] A.Apostolakis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors", In IEEE Transactions on Computers (TC), vol. 58, no. 12, 2009, pp. 1682-1694.

[10] A.Apostolakis, M.Psarakis, D.Gizopoulos, A.Paschalis, I. Parulkar, "Exploiting Thread-Level Parallelism in Functional Self-Testing of CMT Processors", In IEEE European Test Symposium (ETS), pp. 33-38, 2009.

[11] T.Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", In ACM/IEEE International Symposium on Microarchitecture (MICRO), 1999.

[12] A.Avizienis, H.Yutao. Microprocessor entomology: A Taxonomy of Design Faults in COTS Microprocessors. IEEE Dependable Computing for Critical Applications (DCCA), 1999.

[13] F.Bacchini, R.Damiano, B.Bentley, K.Baty, K.Normoyle, M.Ishii, and E.Yogev. Verification: What Works and What Doesn't. In ACM/IEEE Design Automation Conference (DAC), 2004.

[14] I.Bayraktaroglu, J.Hunt, D.Watkins, "Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues", In IEEE Internatioal Test Conference (ITC), paper 27.2, 2006.

[15] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon and M.Vinov, "Industrial Experience with Test Generation Languages for Processor Verification", In ACM/IEEE Design Automation Conference (DAC), 2004.

[16] B.Bentley, "Validating the Intel® Pentium® 4 Microprocesor", In ACM/IEEE Design Automation Conference (DAC), 2001.

[17] G.Bhattacharya, I.Maity, B.K.Sikdar, B.Das, "Exploring impact of faults on Branch Predictor's Power for Diagnosis of Faulty Module", In IEEE Asian Test Symposium (ATS), 2011.

[18] R.Blish, T.Dellin, S.Huber, M.Johnson, J.Maiz, B.Likins, N.Lycoudes, J.McPherson, Y.Peng, C.Peridier, A.Preussger, G.Prokop, L.Tullos, "Critical Reliability Challenges for The Int'l Technology Roadmap for Semiconductors (ITRS)", Technical Report 03024377A-TR, Int'l SEMATECH, http://www.itrs.net/Links/2003ITRS/LinkedFiles/PIDS/4377atr.pdf, [10/01/2015].

[19] T.Bojan, F.Igor and M.Robert, "Intel's Post Silicon Functional Validation Approach", In IEEE High Level Design Validation and Test Workshop (HLDVT), 2007.

[20] T.Bojan, I.Frumkin and R.Mauri, "Intel® First Ever Converged Core Functional Validation Experience: Methodologies, Challenges, Results and Learning". In IEEE Microprocessor Test and Validation (MTV), 2007.

[21] S.Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation", IEEE Micro, 25(6):10–16, 2005.

[22] F.Bower, D.Sorin and S.Ozev, "A mechanism for online diagnosis of hard faults in microprocessors", In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2005.

[23] F.Bower, P.G.Shealy, S.Ozev, D.J.Sorin, "Tolerating Hard Faults in Microprocessor Array Structures", In IEEE/IFIP International Conference on Dependable systems and Networks (DSN), 2004.

[24] M.Butler, L.Barnes, D.Das Sarma, B.Gelinas, "Bulldozer: An Approach to Multithreaded Compute Performance", In IEEE Micro, Mar/Apr 2011.

[25] J.Carretero, X.Vera, J.Abella, T.Ramirez, M.Monchiero and A.Gonzalez, “Hardware/Software-based diagnosis of load-store queues using expandable activity logs”, In IEEE International Symposium on High Performance Computer Architecture (HPCA), 2011.
[26] L.Chen, A.Avizienis, “N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation”, In IEEE Fault Tolerance Computing Symposium (FTCS), 1996.
[27] L.Chen, S.Ravi, A.Raghunathan, S.Dey, “A Scalable Software-Based Self-Test Methodology for Programmable Processors”, In IEEE/ACM Design Automation Conference (DAC), 2003.
[28] Z.Chishti, A.R.Alameldeen, C.Wilkerson, W.Wu, S.-L.Lu, “Improving Cache Lifetime Reliability at Ultra-low Voltages”, In ACM/IEE International Symposium on Microarchitecture (MICRO), 2009.
[29] K.Constantinides, O.Mutlu, T.Austin. Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation. ACM/IEEE International Symposium on Microarchitecture (MICRO), 2008.
[30] F.Corno, E.Sanchez, M.Sonza Reorda, G.Squillero, “Automatic Test Program Generation – a Case Study”, In IEEE Design & Test of Computers (D&T), vol. 21, no. 2, pp. 102–109, 2004.
[31] A.Danowitz, K.Kelley, J.Mao, J.P.Stevenson, M.Horowitz, CPU DB: Recording Microprocessor History, ACM Queue Magazine, vol. 10, no. 4, April 2002.
[32] E.Daoud and N.Nicolici, “Embedded Debug Architecture for Bypassing Blocking Bugs During Post-silicon Validation”, In IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 19(4):559-570, 2011.
[33] Datasheet Intel 4004; http://datasheets.chipdb.org/Intel/MCS-4/datashts/intel-4004.pdf [Accessed 26/11/2014].
[34] N.DeBardeleben, “Reliability requirements for GPUs in HPC”, Tutorial on Soft Error Resilience in GPGPUs for HPC, HiPEAC Conference, 2014.
[35] R.H.Dennard, F.H.Gaenssien, H-N.Yu, V.L.Rideout, E.Bassous, A.LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions”, In IEEE Journal of Solid State Circuit, October, 1974.
[36] J.S.Gardner, “Octeon III starts at Low End”, In Microprocessor Report, Jun 2013.
[37] D.Gizopoulos, Advance in Electronic Testing: Challenges and Methodologies, Springer, 2006.
[38] D.Gizopoulos, M.Psarakis, M.Hatzimihail, M.Maniatakos, A.Paschalis, A.Raghunathan, S.Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, In IEEE Transactions on VLSI Systems (TVLSI), vol.16, no. 11, pp 1441-1453, November 2008.
[39] D.Gizopoulos, S.Mukherjee, “Special Section on Dependable Computer Architecture: Guest Editors’Introduction”, In IEEE Transactions on Computers (TC), vol. 60, no. 1, January 2011.
[40] J.Goodenough and R.Aitken, “Post-silicon is Too Late Avoiding the $50 Million Paperweight Starts with Validated Designs’, In ACM/IEEE Design Automation Conference (DAC), 2010.
[41] S.Gupta, S.Feng, A.Ansari, J.Blome, S.Mahlke, “The StageNet Fabric for Constructing Resilient Multicore Systems”, In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2008.
[42] S.Gurumurthy, S.Vasudevan and J.Abraham, “Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor”, In IEEE International Test Conference (ITC), 2006.
[43] T.P.Haraszti, CMOS Memory Circuits, Kluwer, 2000.
[44] D.Hardy, I.Sideris, N.Ladas, Y.Sazeides, “The performance vulnerability of architectural and non-architectural arrays to performance faults”, In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2012.
[45] D.Hardy, M.Kleanthous, I.Sideris, A.Saidi, E.Ozer, Y.Sazeides, “An Analytical Framework for Estimating TCO and Exploring Data Center Design Space”, In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013.
[46] M.Hatzimihail, M.Psarakis, D.Gizopoulos, A.Paschalis, “A Methodology for Detecting Performance Faults in Microprocessor Speculative Execution Units via Hardware Performance Monitoring”, In IEEE International Test Conference (ITC), paper 29.3, 2007.
[47] J.Hennessy, D.Patterson, Computer Architecture: A Quantitative Approach (5th Edition), Elsevier, 2012.
[48] T.Hong, Y.Li, S-B.Park, D.Mui, D.Lin, Z.A.Kaleq, N.Hakim, H.Naeimi, D.S.Gardner and S.Mitra, “QED: Quick Error Detection Tests for Effective Post-silicon Validation”, In IEEE International Test Conference (ITC), 2010.
[49] T.Y.Hsieh, M.A.Breuer, M.Annavaram, S.K.Gupta, K.-J.Lee, “Tolerance of Performance Degrading Faults for Effective Yield Improvement”, In IEEE International Test Conference (ITC), 2009.
[50] Y.C.Hsu, F.Tsai, W.Jong and Y-T Chang, “Visibility Enhancement for Silicon Debug”, In ACM/IEEE Design Automation Conference (DAC), 2006.
[51] C.Hughes, P.Kaul, S.V.Adve, R.Jain, C.Park, J.Srinivasan, “Variability in the Execution of Multimedia Applications and Implications for Architecture”, In ACM/IEEE International Symposium on Compute Architecture (ISCA), 2001.

[52] Inquirer staff, Intel hidden Xeon, Pentium 4 bugs, http://www.theinquirer.net, [Accessed 04/01/2015].
[53] Inquirer Staff. AMD Opteron bug can cause incorrect results. http://www.theinquirer.net, [Accessed 15/12/2014].
[54] Inquirer Staff. IBM Power PC 1GHz chip only runs properly at 933MHz. http://www.theinquirer.net, [Accessed 15/12/2014].
[55] Intel CEO promises Broadwell PCs on shelves for holidays; http://www.reuters.com/article/2014/05/18/us-intel-chips-idUSBREA4H08P20140518 [Accessed 30/11/2014].
[56] Intel finds flaw in Sandy Bridge chipsets, halt shipment. http://www.techreport.com/discussions.x/20326 [Accessed 1/12/2014].
[57] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation, Nov. 2009.
[58] Intel® 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, Jun.2010.
[59] JEDEC Solid State Technology Association. Failure Mechanisms and Models for Semiconductor Devices. JEDEC Publication JEP122-G, http://www.jedec.org/standards-documents/docs/jep-122e [Accessed 10/01/2015].
[60] D.Josephson, “The Manic Depression of Microprocessor Debug”, In IEEE International Test Conference (ITC), 2002.
[61] D.Josephson, S.Poehhnan and V.Govan, “Debug Methodology for McKinley Processor”, In IEEE International Test Conference (ITC), 2001.
[62] D.Josephson. The Good, the Bad, and the Ugly of Silicon Debug. In ACM/IEEE Design Automation Conference (DAC), 2006.
[63] N.Karimi, M.Maniatakos, C.Tirumurti, A.Jas, Y.Makris, “Impact Analysis of Performance Faults in Modern Microprocessors”, In IEEE International Conference on Computer Design (ICCD), 2009.
[64] J.Keshava, N.Hakim and C.Prudvi, “Post-silicon Validation Challenges: How EDA and Academia Can Help”, In ACM/IEEE Design Automation Conference (DAC), 2010.
[65] A.Krstic, W.C.Lai, K.T.Cheng, L.Chen, S.Dey, “Embedded Software-Based Self-Test for Programmable Core-Based Designs”, In IEEE Design and Test of Computers (D&T), vol. 19, no. 4, pp. 18-27, July-August 2002.
[66] H.Lee, S.Cho, B.R.Childers, “Performance of Graceful Degradation for Cache Faults”, IEEE Comp.Society Symp. on VLSI (ISVLSI), 2007.
[67] R.Leveugle, A.Calvez, P.Maistri, P.Vanhauwaert, “Statistical Fault Injection: Quantified Error and Confidence,” In IEEE Design, Automation & Test In Europe (DATE), 2009.
[68] L.Lingappan, N.K.Jha, “Satisfiability-based automatic test program generation and design for testability for microprocessors”, In IEEE Transactions on VLSI Systems (TVLSI), vol. 15, no. 5, pp. 518-530, May 2007.
[69] Y.Luo, S.Govindan, B.Sharma, M.Santaniello, J.Meza, A.Kansal, J.Liu, B.Khessib, K.Vaid, O.Mutlu, “Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory”, In IEEE/IFIP International Conference on Dependable systems and Networks (DSN), 2014.
[70] R.McLaughlin, S.Venkataraman and C.Lim, “Automated Debug of Speed Path Failures Using Functional Tests”, In IEEE VLSI Test Symposium (VTS), 2009.
[71] A.Meixner and D.Sorin, “Detouring: Translating Software to Circumvent Hard Faults in Simple Cores”, In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2008.
[72] S.Mitra, S.A.Seshia and N.Nicolici, “Post-silicon Validation Opportunities, Challenges and Recent Advances”, In ACM/IEEE Design Automation Conference (DAC), 2010.
[73] G.Moore, “Cramming more components onto integrated circuits”, In Electronics, April, 1965.
[74] S.R.Nassif, N.Mehta, Y.Cao, "A Resilience Roadmap", In IEEE Design, Automation & Test In Europe (DATE), 2010.
[75] S.Nomura, M.D.Sinclair, C.-H.Ho, V.Govindaraju, M.de Kruijf, K.Sankaralingam, “Sampling+DMR: Practical and Low-Overhead Permanent Fault Detection”, In ACM/IEEE International Symposium on Compute Architecture (ISCA), 2011.
[76] N.Oh, S.Mitra and E.J.McCluskey, “ED4I: Error Detection by Diverse Data and Duplicated Instructions”, In IEEE Transactions on Computers (TC), 51(2):180-199, 2002.
[77] OpenSPARC T1 Microarchitecture Specification, Sun Microsystems Inc., Aug. 2006.
[78] Y.Pan, J.Kong, S.Ozdemir, G.Memik, S.W.Chung, “Selecting Wordline Voltage Boosting for Caches to manage Yield under Process Variations”, In ACM/IEEE Design Automation Conference (DAC), 2009.
[79] S.B.Park, T.Hong and S.Mitra, “Post-silicon Bug Localization in Processors Using Instruction Footprint Recording and Analysis (IFRA)”, In IEEE Transactions on Computer-Aided Design (TCAD), 28(10):1545-1558, 2009.
[80] P.Parvathala, L.Maneparambil, W.Lidsay, “FRITS – A Microprocessor Functional BIST Method”, In IEEE International Test Conference (ITC), 2002.

[81] A.Paschalis, D.Gizopoulos, “Effective Software-Based Self-Test Strategies for On-line Periodic Testing of Embedded Processors”, In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 24, no. 1, 2005, pp. 88-99.
[82] P.Patra, “On the Cusp of a Validation Wall”, In IEEE Design & Test (D&T), vol.24, no.2, pp.193-196, March-April 2007.
[83] Pentium FDIV bug. http://www.intel.com/support/processors/pentium/sb/CS-012748.htm, [Accessed 15/12/2014].
[84] M.D.Powell, A.Biswas, S.Gupta, S.S.Mukherjee, “Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance”, In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2009.
[85] M.Psarakis, D.Gizopulos, E.Sanchez, M.Sonza Reorda, “Microprocessor Software-Based Self-Testing”, In IEEE Design & Test of Computers (D&T), May/June 2010.
[86] J.M.Rabaey, Digital Integrated Circuits: a design perspective, Prentice-Hall, Inc., 1996.
[87] R.Raina and R.Molyneaux, “Random Self-test Method –Applications on PowerPCTM Microprocessor Caches”, In IEEE Great Lake Symposium on VLSI, 1998.
[88] G.Reis, J.Chang, N.Vachharajani, R.Rangan and D.August, “SWIFT: Software Implemented Fault Tolerance”, In ACM/IEEE International Symposium on Code Generation and Optimization (CGO), 2005.
[89] D.Roberts, S.K.Nam, T.Mudge, “On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology”, In Microprocessors and Microsystems, vol. 32, no. 5-6, pp. 244-253, Aug. 2008.
[90] B.Romanescu, D.Sorin, “Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults”, In IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008.
[91] H.Rotithor, “Postsilicon Validation Methodology for Microprocessors”, In IEEE Design & Test of Computers (D&T), 17(4):77-88, 2000.
[92] J.Rupley, “AMD’s ‘Jaguar’: A next generation low power x86 core,” Hot Chips 24, Aug, 2012.
[93] S.Sarangi, A.Tiwari, J.Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. ACM/IEEE International Symposium on Microarchitecture (MICRO), 2006.
[94] E.Schuchman, T.N.Vijaykumar, "Rescue: a microarchitecture for testability and defect tolerance", In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2005.
[95] E.Schuchman, T.N.Vijaykumar, “BlackJack: Hard Error Detection with Redundant Threads on SMT”, In IEEE/IFIP International Conference on Dependable systems and Networks (DSN), 2007.
[96] J.Shen, J.A.Abraham, “Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation”, In IEEE International Test Conference (ITC), 1998.
[97] P.Shivakumar, S.Keckler, C.Moore, D.Burger, “Exploiting microarchitectural redundancy for defect tolerance”, In IEEE International Conference on Computer Design (ICCD), 2003.
[98] C.Shum, F. Busaba, S.Dao-Trong, G.Gerwig, C.Jacobi, T.Koehler, E.Pfeffer, B.R.Prasky, J.G.Rell, A.Tsai, “Design and microarchitecture of the IBM System z10 microprocessor”, In IBM Journal of Research and Development, v53,n1, 2009.
[99] E.Singerman, Y.Abarbanel and S.Baartmans, “Transaction Based Pre-to-Post Silicon Validation”, In ACM/IEEE Design Automation Conference (DAC), 2011.
[100] G.Sohi, “Cache Memory Organization to Enhance the Yield of High Performance VLSI Processors”, In IEEE Transactions on Computers (TC), vol. 38, no. 4, pp.484-492, April 1989.
[101] J.Srinivasan, S.Adve, P.Bose, J.Rivers, “Exploiting structural duplication for lifetime reliability enhancement”, In ACM/IEEE International Symposium on Compute Architecture (ISCA), 2005.
[102] S.Sudhakrishnan, L.Su and J.Renau, “Processor Verification with hwBugHunt”, In IEEE International Symposium on Quality Electronic Design (ISQED), 2008.
[103] S.Sudhakrishnan, R.Dicochea and J.Renau, “Releasing Efficient Beta Cores to Market Early”, In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2011.
[104] P.J.Tan, T.Le, K.-H.Ng, P.Mantri, J.Westfall, “Testing of UltraSPARC T1 Microprocessor and its Challenges”, In IEEE International Test Conference (ITC), paper 16.1, 2006.
[105] I.Wagner and V.Bertacco, “Reversi: Post-silicon Validation System for Modern Microprocessors”, In IEEE International Conference on Computer Design (ICCD), 2008.
[106] I.Wagner, V.Bertacco and T.Austin, “Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors”, In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 27(2):380-393, 2008.
[107] L.T.Wang, C.Stroud, N.Touba, “System-on-Chip Test Architectures: Nanometer design for testability”, Elsevier, 2007.

[108] C.Wilkerson, H.Gao, A.R.Alameldeen, Z.Chishti, M.Khellah, S.-L.Lu, “Trading off Cache Capacity for Reliability to Enable Low Voltage Operation”, In ACM/IEEE International Symposium on Compute Architecture (ISCA), 2008.

[109] G.Xenoulis, D.Gizopoulos, M.Psarakis, A.Paschalis, “Instruction-Based Online Periodic Self-Testing of Microprocessors with Floating-Point Units”, In IEEE Transactions on Dependable and Secure Computing, vol. 6, no.2, pp. 124-134, 2009.

[110] M.Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”, In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2007.

[111] COST-ET: COSTing and Exploring TCO for data centers, http://www.cs.ucy.ac.cy/carch/xi/cost-et.php. [Accessed 12/01/2015]