NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Methodologies for Accelerated Analysis of the Reliability and the Energy Efficiency Levels of Modern Microprocessor Architectures

Emmanouil E. Kaliorakis

ATHENS

MARCH 2018

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μεθοδολογίες για την Επιτάχυνση της Ανάλυσης της Αξιοπιστίας και της Ενέργειας Σύγχρονων Αρχιτεκτονικών Μικροεπεξεργαστών

Εμμανουήλ Ε. Καληωράκης

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2018

PhD THESIS

Methodologies for Accelerated Analysis of the Reliability and the Energy Efficiency Levels of Modern Microprocessor Architectures

Emmanouil E. Kaliorakis

ADVISOR: Dimitris Gizopoulos, Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:
Dimitris Gizopoulos, Professor UoA
Antonis Paschalis, Professor UoA
Angeliki Arapoyanni, Professor UoA

SEVEN-MEMBER EXAMINATION COMMITTEE

(Signature)	(Signature)
Dimitris Gizopoulos, Professor UoA	**Antonis Paschalis, Professor UoA**
(Signature)	(Signature)
Ioannis Smaragdakis, Professor UoA	**Dionisios Pnevmatikatos, Professor TUC**
(Signature)	(Signature)
Mihalis Psarakis, Assistant Professor UniPi	**Nectarios Koziris, Professor NTUA**
(Signature)	
Dimitrios Soudris, Associate Professor NTUA	

Examination Date 02/03/2018

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μεθοδολογίες για την Επιτάχυνση της Ανάλυσης της Αξιοπιστίας και της Ενέργειας Σύγχρονων Αρχιτεκτονικών Μικροεπεξεργαστών

Εμμανουήλ Ε. Καληωράκης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Δημήτρης Γκιζόπουλος, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:
Δημήτρης Γκιζόπουλος, Καθηγητής ΕΚΠΑ
Αντώνης Πασχάλης, Καθηγητής ΕΚΠΑ
Αγγελική Αραπογιάννη, Καθηγήτρια ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

(Υπογραφή)	(Υπογραφή)
Δημήτρης Γκιζόπουλος, Καθηγητής ΕΚΠΑ	**Αντώνης Πασχάλης, Καθηγητής ΕΚΠΑ**
(Υπογραφή)	(Υπογραφή)
Ιωάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ	**Διονύσιος Πνευματικάτος, Καθηγητής Πολυτεχνείο Κρήτης**
(Υπογραφή)	(Υπογραφή)
Μιχάλης Ψαράκης, Επίκουρος Καθηγητής Πα.Πει	**Νεκτάριος Κοζύρης, Καθηγητής ΕΜΠ**
(Υπογραφή)	
Δημήτριος Σούντρης, Αναπληρωτής Καθηγητής ΕΜΠ	

Ημερομηνία εξέτασης 02/03/2018

ABSTRACT

The evolution in semiconductor manufacturing technology, computer architecture and design leads to increase in performance of modern microprocessors, which is also accompanied by increase in products' vulnerability to errors. Designers apply different techniques throughout microprocessors life-time in order to ensure the high reliability requirements of the delivered products that are defined as their ability to avoid service failures that are more frequent and more severe than is acceptable.

This thesis proposes novel methods to guarantee the high reliability and energy efficiency requirements of modern microprocessors that can be applied during the early design phase, the manufacturing phase or after the chips release to the market. The contributions of this thesis can be grouped in the two following categories according to the phase of the CPUs lifecycle that are applied at:

- **Early design phase**: Statistical fault injection using microarchitectural structures modeled in performance simulators is a state-of-the-art method to accurately measure the reliability, but suffers from low simulation throughput. In this thesis, we firstly present a novel fully-automated versatile microarchitecture-level fault injection framework (called MaFIN) for accurate characterization of a wide range of hardware components of an x86-64 microarchitecture with respect to various fault models (transient, intermittent, permanent faults). Next, using the same tool and focusing on transient faults, we present several reliability and performance related studies that can assist design decision in the early design phases.

  Moreover, we propose two methodologies to accelerate the statistical fault injection campaigns. In the first one, we accelerate the fault injection campaigns after the actual injection of the faults in the simulated hardware structures. In the second, we further accelerate the microarchitecture level fault injection campaigns by proposing MeRLiN a fault pre-processing methodology that is based on the pruning of the initial fault list by grouping the faults in equivalent classes according to the instruction access patterns to hardware entries.

- **Manufacturing phase and release to the market**: The contributions of this thesis in these phases of microprocessors life-cycle cover two important aspects. Firstly, using the 48-core Intel's SCC architecture, we propose a technique to accelerate online error detection of permanent faults for many-core architectures by exploiting their high-speed message passing on-chip network. Secondly, we propose a comprehensive statistical analysis methodology to accurately predict at the system level the safe voltage operation margins of the ARMv8 cores of the X-Gene 2 chip when it operates in scaled voltage conditions.

SUBJECT AREA: Computer Architecture

KEYWORDS: Dependability, Reliability, Transient Faults, Permanent Faults, Energy Efficiency, Statistical Analysis

ΠΕΡΙΛΗΨΗ

Η εξέλιξη της τεχνολογίας ημιαγωγών, της αρχιτεκτονικής υπολογιστών και της σχεδίασης οδηγεί σε αύξηση της απόδοσης των σύγχρονων μικροεπεξεργαστών, η οποία επίσης συνοδεύεται από αύξηση της ευπάθειας των προϊόντων. Οι σχεδιαστές εφαρμόζουν διάφορες τεχνικές κατά τη διάρκεια της ζωής των ολοκληρωμένων κυκλωμάτων με σκοπό να διασφαλίσουν τα υψηλά επίπεδα αξιοπιστίας των παραγόμενων προϊόντων και να τα προστατέψουν από διάφορες κατηγορίες σφαλμάτων διασφαλίζοντας την ορθή λειτουργία τους.

Αυτή η διδακτορική διατριβή προτείνει καινούριες μεθόδους για να διασφαλίσει τα υψηλά επίπεδα αξιοπιστίας και ενεργειακής απόδοσης των σύγχρονων μικροεπεξεργαστών οι οποίες μπορούν να εφαρμοστούν κατά τη διάρκεια του πρώιμου σχεδιαστικού σταδίου, του σταδίου παραγωγής ή του σταδίου της κυκλοφορίας των ολοκληρωμένων κυκλωμάτων στην αγορά. Οι συνεισφορές αυτής της διατριβής μπορούν να ομαδοποιηθούν στις ακόλουθες δύο κατηγορίες σύμφωνα με το στάδιο της ζωής των μικροεπεξεργαστών στο οποίο εφαρμόζονται:

- **Πρώιμο σχεδιαστικό στάδιο**: Η στατιστική εισαγωγή σφαλμάτων σε δομές που είναι μοντελοποιημένες σε προσομοιωτές οι οποίοι στοχεύουν στην μελέτη της απόδοσης είναι μια επιστημονικά καθιερωμένη μέθοδος για την ακριβή μέτρηση της αξιοπιστίας, αλλά υστερεί στον αργό χρόνο εκτέλεσης. Σε αυτή τη διατριβή, αρχικά παρουσιάζουμε ένα νέο πλήρως αυτοματοποιημένο εργαλείο εισαγωγής σφαλμάτων σε μικροαρχιτεκτονικό επίπεδο που στοχεύει στην ακριβή αξιολόγηση της αξιοπιστίας ενός μεγάλου πλήθους μονάδων υλικού σε σχέση με διάφορα μοντέλα σφαλμάτων (παροδικά, διακοπτόμενα, μόνιμα σφάλματα). Στη συνέχεια, χρησιμοποιώντας το ίδιο εργαλείο και στοχεύοντας τα παροδικά σφάλματα, παρουσιάζουμε διάφορες μελέτες σχετιζόμενες με την αξιοπιστία και την απόδοση, οι οποίες μπορούν να βοηθήσουν τις σχεδιαστικές αποφάσεις στα πρώιμα στάδια της ζωής των επεξεργαστών.

  Τελικά, προτείνουμε δύο μεθοδολογίες για να επιταχύνουμε τα μαζικά πειράματα στατιστικής εισαγωγής σφαλμάτων. Στην πρώτη, επιταχύνουμε τα πειράματα έπειτα από την πραγματική εισαγωγή των σφαλμάτων στις δομές του υλικού. Στη δεύτερη, επιταχύνουμε ακόμη περισσότερο τα πειράματα προτείνοντας τη μεθοδολογία με όνομα MeRLiN, η οποία βασίζεται στη μείωση της αρχικής λίστας σφαλμάτων μέσω της ομαδοποίησής τους σε ισοδύναμες ομάδες έπειτα από κατηγοριοποίηση σύμφωνα με την εντολή που τελικά προσπελαύνει τη δομή που φέρει το σφάλμα.

- **Παραγωγικό στάδιο και στάδιο κυκλοφορίας στην αγορά**: Οι συνεισφορές αυτής της διδακτορικής διατριβής σε αυτά τα στάδια της ζωής των μικροεπεξεργαστών καλύπτουν δύο σημαντικά επιστημονικά πεδία. Αρχικά, χρησιμοποιώντας το ολοκληρωμένο κύκλωμα των 48 πυρήνων με ονομασία Intel SCC, προτείνουμε μια τεχνική επιτάχυνσης του εντοπισμού μονίμων σφαλμάτων που εφαρμόζεται κατά τη διάρκεια λειτουργίας αρχιτεκτονικών με πολλούς πυρήνες, η οποία εκμεταλλεύεται το δίκτυο υψηλής ταχύτητας μεταφοράς μηνυμάτων που διατίθεται στα ολοκληρωμένα κυκλώματα αυτού του είδους. Δεύτερον, προτείνουμε μια λεπτομερή στατιστική μεθοδολογία με σκοπό την ακριβή πρόβλεψη σε επίπεδο συστήματος των ασφαλών ορίων λειτουργίας της τάσης των πυρήνων τύπου ARMv8 που βρίσκονται πάνω στη CPU X-Gene 2.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Αξιοπιστία, Φερεγγυότητα, Παροδικά Σφάλματα, Μόνιμα Σφάλματα, Ενεργειακή Απόδοση, Στατιστική Ανάλυση

Στην Άννα.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Dimitris Gizopoulos, who gave me the opportunity to work in very interesting research topics and projects of computer architecture, while he advised me throughout these years. His guidance and encouragement especially in times of failure have been invaluable to the conduction of my dissertation. The interaction with Professor Dimitris Gizopoulos provided me with several knowledge and skills that would be very helpful in my future career and life.

I also like to thank all my collaborators throughout these years and especially all the members of Computer Architecture Lab of University of Athens with whom I shared the good and the bad moments in the lab. I learned a lot from them, hoping the same for them. I also want to thank Dr. Shidhartha Das from ARM Ltd. in Cambridge for his hospitality and his guidance during my three-month PhD internship in UK.

Especially, I would also like to thank my family and my friends who always stand by me. Finally, I would like to thank Anna who encouraged me in the times of failure and celebrated my successes, while she always shows me the way to become a better man. I am grateful for her support and I would like to dedicate my dissertation to her.

LIST OF PUBLICATIONS

Peer reviewed conferences:

- G.Papadimitriou, A.Chatzidimitriou, M.Kaliorakis, Y.Vastakis, D.Gizopoulos, “Micro-Viruses for fast system-level voltage margins characterization in multicore CPUs”, IEEE International Symposium on Performance Analysis of Systems and Software (**ISPASS**), 2018.
- G.Karakonstantis, K.Tovletoglou, L.Mukhanov, H.Vandierendonck, D.S.Nikolopoulos, P.Lawthers, P.Koutsovasilis , M.Maroudas, C.D.Antonopoulos, C.Kalogirou, N.Bellas, S.Lalis, S.Venugopal, A.Prat-Perez, A.Lampropulos, M.Kleanthous, A.Diavastos, Z.Hadjilambrou, P.Nikolaou, Y.Sazeides, P.Trancoso, G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, D.Gizopoulos, S.Das, “An energy-efficient and error-resilient server ecosystem exceeding conservative scaling limits”, IEEE/ACM Design, Automation & Test in Europe Conference (**DATE**), 2018.
- G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, D.Gizopoulos, P.Lawthers, S.Das, “Harnessing voltage margins for energy efficiency in multicore CPUs”, IEEE/ACM International Symposium on Microarchitecture (**MICRO**), 2017.
- G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, C.Magdalinos, D.Gizopoulos, “Voltage margins identification on commercial x86-64 multicore microprocessors”, IEEE International On-Line Testing Symposium (**IOLTS**), 2017.
- A.Chatzidimitriou, M.Kaliorakis, D.Gizopoulos, M.Pipponzi, R.Mariani, S.Di Carlo, “RT Level vs. microarchitecture level reliability assessment: case study on ARM Cortex-A9 CPU”, IEEE/IFIP International Conference on Dependable Systems and Networks (**DSN**), 2017.
- M.Kaliorakis, D.Gizopoulos, R.Canal, A.Gonzalez, “MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment”, ACM/IEEE International Symposium on Computer Architecture (**ISCA**), 2017.
- A.Chatzidimitriou, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Performance-aware reliability assessment of heterogeneous chips”, IEEE VLSI Test Symposium (**VTS**), 2017.
- A.Vallero, A.Savino, G.Politano, S.Di Carlo, A.Chatzidimitriou, S.Tselonis, M.Kaliorakis, D.Gizopoulos, M.R.Villanueva, R.Canal, A.Gonzalez, M.Kooli, A.Bosio, G.Di Natale, “Cross-Layer system reliability assessment framework for hardware faults”, IEEE International Test Conference (**ITC**), 2016.
- S.Tselonis, M.Kaliorakis, N.Foutris, G.Papadimitriou, D.Gizopoulos, “Microprocessor reliability-performance tradeoffs assessment at the microarchitecture level”, IEEE VLSI Test Symposium (**VTS**), 2016.
- M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, D.Gizopoulos, “Accelerated microarchitectural fault injection-based reliability assessment”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (**DFTS**), 2015.
- M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, “Differential fault injection on microarchitectural simulators”, IEEE International Symposium on Workload Characterization (**IISWC**), 2015.
- A.Vallero, A.Savino, S.Tselonis, N.Fourtis, M.Kaliorakis, G.Politano, D.Gizopoulos, S.Di Carlo, “Bayesian network early reliability evaluation analysis for both permanent and transient faults”, IEEE International On-Line Testing Symposium (**IOLTS**), 2015.

- A.Vallero, A.Savino, S.Tselonis, N.Fourtis, M.Kaliorakis, G.Politano, D.Gizopoulos, S.Di Carlo, “A bayesian model for system level reliability estimation”, IEEE European Test Symposium (**ETS**), 2015.
- N.Foutris, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Versatile architecture-level fault injection framework for reliability evaluation: a first report”, IEEE International On-Line Testing Symposium (**IOLTS**), 2014.
- M.Kaliorakis, M.Psarakis, N.Foutris, D.Gizopoulos, “Accelerated online error detection in many-core microprocessor architectures”, IEEE VLSI Test Symposium (**VTS**), 2014.
- M.Kaliorakis, N.Foutris, D.Gizopoulos, M.Psarakis, "Online error detection in multiprocessor chips: A test scheduling study", IEEE International On-Line Testing Symposium (**IOLTS**), 2013.

Peer reviewed journals:

- M.Kaliorakis, A.Chatzidimitriou, G.Papadimitriou, D.Gizopoulos, “Statistical analysis of multicore CPUs operation in scaled voltage conditions”, **IEEE Computer Architecture Letters (CAL)**, Jan. 2018.
- A.Vallero, S.Tselonis, N.Foutris, M.Kaliorakis, M.Kooli, A.Savino, G.Politano, A.Bosio, G.Di Natale, D.Gizopoulos, S.Di Carlo, “Cross-layer reliability evaluation, moving from the hardware architecture to the system level: a CLERECO EU Project overview”, **Journal of Microprocessors and Microsystems**, June 2015.

Peer reviewed conferences/workshops with informal proceedings:

- G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, D.Gizopoulos, G.Favor, K.Sankaran, S.Das, “A system-level voltage/frequency scaling characterization framework for multicore CPUs”, IEEE Workshop on Silicon Errors in Logic - System Effects (**SELSE**), 2017.
- K.Tovletoglou, C.Chalios, G.Karakonstantis, L.Mukhanov, H.Vandierendonck, D.S.Nikolopoulos, P.Koutsovasilis, M.Maroudas, C.Antonopoulos, C.Kalogirou, N.Bellas, S.Lalis, M.M.Rafique, S.Venugopal, A.Prat-Perez, A.Diavastos, Z.Hadjilambrou, P.Nikolaou, Y.Sazeides, P.Trancoso, G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou D.Gizopoulos, “An energy-efficient and error-resilient server ecosystem exceeding conservative scaling limits”, Workshop on Energy-efficient Servers for Cloud and Edge Computing (**EnESCE**), in conjunction with HiPEAC 2017.
- A.Vallero, A.Savino, G.Politano, S.Di Carlo, A.Chatzidimitriou, S.Tselonis, M.Kaliorakis, D.Gizopoulos, M.Riera, R.Canal, A.Gonzalez, M.Kooli, A.Bosio, G.Di Natale, “Early component-based system reliability analysis for approximate computing systems”, 2nd Workshop On Approximate Computing (**WAPCO**) in conjunction with HiPEAC, 2016.
- M.Kaliorakis, D.Gizopoulos, “Ensuring dependability of modern computing systems”, 11th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (**HiPEAC ACACES**), 2015.
- 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**) in conjunction with HiPEAC, 2014.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Η εξέλιξη της τεχνολογίας κατασκευής ημιαγωγικών κυκλωμάτων και της αρχιτεκτονικής και σχεδίασης υπολογιστών δίνει στους σχεδιαστές ολοκληρωμένων κυκλωμάτων την ευκαιρία να ενισχύσουν την απόδοση (performance) των σύγχρονων υπολογιστικών συστημάτων τα οποία χρησιμοποιούνται σε διάφορους τομείς της πληροφορικής και των τηλεπικοινωνιών. Οι σχεδιαστές υπολογιστών μπορούν να βελτιώσουν την υπολογιστική απόδοση χρησιμοποιώντας πιο πολύπλοκες πολιτικές λειτουργίας των επεξεργαστών, καθώς η τεχνολογία ημιαγωγών μειώνει το μέγεθος των τρανζίστορ δίνοντας έτσι τη δυνατότητα ολοκλήρωσης όλο και περισσότερων τρανζίστορ πάνω στο ίδιο κύκλωμα. Έτσι από την μια πλευρά, η εξέλιξη της τεχνολογίας ημιαγωγών και της αρχιτεκτονικής υπολογιστών κάνει τους υπολογιστές όλο και πιο αποδοτικούς ως προς τις επιδόσεις, αλλά από την άλλη πλευρά τους κάνει όλο και πιο σύνθετους.

Όμως, αυτή η αύξηση σε απόδοση συνοδεύεται και από αύξηση στην ευπάθεια (ή αντίστοιχα μείωση στην αξιοπιστία) των επεξεργαστών καθώς η ποιότητα των προϊόντων περιορίζεται εξαιτίας: (α) των αυστηρών χρονοδιαγραμμάτων τα οποία ορίζονται για να μειωθεί ο χρόνος που απαιτείται μέχρι το προϊόν να κυκλοφορήσει στην αγορά (άρα και ο χρόνος για τον έλεγχο της αξιοπιστίας του), (β) των σύγχρονων τεχνικών κατασκευής ολοκληρωμένων κυκλωμάτων όλο και μικρότερης κλίμακας (κάνοντάς τα όλο και πιο ευάλωτα στην ακτινοβολία και πιο επιρρεπή σε κατασκευαστικές ατέλειες), και (γ) της αυξημένης πολυπλοκότητας του σχεδίου των κυκλωμάτων (κάνοντας σε πολλές περιπτώσεις τον έλεγχο ορθής λειτουργίας τους σε εύλογο χρονικό διάστημα πολύ δύσκολο). Ειδικότερα, οι σύγχρονοι μικροεπεξεργαστές αντιμετωπίζουν σοβαρά προβλήματα αξιοπιστίας κατά τη διάρκεια της ζωής τους εξαιτίας: (i) των σφαλμάτων που προέρχονται από την κοσμική ακτινοβολία και από τα φορτισμένα ηλεκτρικά σωματίδια που βάλλουν τα κυκλώματα ακόμα και στο επίπεδο της θάλασσας, (ii) της γήρανσης και φθοράς των κυκλωμάτων με την πάροδο του χρόνου, και (iii) των κατασκευαστικών ατελειών που δημιουργούνται κατά την παραγωγή των ολοκληρωμένων κυκλωμάτων. Κάποιες από αυτές τις κατασκευαστικές αστοχίες που κάνουν ακόμα και κυκλώματα που θεωρητικά είναι κατασκευασμένα να λειτουργούν υπό τις ίδιες συνθήκες τάσης τελικά να λειτουργούν ορθά κάτω από διαφορετικές συνθήκες, συνήθως ωθούν τους κατασκευαστές ολοκληρωμένων κυκλωμάτων στην υιοθέτηση απαισιόδοξων ορίων τάσης λειτουργίας, τα οποία κατ' επέκταση θυσιάζουν την ενεργειακή απόδοση των προϊόντων.

Για τους παραπάνω λόγους οι σχεδιαστές χρειάζεται να διασφαλίσουν υψηλά επίπεδα αξιοπιστίας και ενεργειακής απόδοσης των ολοκληρωμένων κυκλωμάτων πριν αυτά διοχετευθούν στην αγορά. Ο σκοπός αυτής της διδακτορικής διατριβής είναι να προτείνει νέες τεχνικές που εφαρμόζονται σε διαφορετικές φάσεις της διάρκειας ζωής των ολοκληρωμένων κυκλωμάτων με σκοπό την επίλυση σημαντικών προβλημάτων ως προς την αξιοπιστία και την ενεργειακή απόδοση των προϊόντων μικροεπεξεργαστών. Γενικά, η ζωή των ολοκληρωμένων κυκλωμάτων διακρίνεται στα ακόλουθα στάδια: (α) το πρώιμο σχεδιαστικό στο οποίο καθορίζονται οι απαιτήσεις του κυκλώματος στους τομείς της απόδοσης, της αξιοπιστίας και της κατανάλωσης ενέργειας, (β) το στάδιο του σχεδιασμού στο οποίο οι σχεδιαστές υλοποιούν με γλώσσες περιγραφής υλικού το κύκλωμα σύμφωνα με τις απαιτήσεις του προηγούμενου σταδίου, (γ) το στάδιο της παραγωγής στο οποίο αρχικά φτιάχνεται ένας μικρός αριθμός από κυκλώματα και μετά την επιτυχημένη αξιολόγησή τους συνεχίζεται η μαζική παραγωγή κυκλωμάτων, και (δ) το στάδιο κατά το οποίο τα κυκλώματα διοχετεύονται στην αγορά. Να σημειώσουμε ότι στο τελευταίο στάδιο της ζωής των ολοκληρωμένων κυκλωμάτων οι σχεδιαστές δεν έχουν πλέον φυσική επαφή με τα κυκλώματα και κατ' επέκταση πρέπει από τα προηγούμενα στάδια να διασφαλίσουν την υψηλή ποιότητα των προϊόντων ως προς την απόδοση, την αξιοπιστία και την ενεργειακή κατανάλωση.

Οι τεχνικές που προτείνονται σε αυτή τη διδακτορική διατριβή μπορούν να εφαρμοστούν κατά τη διάρκεια του πρώιμου σχεδιαστικού σταδίου (α), του σταδίου παραγωγής (γ) ή του σταδίου της κυκλοφορίας των ολοκληρωμένων κυκλωμάτων στην αγορά (δ). Στη συνέχεια παρουσιάζουμε συνοπτικά τις συνεισφορές της διατριβής αυτής ανάλογα με το στάδιο της ζωής των ολοκληρωμένων κυκλωμάτων στο οποίο εφαρμόζονται.

- **Πρώιμο σχεδιαστικό στάδιο**: Ένα πολύ σημαντικό βήμα κατά τη διάρκεια των πρώτων σταδίων της σχεδίασης των ολοκληρωμένων κυκλωμάτων είναι η εκτίμηση της αξιοπιστίας του κυκλώματος ως προς τα παροδικά σφάλματα (transient faults) που είναι πιθανόν να επηρεάσουν την ορθή λειτουργία των κυκλωμάτων. Οι απαιτήσεις ως προς της αξιοπιστία και την απόδοση καθορίζουν σχεδιαστικές αποφάσεις που εφαρμόζονται στα επόμενα στάδια της της ζωής των ολοκληρωμένων κυκλωμάτων, όπως είναι για παράδειγμα η ενσωμάτωση μηχανισμών στο κύκλωμα για την προστασία από τέτοιου είδους σφάλματα ή ακόμα και ο καθορισμός συγκεκριμένων μικροαρχιτεκτονικών παραμέτρων (π.χ. μέγεθος μικροαρχιτεκτονικών δομών και πολιτικών κλπ.) που μπορούν να επηρεάσουν τόσο την αξιοπιστία όσο και την απόδοση. Η ακριβής εκτίμηση της ανθεκτικότητας (ή αντίστροφα της ευπάθειας) των κυκλωμάτων ως προς αυτά τα σφάλματα είναι ζωτικής σημασίας για να αποφευχθεί κάποια οικονομική καταστροφή για την εταιρία κατασκευής τους σε περίπτωση που το πρόβλημα παρουσιαστεί αφού κυκλοφορήσει το προϊόν στην αγορά ή για να εξοικονομηθούν χρήματα, κόπος και τμήμα από τη διαθέσιμη συνολική επιφάνεια του ολοκληρωμένου κυκλώματος που μπορεί να σπαταληθεί στην υλοποίηση κάποιου μηχανισμού προστασίας που στην πραγματικότητα δεν είναι απαραίτητος. Για αυτούς τους λόγους, οι εταιρίες κατασκευής μικροεπεξεργαστών δίνουν ιδιαίτερη βαρύτητα σε αυτό το πρώιμο σχεδιαστικό στάδιο αξιολόγησης της αξιοπιστίας των κυκλωμάτων.

  Η στατιστική εισαγωγή παροδικών σφαλμάτων (statistical fault injection) σε δομές που είναι μοντελοποιημένες σε προσομοιωτές οι οποίοι στοχεύουν στην μελέτη της απόδοσης (performance simulators) είναι μια επιστημονικά καθιερωμένη μέθοδος για την ακριβή μέτρηση της αξιοπιστίας, αλλά υστερεί στον αργό χρόνο εκτέλεσης. Η στατιστική εισαγωγή παροδικών σφαλμάτων μοντελοποιείται στους προσομοιωτές αυτούς με την τροποποίηση της πραγματικής τιμής ενός bit μιας δομής υλικού όπως για παράδειγμα η στιγμιαία αλλαγή της τιμής ενός bit σε ένα αρχείο καταχωρητών ή σε κάποια κρυφή μνήμη.

  Η διδακτορική διατριβή παρουσιάζει διάφορες συνεισφορές στο επιστημονικό πεδίο που σχετίζεται με την αξιολόγηση της αξιοπιστίας των μονάδων υλικού στο πρώιμο σχεδιαστικό στάδιο των ολοκληρωμένων κυκλωμάτων. Αρχικά, με την εργασία [27] παρουσιάζουμε ένα νέο πλήρως αυτοματοποιημένο εργαλείο εισαγωγής σφαλμάτων σε μικροαρχιτεκτονικό επίπεδο (με την ονομασία MaFIN) που στοχεύει στην ακριβή αξιολόγηση της αξιοπιστίας ενός μεγάλου πλήθους μονάδων υλικού της μικροαρχιτεκτονικής x86-64, σε σχέση με διάφορα μοντέλα σφαλμάτων (παροδικά, διακοπτόμενα, μόνιμα σφάλματα). Για την ρεαλιστικότερη μοντελοποίηση της λειτουργίας των κυκλωμάτων και την ακριβή μέτρηση της αξιοπιστίας χρειάστηκε να ενσωματωθούν στον μικροαρχιτεκτονικό προσομοιωτή που χρησιμοποιήθηκε τα πεδία των δεδομένων σε όλες τις κρυφές μνήμες. Να σημειώσουμε ότι οι προσομοιωτές απόδοσης (performance simulators) είναι κατασκευασμένοι αποκλειστικά για το σκοπό αυτό με χρήση γλωσσών προγραμματισμού υψηλού επιπέδου όπως η C/C++ και κάποιες λειτουργίες που δεν επηρεάζουν τις μετρήσεις ως προς την απόδοση παραλείπονται ή μοντελοποιούνται με τέτοιο τρόπο ώστε να επιταχύνεται ο χρόνος της προσομοίωσης. Η ανάπτυξη του εργαλείου που χρησιμοποιείται για την ακριβή αξιολόγηση τόσο της αξιοπιστίας όσο και της απόδοσης στα πρώτα στάδια μελέτης αυτής της επιστημονικής περιοχής ήταν απαραίτητη καθώς τέτοιου είδους εργαλεία δεν υπάρχουν δημοσίως διαθέσιμα. Στη συνέχεια, χρησιμοποιώντας το ίδιο εργαλείο και στοχεύοντας τα παροδικά σφάλματα, παρουσιάζουμε δύο διαφορετικές μελέτες σχετιζόμενες με την αξιοπιστία και την απόδοση, οι οποίες μπορούν να βοηθήσουν τις αποφάσεις των σχεδιαστών ολοκληρωμένων κυκλωμάτων στα πρώιμα σχεδιαστικά στάδια.

Στην πρώτη μελέτη [28], αξιολογούμε την στενή σύνδεση αξιοπιστίας και απόδοσης σε μια μικροαρχιτεκτονική x86-64 μεταβάλλοντας τις σχεδιαστικές παραμέτρους σημαντικών μονάδων του υλικού του μικροεπεξεργαστή. Συγκεκριμένα, πειραματιζόμαστε με το μέγεθος του αρχείου φυσικών ακεραίων καταχωρητών (physical integer register file), με τις κρυφές μνήμες πρώτου και δεύτερου επιπέδου (L1 Instruction cache, L1 Data cache, L2 cache) και με την ουρά φόρτωσης/αποθήκευσης (load/store queue) και τροποποιούμε παραμέτρους όπως το μέγεθος όλων των δομών, την πολιτική εγγραφής (write policy) των κρυφών μνημών, το είδος συσχετιστικότητας των κρυφών μνημών (cache associativity) και την ύπαρξη ή όχι κυκλωμάτων εκ των προτέρων προσκόμισης (prefetchers) στις κρυφές μνήμες πρώτου επιπέδου. Στη συγκεκριμένη μελέτη, για το συγκερασμό των μετρήσεων απόδοσης και αξιοπιστίας προτείνουμε μια νέα συνάρτηση (με την ονομασία fitness function). Τα αποτελέσματα αυτής της μελέτης μπορούν να καθοδηγήσουν σχεδιαστικές αποφάσεις σχετικά με την αξιοπιστία και την απόδοση των πραγματικών μονάδων υλικού των μικροεπεξεργαστών.

Στη δεύτερη μελέτη [29], χρησιμοποιούμε τον MaFIN μαζί με ένα άλλο εργαλείο μέτρησης της αξιοπιστίας των μονάδων υλικού σε μικροαρχιτεκτονικό επίπεδο (με την ονομασία GeFIN [30]) με σκοπό να αξιολογήσουμε: (α) την ευαισθησία ως προς την αξιοπιστία διάφορων μικροαρχιτεκτονικών δομών του ίδιου αρχιτεκτονικού συνόλου εντολών (ISA x86-64) που είναι υλοποιημένα σε διαφορετικούς μικροαρχιτεκτονικούς προσομοιωτές, και (β) την ευαισθησία ως προς την αξιοπιστία μεταξύ δύο διαφορετικών αρχιτεκτονικών συνόλου εντολών (ARM και x86-64). Η μελέτη αυτή αποκάλυψε πολύ ενδιαφέροντα συμπεράσματα για τα χαρακτηριστικά των εργαλείων που χρησιμοποιούνται στις μελέτες αξιολόγησης της αξιοπιστίας και μπορούν να επηρεάσουν τις μετρήσεις, καθώς και τις μικροαρχιτεκτονικές διαφορές και τις παραμέτρους των αρχιτεκτονικών συνόλου εντολών (ISA) που επηρεάζουν την αξιοπιστία. Είναι η πρώτη φορά που καταγράφεται μια μελέτη αυτού του είδους και τα συμπεράσματά της μπορούν να βοηθήσουν τους αρχιτέκτονες υπολογιστών στην επιλογή των εργαλείων για την αξιολόγηση της αξιοπιστίας, στην επιλογή της καταλληλότερης αρχιτεκτονικής συνόλου εντολών, στην επιλογή της καταλληλότερης υλοποίησης μια δομής υλικού σε μικροαρχιτεκτονικό επίπεδο και στην επιλογή των κατάλληλων μηχανισμών προστασίας που χρειάζεται να προστεθούν με τελικό σκοπό να επιτευχθούν οι υψηλοί στόχοι αξιοπιστίας των προϊόντων που τίθενται στα πρώιμα σχεδιαστικά στάδια.

Μια μεγάλη πρόκληση για την αξιολόγηση της αξιοπιστίας των μονάδων υλικού ενάντια σε παροδικά σφάλματα σε μικροαρχιτεκτονικό επίπεδο χρησιμοποιώντας στατιστική εισαγωγή σφαλμάτων είναι ότι τα μαζικά πειράματα που τελικά εξασφαλίζουν εκτιμήσεις αξιοπιστίας υψηλής στατιστικής ακρίβειας είναι εξαιρετικά χρονοβόρα. Στη διδακτορική διατριβή παρουσιάζουμε δύο μελέτες για να επιταχύνουμε τα μαζικά πειράματα εισαγωγής σφαλμάτων υψηλής ακρίβειας.

Στην πρώτη μελέτη που παρουσιάζουμε [31] χρησιμοποιώντας τον MaFIN, προτείνουμε δύο διαφορετικές τεχνικές για να επιταχύνουμε τα πειράματα. Οι τεχνικές αυτές λαμβάνουν χώρα μετά από την πραγματική εισαγωγή των σφαλμάτων στις δομές του υλικού κατά τη διάρκεια ζωής του σφάλματος μέσα μέσα στη δομή. Συγκεκριμένα, στην πρώτη τεχνική που προτείνεται σταματάμε τα πειράματα της εισαγωγής σφαλμάτων μόλις το παροδικό σφάλμα επικαλυφθεί από κάποια άλλη εγγραφή στην ίδια θέση της μικροαρχιτεκτονικής δομής. Τα σφάλματα αυτά είμαστε σίγουροι ότι δεν μπορούν να επηρεάσουν τη σωστή λειτουργία του προγράμματος και κατ' επέκταση μπορούν να τερματίσουν πρόωρα χωρίς καθόλου απώλειες ακρίβειας στην τελική εκτίμηση. Στη δεύτερη τεχνική που προτείνεται στην ίδια μελέτη, σταματάμε τα πειράματα είτε όταν το σφάλμα έχει επικαλυφθεί (όμοια με την προηγούμενη τεχνική) είτε όταν το σφάλμα διαβάζεται από μια αρχιτεκτονική εντολή και αυτή η εντολή τελικά διεκπεραιώνεται (commit) χωρίς να περιμένουμε την ολοκλήρωση του προγράμματος. Η ανάγνωση αυτή του σφάλματος και η διεκπεραίωσή της από την εντολή σηματοδοτεί τη μετάβασή του από το μικροαρχιτεκτονικό στο αρχιτεκτονικό επίπεδο (επίπεδο του λογισμικού). Η δεύτερη προτεινόμενη τεχνική επιφέρει ακόμα μεγαλύτερη επιτάχυνση από την πρώτη, η οποία όμως συνοδεύεται από μερική απώλεια ακρίβειας. Η αξιολόγηση των προτεινόμενων τεχνικών έγινε σε 6 διαφορετικές δομές υλικού: στο αρχείο φυσικών καταχωρητών, στις κρυφές μνήμες πρώτου και δεύτερου επιπέδου (L1 Data cache, L1 Instruction cache, L2 cache) και σε δύο πεδία της ουράς φόρτωσης/αποθήκευσης (load/store queue data and address fields). Από την πειραματική διαδικασία παρατηρήσαμε ότι η πρώτη τεχνική που δεν επιφέρει απώλεια ακρίβειας δίνει επιτάχυνση που μπορεί να φτάσει μέχρι τις 2,92 φορές, ενώ η δεύτερη τεχνική που επιφέρει απώλεια ακρίβειας μπορεί να οδηγήσει σε επιτάχυνση μέχρι 4,06 φορές. Επιπλέον, παρατηρήσαμε ότι η απώλεια ακρίβειας που επέρχεται από τη δεύτερη προτεινόμενη τεχνική είναι μικρή (έως 6,58 ποσοστιαίες μονάδες) για τις περιπτώσεις των δομών που βρίσκονται μέσα στον επεξεργαστή (αρχείο φυσικών καταχωρητών και ουρά φόρτωσης/αποθήκευσης), ενώ δεν είναι αμελητέα (έως 20,13 ποσοστιαίες μονάδες) στην περίπτωση των κρυφών μνημών. Κατ' επέκταση, η δεύτερη τεχνική μπορεί να χρησιμοποιηθεί χωρίς κίνδυνο απώλειας ακρίβειας για τις δομές που βρίσκονται μέσα στον επεξεργαστή.

Στη συνέχεια, για να επιταχύνουμε ακόμα περισσότερο τα πειράματα στατιστικής εισαγωγής σφαλμάτων υψηλής στατιστικής ακρίβειας προτείνουμε μια νέα τεχνική που εφαρμόζεται σε μικροαρχιτεκτονικό επίπεδο με την ονομασία MeRLiN [32]. Η μέθοδος αυτή είναι διαφορετική από τις τεχνικές που περιγράφονται στο [31], γιατί εφαρμόζεται πριν από την πραγματική εισαγωγή των σφαλμάτων στις μονάδες του υλικού και την εκτέλεση των προσομοιώσεων. Πιο συγκεκριμένα, η μέθοδος MeRLiN μειώνει το πλήθος της αρχικής λίστας σφαλμάτων, τα οποία χρησιμοποιούνται σε μαζικά πειράματα εισαγωγής σφαλμάτων και διασφαλίζουν εκτίμηση αξιοπιστίας υψηλής στατιστικής ακρίβειας. Αυτή η αρχική λίστα σφαλμάτων χαρακτηρίζεται από το πολύ μεγάλο πλήθος σφαλμάτων που την αποτελούν, τα οποία επιβάλλεται να εισαχθούν ώστε η τελική εκτίμηση να είναι υψηλής στατιστικής ακρίβειας. Η εισαγωγή όλων αυτών των σφαλμάτων είναι από τους σημαντικότερους λόγους για την μεγάλη καθυστέρηση των μαζικών πειραμάτων.

Η μεθοδολογία MeRLiN βασίζεται σε δύο βασικά βήματα. Στο πρώτο βήμα γίνεται μια αρχική ανάλυση της εφαρμογής (χωρίς καμία εισαγωγή σφάλματος) με σκοπό τον εντοπισμό των χρονικών διαστημάτων για κάθε οντότητα υλικού κατά τη διάρκεια των οποίων ένα σφάλμα είναι σίγουρο ότι δεν πρόκειται να επηρεάσει την ορθή λειτουργία της εφαρμογής. Αυτά τα σφάλματα έπειτα από μία και μόνο εκτέλεση της εφαρμογής μπορούν να αφαιρεθούν από την αρχική λίστα σφαλμάτων χωρίς να χρειάζεται να τρέξουν πειράματα εισαγωγής και χωρίς να επηρεαστεί η ακρίβεια. Επιπλέον, κατά τη διάρκεια αυτού του βήματος της μεθοδολογίας καταγράφεται και αποθηκεύεται η πληροφορία της εντολής η οποία τελικά προσπελαύνει την μονάδα υλικού που φέρει το σφάλμα: δείκτης εντολής (RIP) και μετρητής μικροπρογράμματος (uPC) της εντολής. Αυτή η πληροφορία είναι απαραίτητη για το δεύτερο βήμα της μεθόδου. Στο δεύτερο βήμα της μεθόδου MeRLiN, χρησιμοποιούνται μόνο τα σφάλματα που απομένουν από το πρώτο βήμα της μεθόδου. Αυτά τα σφάλματα ομαδοποιούνται ανάλογα με την εντολή που τελικά προσπελαύνει τη μονάδα υλικού που φέρει το σφάλμα (αυτή η πληροφορία είναι καταχωρημένη από το πρώτο βήμα της μεθόδου). Μετά την ομαδοποίηση αυτή, μόνο κάποιοι αντιπρόσωποι από κάθε ομάδα επιλέγονται για εισαγωγή σύμφωνα με τη θέση του byte της δομής στο οποίο βρίσκεται το σφάλμα. Οι αντιπρόσωποι που έχουν επιλεγεί από κάθε ομάδα σφαλμάτων αναμένεται ότι επιφέρουν το ίδιο αποτέλεσμα στην εκτέλεση που προγράμματος με τα άλλα σφάλματα που περιέχονται στην ίδια ομάδα. Αυτή η υπόθεση έχει αξιολογηθεί στην πειραματική μας διαδικασία με τη χρήση της μεταβλητής της ομοιογένειας (homogeneity) των ομάδων που έχουμε ορίσει για το σκοπό αυτό. Η μέθοδος MeRLiN οδηγεί σε ακριβής εκτιμήσεις της αξιοπιστίας των μονάδων του υλικού, ενώ η εισαγωγή τελικά μόνο των αντιπροσώπων από κάθε ομάδα οδηγεί σε μεγάλη επιτάχυνση της διαδικασίας.

Για την αξιολόγηση της μεθοδολογίας του MeRLiN ως προς την επιτάχυνση και την ακρίβεια στοχεύσαμε τέσσερις αντιπροσωπευτικές μονάδες. Οι τρεις από αυτές είναι μονάδες που σχετίζονται με δεδομένα: (1) κρυφή μνήμη δεδομένων πρώτου επιπέδου - L1D cache, (2) αρχείο φυσικών ακέραιων καταχωρητών - physical integer register file, και (3) ουρά αποθήκευσης - store queue. Η τέταρτη μονάδα σχετίζεται με εντολές (ουρά εντολών - issue queue). Στα πειράματά μας αξιολογήσαμε τον MeRLiN για διαφορετικά μεγέθη των τεσσάρων μονάδων υλικού, ενώ το πλήθος σφαλμάτων της αρχικής λίστας σφαλμάτων είναι 60.000 σφάλματα, πλήθος που διασφαλίζει τελικές εκτιμήσεις αξιοπιστίας υψηλής στατιστικής ακρίβειας (~0,63% error margin, 99.8% confidence level).

Η πειραματική διαδικασία επιβεβαίωσε ότι ο MeRLiN εκτιμά την αξιοπιστία των δομών υλικού με πολύ μεγάλη ακρίβεια, αλλά ταυτόχρονα προσφέρει επιτάχυνση 93X, 225X, 68X και 28X κατά μέσο όρο για το αρχείο φυσικών καταχωρητών, την ουρά αποθήκευσης, την κρυφή μνήμη δεδομένων πρώτου επιπέδου και την ουρά εντολών όταν εκτελούνται 10 μετροπρογράμματα από τη σουίτα MiBench. Όταν εκτελούμε τη μεθοδολογία MeRLiN με 10 μετροπρογράμματα από τη σουίτα SPEC CPU2006, τότε κατά μέσο όρο η επιτάχυνση είναι 1644X, 2018X και 171X για το αρχείο φυσικών καταχωρητών, την ουρά αποθήκευσης και την κρυφή μνήμη δεδομένων πρώτου επιπέδου, αντίστοιχα. Η επιτάχυνση που προσφέρει ο MeRLiN σε συνδυασμό με τη διατήρηση της ακρίβειας των εκτιμήσεων αξιοπιστίας, βοηθούν την ανάλυση της αξιοπιστίας σε μικροαρχιτεκτονικό επίπεδο με χρήση της μεθοδολογίας στατιστικής εισαγωγής σφαλμάτων κατά τα πρώιμα σχεδιαστικά στάδια ενός επεξεργαστή μειώνοντας με αυτό τον τρόπο τον χρόνο που απαιτείται ώστε το προϊόν να κυκλοφορήσει στην αγορά.

Τέλος, άλλες μελέτες που βασίζονται στα παραπάνω εργαλεία και μεθόδους είναι οι εξής: [90] [91] [92] [93] [94] [95].

- **Παραγωγικό στάδιο και στάδιο κυκλοφορίας στην αγορά**: Οι συνεισφορές της διδακτορικής διατριβής σε αυτά τα στάδια της ζωής των ολοκληρωμένων κυκλωμάτων καλύπτουν δύο σημαντικά ερευνητικά πεδία: (α) τον εντοπισμό μονίμων σφαλμάτων σε αρχιτεκτονικές πολλών πυρήνων με σκοπό τη διασφάλιση της ορθής λειτουργίας τους, και (β) τη διασφάλιση της βέλτιστης ενεργειακής απόδοσης των πολυπύρηνων ολοκληρωμένων κυκλωμάτων σε συνδυασμό με τη διατήρηση της αξιοπιστίας τους σε υψηλά επίπεδα.

  Η πρώτη συνεισφορά που παρουσιάζουμε συνοπτικά σε αυτό το σημείο της διατριβής σχετίζεται με την επιτάχυνση της διαδικασίας εντοπισμού των μονίμων σφαλμάτων σε αρχιτεκτονικές πολλών πυρήνων [33]. Τα σφάλματα αυτού του είδους μπορούν να προκύψουν από κατασκευαστικές ατέλειες ή μπορούν να προκύψουν λόγω της γήρανσης του ολοκληρωμένου κυκλώματος αφού αυτό έχει κυκλοφορήσει και χρησιμοποιηθεί στην αγορά. Οι αρχιτεκτονικές με πολλούς πυρήνες εμφανίζουν ραγδαία ανάπτυξη τα τελευταία χρόνια, ενώ και η επιστημονική κοινότητα δίνει ιδιαίτερη βαρύτητα στον εντοπισμό τεχνικών για τη διασφάλιση της ορθής λειτουργίας τους.

  Σε αυτό το πλαίσιο και χρησιμοποιώντας το ολοκληρωμένο κύκλωμα των 48 πυρήνων με ονομασία Intel Single-chip Cloud Computer (SCC), προτείνουμε μια τεχνική επιτάχυνσης του εντοπισμού μονίμων σφαλμάτων σε αρχιτεκτονικές με πολλούς πυρήνες κατά τη διάρκεια λειτουργίας τους. Η προτεινομένη μέθοδος εκμεταλλεύεται το δίκτυο υψηλής ταχύτητας μεταφοράς μηνυμάτων που διατίθεται στα ολοκληρωμένα κυκλώματα αυτού του είδους με σκοπό την παραλληλοποίηση του προγράμματος ελέγχου ορθής λειτουργίας. Η επιτάχυνση εμφανίζεται στις περιπτώσεις που τα προγράμματα ελέγχου ορθής λειτουργίας δημιουργούν συμφόρηση περισσότερο στη μνήμη του κυκλώματος, παρά στην ίδια την κεντρική μονάδα επεξεργασίας. Η πειραματική διαδικασία απέδειξε ότι η προτεινόμενη μεθοδολογία επιτάχυνσης των προγραμμάτων ελέγχου που δημιουργούν συμφόρηση στη μνήμη μπορεί να επιφέρει επιτάχυνση μέχρι και 38,2X για όλους τους πυρήνες του επεξεργαστή. Επιπλέον, χρησιμοποιώντας την προτεινόμενη μεθοδολογία για προγράμματα ελέγχου λειτουργίας που δημιουργούν συμφόρηση στη μνήμη και συνδυάζοντάς την παράλληλα με προγράμματα ελέγχου λειτουργίας που δημιουργούν συμφόρηση στην κεντρική μονάδα επεξεργασίας μετρήσαμε επιτάχυνση 47,6X. Η προτεινόμενη μεθοδολογία επιταχύνει αισθητά τη διαδικασία εντοπισμού μονίμων σφαλμάτων σε αρχιτεκτονικές πολλών πυρήνων και μπορεί να χρησιμοποιηθεί είτε κατά την παραγωγική διαδικασία είτε μετά την κυκλοφορία του επεξεργαστή στην αγορά.

  Η δεύτερη συνεισφορά που εφαρμόζεται τόσο στο στάδιο της παραγωγής όσο και μετά την κυκλοφορία στην αγορά σχετίζεται με την εξασφάλιση της βέλτιστης ενεργειακής απόδοσης και αξιοπιστίας σε ένα πολυπύρηνο ολοκληρωμένο κύκλωμα με ονομασία X-Gene 2, το οποίο διαθέτει 8 πυρήνες μικροεπεξεργαστή τύπου ARMv8. Οι διάφορες κατασκευαστικές ατέλειες των ολοκληρωμένων κυκλωμάτων προκαλούν τη διαφοροποίηση των ορίων ασφαλούς λειτουργίας (ως προς την τάση) ακόμα και ανάμεσα σε όμοιους πυρήνες που υποτίθεται ότι έχουν σχεδιαστεί ώστε να λειτουργούν ορθά υπό τις ίδιες ακριβώς συνθήκες. Επιπλέον, οι ίδιες ατέλειες και το φαινόμενο της γήρανσης του κυκλώματος μπορεί να οδηγήσει σε λάθη χρονισμού (timing errors) που μπορούν να επηρεάσουν την ορθή λειτουργία του κυκλώματος όταν αυτό έχει ήδη κυκλοφορήσει στην αγορά. Για την αντιμετώπιση αυτών των φαινομένων, οι σχεδιαστές συχνά υιοθετούν απαισιόδοξα όρια τάσης λειτουργίας των επεξεργαστών, κατασπαταλώντας με αυτό τον τρόπο την ενεργειακή απόδοση του κυκλώματος στο βωμό της αξιοπιστίας. Ο εντοπισμός των βέλτιστων τιμών τάσης λειτουργίας ώστε να υπάρχει μια ισορροπία ανάμεσα στην αξιοπιστία και την ενεργειακή κατανάλωση είναι ζωτικής σημασίας για τα σύγχρονα υπολογιστικά συστήματα.

Προς αυτή την επιστημονική κατεύθυνση, στη διατριβή αυτή παρουσιάζουμε τις μελέτες [35] [167], στις οποίες προτείνουμε μια στατιστική μεθοδολογία που βασίζεται στη μέθοδο linear regression (γραμμικής παλινδρόμησης) με σκοπό την ακριβή πρόβλεψη σε επίπεδο συστήματος των ασφαλών ορίων λειτουργίας της τάσης των πυρήνων τύπου ARMv8 που βρίσκονται πάνω στο ολοκληρωμένο κύκλωμα X-Gene 2. Η πειραματική διαδικασία απέδειξε ότι μπορούμε να προβλέψουμε το ασφαλές όριο τάσης λειτουργίας των πυρήνων με πολλή μεγάλη ακρίβεια κερδίζοντας σε ενέργεια έως 20,28%.

Ελπίζουμε ότι οι τεχνικές που παρουσιάζονται στη διατριβή αυτή, θα ενισχύσουν την αξιοπιστία και την ενεργειακή απόδοση των σύγχρονων επεξεργαστών. Τέλος, ελπίζουμε ότι οι προσεγγίσεις μας θα αποτελέσουν εφαλτήριο πολλών μελλοντικών ερευνητικών μελετών βελτιώνοντας την ποιότητα των σύγχρονων υπολογιστικών συστημάτων.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This thesis was conducted in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens during the period 2014-2018.

1. INTRODUCTION

The evolution in semiconductor technology and computer architecture gave designers the opportunity to boost the performance of modern computing systems that are used in several domains of information and communication technology systems. Despite the changes in Moore's Law [1] and Dennard Scaling [2], computer architects and designers are still able to improve processor performance by using more aggressive and sophisticated techniques. The combined effect of the progress in semiconductor technology and the evolution in computer architecture makes microprocessors more efficient, but on the other hand much more complex.

However, the scaling in performance is also accompanied by increase in the vulnerability (or decrease in reliability) of microprocessors due to: (a) the strict deadlines that are required to minimize Time-to-Market (TTM) (minimizing also the time needed to test the circuits), (b) the modern device integration techniques that make processors more vulnerable to the radiation and also increase the occurrence of manufacturing defects, and (c) the increased design complexity that makes the testing process of the microprocessor products very difficult and unaffordable for the available TTM. Specifically, the modern microprocessors face serious reliability issues during their entire life-cycle due to: (i) the errors that come from *transient faults* caused by cosmic rays, alpha particles and electromagnetic interference and are manifested as instantaneous flips of the values of real hardware bits, (ii) aging that leads to errors that appear at regular time intervals (*intermittent errors*) or exist indefinitely (*permanent errors*), and (iii) manufacturing defects that can either be manifested as permanent errors or lead to timing errors when the chips operate beyond their nominal voltage and frequency conditions. These manufacturing imperfections usually force computer architects to adopt pessimistic operation margins in terms of voltage in order to protect the chips, while sacrificing the energy efficiency of the delivered product.

Designers need to ensure sufficient reliability levels (according to the needs of the application domain) and energy efficiency levels of the chips after they are released to the market. The goal of this thesis is to propose different techniques that are applied in different phases of the entire life-cycle of a microprocessor in order to solve important reliability challenges of current and future microprocessor products.

1.1 The evolution of microprocessor design

The use of electronic devices is increased every year far beyond all the pessimistic expectations from the 1971's, when the first commercial processor (Intel 4004 [3]) was released to the market. Indicatively, Figure 1 presents the increase of population of global connected electronic devices in the past years (starting from 2015) until nowadays (in 2018). Also, the same figure presents a forecast for the population of the devices that are going to be connected each year until 2025. All these devices are used in all the major fields of the electronics ecosystem such as Desktop PCs, Portable PCs, Smartphones and Tablets. From 2015 until this year (2018), we observe a 1.50X increase in the population of consumer electronic devices that corresponds to 23.14 billions of devices, while for the next years there will be an explosion (increase of 3.26X from 2018 to 2025 that is translated into 75.44 billions of devices). Note that the field of Internet of Things (IoT) that rapidly evolves the last years is a major contributor to this explosion of the total population of the electronic devices that are used in the entire ecosystem of electronics.

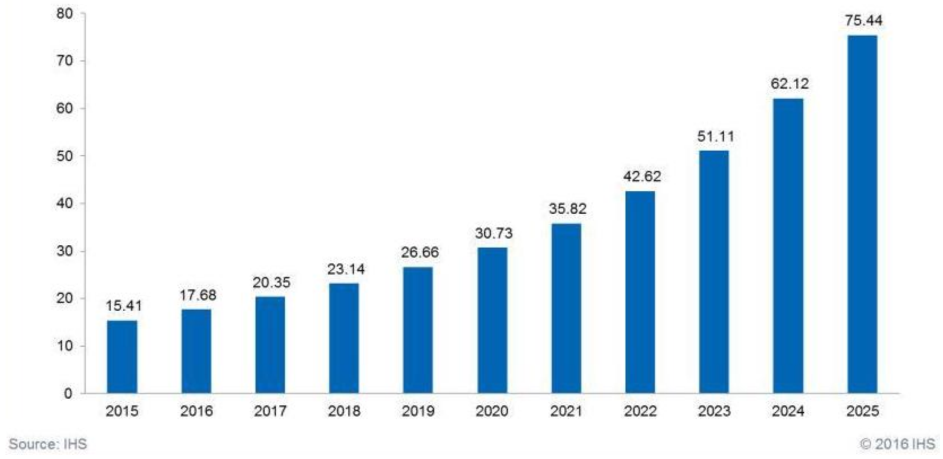


Figure 1: The evolution forecast of global connected electronic devices until 2025 in Billions of electronic devices (source: IHS, www.ihs.com).

Moreover, as the size of transistors shrinks, much more transistors are inserted on the same silicon die. Figure 2 illustrates the transistor density of a chip die per mm^2 in conjunction with the evolution of technology nodes starting from the 45nm lithography process that was widely used in 2008, until the 10nm that is currently used. From this figure, it is observed that there is a large increase equal to 30.55X in the transistors density that are used in the same die during the last 10 years, starting from 3.3 Millions per mm^2 in 2008 to 100.8 Millions of transistors per mm^2 in 2018. Today, the feature size that is used for the new massively produced chips is in the range of 10nm and is already announced that some chips will start to use 7nm technology (e.g. AMD is going to reveal AMD Starship with up to 48 Zen 2 Cores in Q2 of 2018 based on 7nm FinFET technology process [4]).

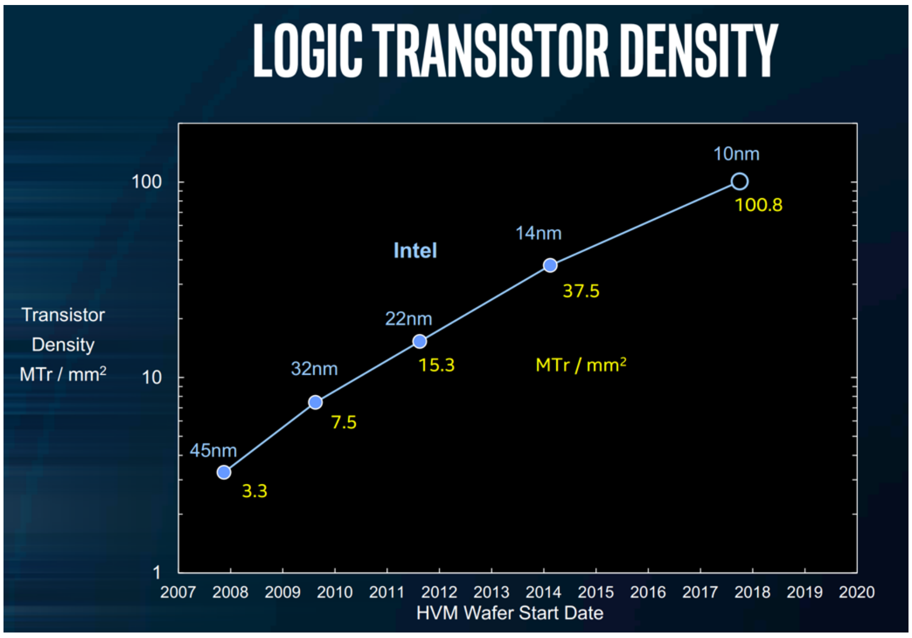


Figure 2: Transistor density per Million Transistor per mm^2 (MTr / mm^2) until 2018 [5].

The increase of transistors density and the evolution in lithography process gives computer architects the opportunity to implement more aggressive and sophisticated mechanisms in their chips that deliver superior levels of scalability by building and using multi-dimensional transistors. These mechanisms that enhance performance of microprocessors include aggressive speculative mechanisms (e.g. branch prediction units, data and instruction prefetchers, and value predictors), larger capacity structures (caches, memories, queues) and mechanisms that exploit parallelism of different levels (Instruction-Level, Thread-Level and Data-Level parallelism). Figure 3 illustrates the increase in microprocessor performance since 1978 compared to the performance relative of the VAX 11/780. From this figure, we observe a 25% increase in processor performance per year before 1986 that was largely technology driven. From 1987 to 2004, the increase in performance was about 52% per year that mainly came from more advanced architectural and organizational ideas. Finally, from 2003 to 2010, processor performance improvement has dropped to about 22% per year mainly due to the following obstacles: (a) heat dissemination, (b) little Instruction-Level Parallelism (ILP) left to exploit, and (c) limitations due to memory delays. These obstacles signal historic switch from relying solely on ILP to Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP) [6].

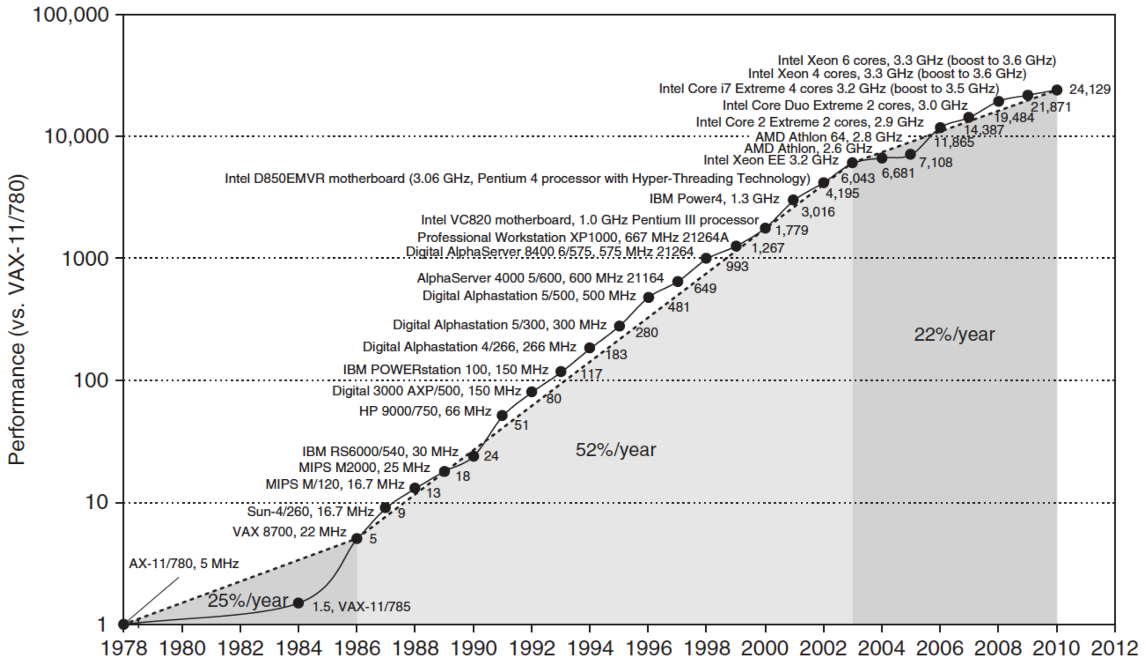


Figure 3: Increase in processor performance relative to the VAX 11/780 as measured by the SPECint benchmarks.

However, the implementation of aggressive and sophisticated mechanisms to boost performance in conjunction with the decrease of the size of transistors make the chips more and more complex, while they also suffer from many reliability issues that are manifested before or after they are released to the market. There are many reports of microprocessors operational failures of real hardware chips that can come from: (i) transient faults [7] [8] [9], (ii) aging [10], and (iii) manufacturing defects that can be manifested as permanent errors [11] or can lead to operation divergences among the chips of the same design; this forces the designers to adopt pessimistic voltage and frequency operation margins sacrificing a part of the energy efficiency or the performance of the product in order to ensure the correct operation of microprocessors [12].

To handle all these reliability issues the major microprocessor manufacturers such as Intel, AMD, ARM and IBM spend a huge amount of human resources, time, chip area and money to protect the chips by developing techniques that are implemented either before or during the fabrication process of the chips or after they are released to the market. On top of all these challenges that the companies have to handle, they also have to face the strict deadlines of minimizing Time-to-Market (TTM) and satisfying the high requirements of the costumers in terms of high reliability, performance and energy efficient requirements of the consumed electronic devices. Towards this direction, this thesis provides solutions that can be implemented in different phases of the life-cycle of microprocessors in order to boost their reliability.

1.2 Design life-cycle of microprocessors

Each microprocessor product has a long life-cycle that consists of many design, analysis and manufacturing phases before it is integrated in a computing system and released to the market. Figure 4 presents all the design phases of a microprocessor that are summarized below:

- **Planning**: In this phase, computer architects and designers define the microprocessor product design and manufacturing strategy. In particular, they define: (i) the product requirements in terms of functionality, performance, power and reliability, (ii) the technology node that is going to be used, (iii) the design methodology, and (iv) the tools that are going to be used during the entire manufacturing procedure. All these decisions are guided according to the nature of the market segment that the product targets, the budget and finally the TTM. For instance, a microprocessor that targets the market of aerospace or automotive should satisfy different reliability and power consumption requirements compared to a product that is going to be used in the desktop market segment or in low-end consumer electronics.

  Today, during the design planning phase computer architects are able to model in detail all the major architectural and microarchitectural features of the microprocessor product (such as all the assembly instructions of the entire Instruction Set Architecture - ISA, the size of the hardware components, the different microarchitectural policies etc.) by using tools that are written in a high-level programming languages, such as C/C++ [13] [14] [15]. Apart from the processor, these tools give architects the opportunity to model the entire system stack; starting from the lower hardware levels up to the higher levels of the firmware and the operating system. This model represents the first formalized reference of the final system's behavior. Thus, these models of high abstraction level make designers able to predict to a certain extend if the features that were selected for the product at this early design phase are suitable to meet all the design targets in terms of functionality, performance, reliability and power before starting the next design and fabrication procedures.

- **Development**: After satisfying the architectural and microarchitectural requirements that were defined in high level of abstraction during the design planning phase, the designers are ready to proceed into the next phase where the actual hardware design takes place. In the first stage of this phase, a Hardware Description Language (HDL), such as Verilog or VHDL, is exploited to describe and simulate the hardware design. The logic design phase passes through three stages depending on the level of abstraction of the described hardware model. The first stage is the implementation of the behavior model that maps major microprocessor events without the time notion. Next, the Register Transfer Level - RTL models the processor clock along with the detailed description of the events occurred in each clock cycle and finally, the structural level model that represents the gate - level implementation of the design.

  The last phase on the development cycle of a microprocessor incorporates the circuit and layout design processes. 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. At the end of the layout design phase, the first silicon prototypes are produced.

- **Production**: During this phase of processor life-cycle the first silicon prototypes are manufactured and thoroughly validated. Moreover, during this phase several design fixes are applied in order to adjust processor functionality and performance according to the design specifications.

- **Runtime**: This is the last phase of the life-cycle of a microprocessor, where the product is massively shipped to the market. At this stage, the microprocessor is fully functional and meets all the design specifications. After the release of the product to the market, the design teams stop to have any interaction with the developed design.

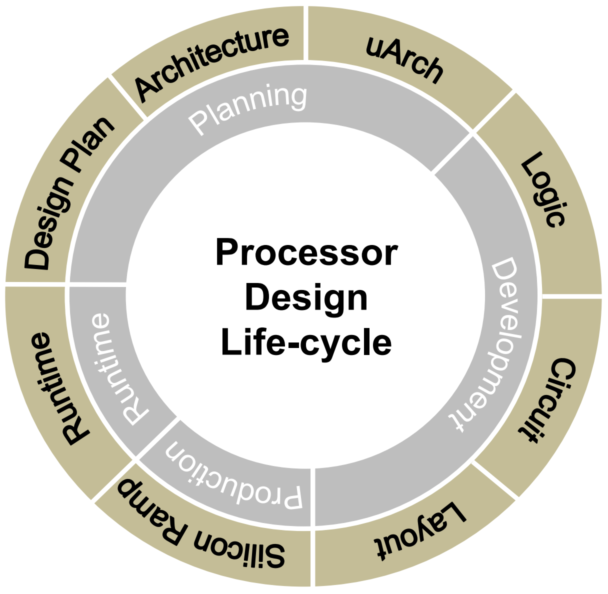


Figure 4: Processor design life-cycle.

1.3 Reliability life-cycle of microprocessors

As the field of computer architecture evolves and the size of transistors shrinks, computer architects are facing progressively more challenges to ensure that the delivered products satisfy high quality levels. Reliability is defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [16] [17]. A product that does not satisfy its reliability requirements could lead to financial disaster for the manufacturer or even to jeopardize the human life if the product is used in safety critical computing domains such as automotive or aerospace. Thus, manufacturers choose to spend extra effort, time, budget and chip area to ensure the correct operation of the delivered products. To achieve high reliability requirements, manufacturers apply a sequence of verification tasks during the entire microprocessor life-cycle in order to protect the chips' functionality from the different kind of errors that could be manifested after the products' release to the market. These verification tasks that are applied during the processor life-cycle are illustrated in Figure 5 and are summarized below:

- **Reliability Estimation**: During this verification task computer architects assess the expected reliability level of a microprocessor for any fault model (i.e. transient, permanent or intermittent faults). The early reliability assessment of a microprocessor is vital for two reasons. Firstly, to determine the microarchitectural design parameters that could influence the vulnerability of the product in order to satisfy the defined reliability requirements of the design planning phase. For instance, the size and design specifics of the hardware structures can influence the vulnerability of the processor apart from its performance. Secondly, to determine design decisions related to the required mechanisms for in-field error detection and recovery/repair. These mechanisms may impose significant area, power and performance overheads. Thus, inaccurate assessment of the reliability during the early design phases could easily make the cost of protection unaffordable or lead to an expensive, over-designed chip. For example, typical memory error protection and detection techniques can have a cost that ranges from 1% to 125% in terms of added memory capacity, depending on the complexity of the protection mechanism [18]. Moreover, detection and protection mechanisms against any fault model must be decided as early as possible in order to avoid costly redesign cycles for late integration of such mechanisms.

- **Pre-Silicon Verification**: This verification task is mainly based on simulation using tests on RTL model that are compared to those of the golden architectural model. Any discrepancies and indicators of design bugs are identified during this stage and fixed. Moreover, the pre-silicon verification engineers employ formal methods based on mathematical proofs to guarantee the absence of certain types of errors. Unfortunately, these methods lack of scalability when they target complex RTL models; thus, they are used only to a few limited blocks.
- **Silicon Debug**: This is the verification task in which the validation and debugging of a new microprocessor design on its first silicon prototype chips takes place. The goal of this task is to detect any "difficult" bug that escaped from the pre-silicon verification task and to ensure that a chip's actual silicon implementation matches its specification as was defined in the early design planning phase. For this reason, during silicon debug a comprehensive suite of test programs (such as automated generated random test programs, legacy tests and real word applications) covering many test scenarios are executed on the prototype chips to detect any anomalous behavior. When a bug is detected in this stage, the RTL model has to be modified to correct the issue and the manufacturing process of some prototypes must start again.
- **Manufacturing Testing**: This is the next task after the silicon debug, when some test metrics are used such as stuck-at coverage, transition fault coverage and N-detect coverage. When a sufficient level of coverage is reached for every single fabricated chip, then the chip is ready for market release. Manufacturing testing techniques aim to maximize the fault coverage, while minimizing the test costs in terms of time and resources. The traditional manufacturing testing approaches can be divided into functional and structural test approaches [19]. The functional approaches such as the Software-based Self-testing (SBST) utilize the on-chip programmable resources to apply at-speed the test and collect the test responses from memory in order to make the final pass/fail decision. On the contrary, structural test approaches (such as the scan-based testing) exploit the knowledge of the circuit structure and the corresponding fault model to generate the test patterns. Structural testing usually places the design in specific self-test mode and may cause excessive power consumption and over-testing; consequently, the yield loss of structural approaches is higher compared to the loss of functional methods.
- **In-field Verification**: This task contains all the protection mechanisms that are implemented on the chips to ensure their functionality after they are released to the market. Note that at this stage, the designers do not have interaction with the design product; thus, they should carefully have implemented efficient protection mechanisms against the aging and wear-out effects, failures that may come from manufacturing defects and process variation, as long as failures that come from the environmental phenomena such as cosmic rays, alpha particles and electromagnetic interference. Dual- and triple- modular redundancy are some

very common protection techniques, but they are accompanied by high costs. Also, parity and Error Correction Code (ECC) techniques are very common to protect buses, memories or other array-based structures of the microprocessor. Finally, there are many proposed techniques to protect SRAM caches [20], pipeline flip-flops and combinational logic [21].

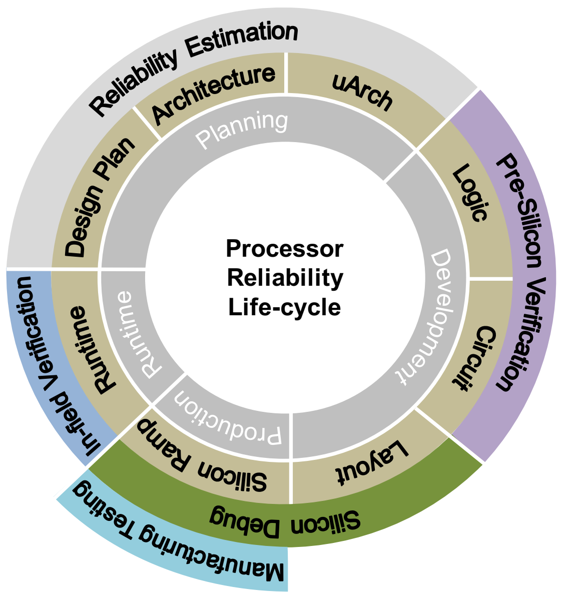


Figure 5: Processor reliability life-cycle.

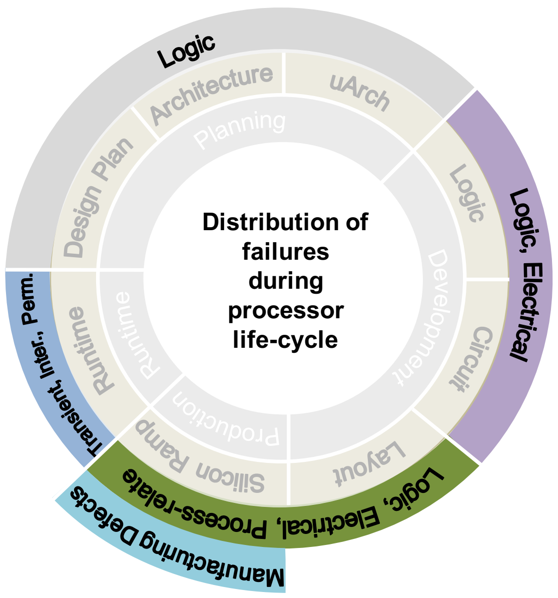


Figure 6: Distribution of failures during processor life-cycle.

1.4 Distribution of failures in different phases of the processor life-cycle

There are different types of failures that can be introduced during the lifetime of a microprocessor. The distribution and the nature of these failures during processor reliability life-cycle are presented in Figure 6 and summarized below:

- **Design bugs**: This category contains all the logic, electrical and process-related bugs [22] [23] that are introduced during the design planning and the development phase. The common sources of these bugs are [24]: (i) the limited throughput of the verification techniques that cannot keep pace with the complexity and the amount of the developed code that are used to design the modern microprocessors, (ii) the synthesis tools that may impede the accuracy of the synthesized design leading to functional inaccuracies between the intended and the developed design, (iii) the inaccuracies that are created during the place and route process leading to partial achievement of the design specifications during the layout process, and (iv) the combination of process variations and smaller design margins that prevents microprocessors to work at the intended frequency and voltage levels.
- **Manufacturing defects [25]**: This category contains all the manufacturing-related defects that are introduced in the design during the high-volume manufacturing (HVM) phase. These inaccuracies can be the result of optical proximity effects, airborne impurities, and processing material defects. Moreover, as the gate oxides have become so thin, transistors functionality can be easily affected by the variations of the current. Most of these defects are detected by dedicated machines (testers) with a very time-consuming procedure; but still some untested defects escape to the field.

- **In-field errors [25]**: This category contains all the failures that can be manifested after the chip is released to the market. These malfunctions can be: (i) transient errors that can be created by Single Event Upsets (SEUs), which potentially corrupt the computational logic and state bits, (ii) intermittent and permanent errors that can come from material aging and wear-out effects, or they could have also escaped from the high-volume production testing, and (iii) process variation defects that make microprocessors chips that are designed to be identical to present divergences in terms of performance and power consumption. Note that the reliability assessment of the chip concerning the transient, intermittent and permanent faults takes place in the very early stages of the design.

For the manufacturers, the detection of any kind of malfunction is of great importance to take place as soon as possible during the life-cycle. The reason is that the cost of redesign the product and fix any detected bug or add any protection mechanism increases significantly throughout the microprocessor life-cycle. Figure 7 illustrates the relative cost of finding bugs for all the phases of the microprocessor's lifetime [26]. The cost to re-start the design planning phase when the manufacturer detects a bug before the start of the layout process ranges to hundreds of dollars, while the cost explodes to more than tens of million dollars when the detection of the bug takes place after the massive release of the product to the market.

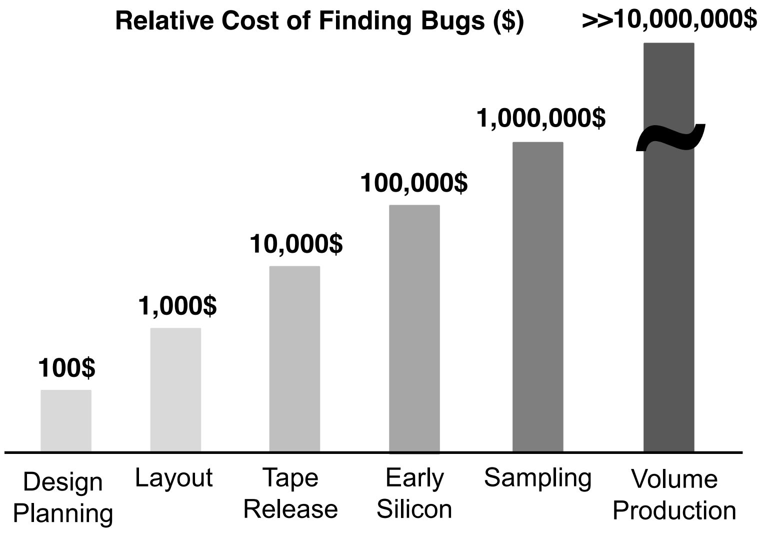


Figure 7: Relative cost of finding bugs throughout processor life-cycle [26].

1.5 Contribution of this thesis

The validation techniques that are implemented throughout the processor life-cycle in order to protect the chips from the different types of malfunctions are very important to ensure the design requirements in terms of functionality, performance, power and reliability of the delivered products. The goal of this thesis is to provide solutions to different validation challenges during the products' lifetime. The contributions of this thesis can be grouped in the two following categories according to the time interval they can be used during the microprocessor life-cycle (see Figure 8):

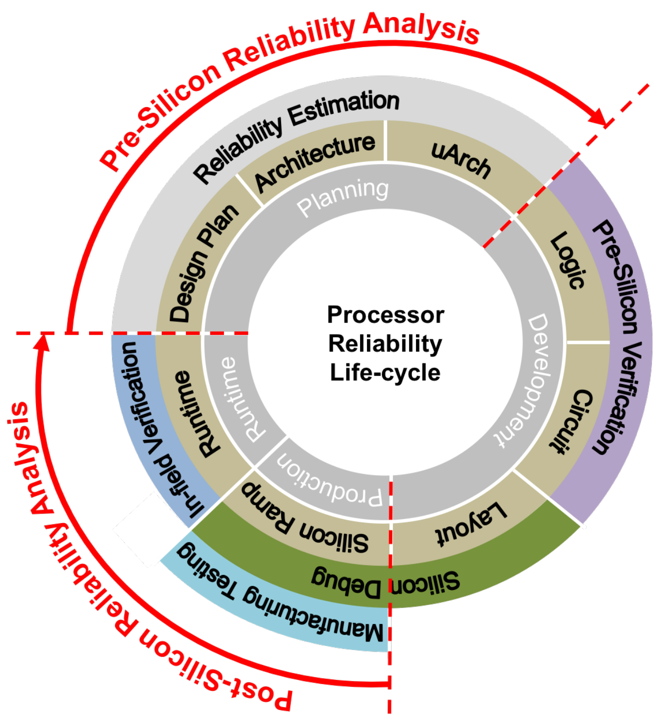


Figure 8: Contributions of this thesis in the processor life-cycle.

- **Pre-Silicon Reliability Analysis**: A very important task during the early design phases is the reliability estimation of the hardware structures and the entire chips against transient faults. The reliability and performance requirements that are defined during the planning design phase can guide several design decisions in the next phases of the processor life-cycle, such as implementation of protection mechanisms or even determination of several microarchitectural features (size of hardware structures, policies, etc.) that can influence not only the vulnerability of a chip but also its performance. Statistical fault injection of transient faults (flips of real hardware bit values) on microarchitectural structures modeled in performance simulators is a state-of-the-art method to accurately measure the reliability, but suffers from low simulation throughput.

  This thesis presents several contributions in the research field of *pre-silicon reliability analysis* phase of processors reliability life-cycle. Firstly, in [27] we present a novel fully-automated versatile architecture-level fault injection framework (called MaFIN) that is built on top of a state-of-the-art x86-64 microprocessor simulator (called Marssx86), for thorough and fast characterization of a wide range of hardware components with respect to various fault models (transient, intermittent, permanent faults). Next, by using the same tool and focusing mainly on the transient faults we executed two reliability evaluation studies. In the first study, we evaluated the reliability and performance tradeoffs for major hardware components of an x86-64 microprocessor across several important parameters of their design (size, associativity, write policy, etc.) [28]. In the second study [29], we used MaFIN in conjunction with a different tool (called GeFIN [30]) that is also used for early reliability assessments at the microarchitecture level to evaluate in a differential way: (a) the reliability sensitivity of several microarchitecture structures for the same ISA (x86-64) implemented on two different simulators, and (b) the reliability of workloads and microarchitectures for two popular ISAs (ARM vs. x86-64).

A major challenge of the early reliability assessments to soft errors at the microarchitecture level using statistical fault injection is that the campaigns that provide estimations of high statistical significance require excessively long experimental time. This thesis addresses this challenge by proposing two methodologies. Firstly, we propose to accelerate the individual fault injection campaigns by using several techniques that are implemented in the simulator and take place after the fault is actually injected in the hardware structure [31]. Secondly, to further accelerate the microarchitecture level fault injection campaigns we propose MeRLiN methodology [32] that provides a final speedup of several orders of magnitude, while keeping the accuracy of the assessments unaffected even for large injection campaigns with very high statistical significance. The core of this methodology is the pruning of the initial fault list by grouping the faults in equivalent classes according to the instruction that finally accesses the faulty entry. Faults that belong to the same group are very likely to lead to the same fault effect; thus, fault injection is performed only in a few representatives from each group. MeRLiN methodology constitutes a major breakthrough in the field of accelerating the reliability estimations of hardware components at the microarchitecture level with negligible loss of accuracy.

- **Post-Silicon Reliability Analysis**: Another important phase during the processor reliability life-cycle is the *Post-Silicon Reliability Analysis* that consists of the manufacturing testing and the in-field verification that take place during the fabrication process and after the release of the chips to the market, respectively. Note that in contrast to *Pre-Silicon Reliability Analysis*, in this phase the validation targets implemented circuits and especially after their release to the market the designers have no longer interaction with the design. The contributions of this thesis in this phase of the life-cycle cover two important research fields:
    - **Acceleration of permanent fault online detection in many-core architectures**: The extreme complexity of many-core processor architectures and the pressure for reduced time-to-market renders even the most comprehensive verification and testing campaign before and during mass production incomplete. A significant population of manufacturing faults escape in the field of operation and jeopardize correctness of the chip. Online functional testing is an attractive low-cost error detection solution, but it should be fast enough in order to not impact the system performance. This thesis faces this challenge by proposing an effective parallelization methodology [33] to accelerate online error detection for many-core architectures by exploiting the high-speed message passing on-chip network to accelerate the parallel execution of the test preparation phase of memory-intensive test programs. To demonstrate the efficiency of the proposed methodology we used a 48-core real hardware chip, Intel's Single-chip Cloud Computer (SCC) [34].
    - **Statistical analysis to predict the safe voltage margins in multicore CPUs for energy efficiency**: Reduction of the voltage operation margins of multicore chips is a major challenge for the designers to gain in terms of power**.** Unfortunately, this reduction leads to several reliability issues due to the manufacturing defects that make hardware cores of the same chip to present variations in their safe voltage and frequency operation limits. These variations that remain constant after the release of the chip to the market are classified as static variations. On top of that, transistor aging and dynamic variations in supply voltage and temperature, caused by different workload interactions can also affect the correct operation of a microprocessor. Thus,

the designers choose to insert conservative guard-bands in the operating voltage (and frequency) to protect the chips from the effects of the static and the dynamic variations, despite the induced cost in terms of energy (and performance). The contribution of this thesis to this challenge is to propose a detailed statistical analysis methodology [35] [167] to accurately predict at the system level the safe voltage operation margins of the eight ARMv8 cores of the X-Gene 2 chip [36] fabricated on 28nm technology. Our analysis uses as inputs the microprocessor's performance counters values of benchmarks that were collected in nominal voltage conditions execution and the results of the characterization phase when the chip operates in scaled voltage conditions.

1.6 Thesis outline

The remainder of this thesis is organized as follows:

Chapter 2 and Chapter 3 present all the contributions of this thesis during the *Pre-Silicon Reliability Analysis* phase focusing on the early reliability estimation of several microarchitectural structures to transient faults. Chapter 2 is divided in two parts: the first part shows the background details of reliability assessment studies, while the second part firstly presents a versatile fault injection framework (called MaFIN) to evaluate the reliability of modern x86-64 microprocessors. Next, the same part illustrates three studies that were launched using MaFIN to: (a) assess the reliability-performance tradeoffs, (b) evaluate different ISAs, microarchitectures and tools in terms of reliability, and (c) to accelerate the fault injection runs based on the faults lifetime. Finally, Chapter 3 illustrates MeRLiN, a state-of-the-art method to accelerate the fault injection campaigns of high statistical significance with negligible loss of accuracy by pruning the faults of the initial fault list.

Chapter 4 is divided in two parts that discuss two proposed techniques that can be implemented during *Post-Silicon Reliability Analysis* phase, when the massive production of chips begins or the chips are already released to the market. The first part presents a proposed technique that targets to accelerate the online detection of permanent faults in many-core architectures using the 48-core Intel's SCC architecture as experimental vehicle, while the second describes a statistical analysis methodology that was proposed to boost the energy efficiency of the eight ARMv8-based cores of the X-Gene 2 chip by predicting the safe voltage operation margins of each individual core.

Finally, Chapter 5 presents the conclusions of this thesis and discusses possible directions for future work.

2. PRE-SILICON RELIABILITY ANALYSIS

2.1 Background of early reliability estimation at the microarchitecture level

The reliable operation of modern and forthcoming computing systems can be affected by different types of hardware faults such as transient, intermittent, and permanent faults [37] [38] [39]. Transient faults (on which this chapter mainly focuses on) can be caused by external factors such as cosmic rays, alpha particles and electromagnetic interference. As the size of transistors scales, more and more reliability issues arise due to transient faults that hit commercial chips and are recorded in the literature [7] [8] [9].

Early assessment of the expected reliability of a computing system (or equivalently its resiliency to hardware faults including also the transient faults) is an important task which steers design decisions related to the required mechanisms for the detection and diagnosis of the faults, and the recovery of the system from their effects. Such fault tolerance mechanisms always impose area, power and performance overheads. Straightforward over-protection of the system with inaccurate knowledge of the effect of the faults can easily make the costs of protection against the hardware faults excessive. For example, typical memory error detection and correction techniques can have a cost (in terms of added memory capacity) which ranges from 1% to 125% depending on the detection and correction capabilities of each technique [18]. Clearly, the selection of the most appropriate protection techniques depends on the required reliability levels and studies of its inherent resiliency to hardware faults that take place at the early design phases.

Section 2.1 is mainly focused on the background concerning the early reliability estimation of microarchitectural structures to transient faults by presenting the metrics, the fault effect classification, the tools and the methodologies that are used to evaluate reliability at the microarchitecture level.

2.1.1 Metrics used for reliability assessments

Firstly, we present all the metrics that are used to measure the reliability (or its reciprocal – the vulnerability) of an individual hardware structure and of the entire system as summarized in [40]. Note that in the formulas presented in this section with the term structures, we define the array-based structures with a particular number of bits and not the logic structures.

Soft error rate (SER) is the rate at which a device or system encounters or is predicted to encounter soft errors. The SER is affected by: (i) the technology model (e.g. 22nm Bulk Planar, 22nm SOI Planar, 20nm Bulk FinFET, etc.), (ii) the voltage, (iii) the temperature, and (iv) the location (latitude, longitude and altitude) that determine the relative neutron fluxes [41].

Time to Failure (TTF) expresses the soft error rate, even though the term TTF refers specifically to failures. As the name suggests, TTF is the time to a fault or an error to occur. For example, if an error occurs after 3 years of operation, then the TTF of that system for that instance is 3 years. Similarly, MTTF (Mean Time to Failure) expresses the mean time elapsed between two faults or errors. Thus, if a system gets an error every 3 years, then that system’s MTTF is 3 years.

The MTTF of various structures comprising a system can be combined to obtain the MTTF of the whole system. For example, if a system is composed of two structures, each with an MTTF of 6 years, then the MTTF of the whole system is:

$$MTTF_{system} = \frac{1}{\frac{1}{MTTF_{structure1}} + \frac{1}{MTTF_{structure2}}} = \frac{1}{\frac{1}{6}+\frac{1}{6}} = 3 \qquad (1)$$

More generally,

$$MTTF_{system} = \frac{1}{\sum_{i=1}^{n} \frac{1}{MTTF_i}} \qquad (2)$$

Apart from the term MTTF, reliability engineers often prefer the term *failure in time* (FIT), which is additive. One FIT represents an error in a billion (10^9) hours of operation. Thus, if a system is composed of two hardware structures, each having an error rate of 10 FIT, then the system has a total error rate of 20 FIT. The summation assumes that the errors in each structure are independent. The error rate of a system is often referred to as its FIT rate. Thus, the FIT rate equation of a system is:

$$FIT\ rate_{system} = \sum_{i=1}^{n} FIT\ rate_i \qquad (3)$$

FIT rate and MTTF are inversely related as described by the following equation:

$$MTTF\ in\ years = \frac{10^9}{\text{FIT rate} \times 24 \text{ hours} \times 365 \text{ days}} \qquad (4)$$

Consequently, one FIT is almost equal with 114K years MTTF or equivalently, one year of MTTF is almost equal to 114K FIT. The FIT of an individual hardware structure is calculated according to the following formula:

$$\text{FIT}_{\text{structure}} = \text{FIT}_{\text{Bit}} \times \text{AVF}_{\text{structure}} \times \#\text{Bits}_{\text{structure}} \qquad (5)$$

where,

- FIT_{Bit}: It represents the raw FIT rate per bit (FIT/bit) and it depends on the technology model, the voltage, the temperature, and the geographical location [41].
- $\text{AVF}_{\text{structure}}$: The Architectural Vulnerability Factor (AVF) [40] represents the probability that a soft error in a bit of the structure affects the program execution. It takes values in the interval [0, 1] and depends on the microarchitecture (Hardware Vulnerability Factor – HVF) and the software (Program Vulnerability

Factor – PVF) that represent the vulnerability that comes from the hardware and the software level, respectively [42] [43].

For the case of estimating the AVF using fault injection approaches (actual injection of the fault in the hardware structures), the AVF is the ratio of the injections that affect the program execution over the total population of injections:

$$AVF_{\text{uArch Fault Injection}} = \frac{\text{\#Injections that affect program output}}{\text{\#Total Injections}} \qquad (6)$$

For example, assume that a fault injection campaign of 1000 injections in total takes place and that 200 of these injections lead to a program corruption. In this case, the AVF is equal to 0.2 (or 20%). The way that a transient fault can affect the program execution is described in detail in Section 2.1.2.

- $\#Bits_{structure}$: It represents the size of the hardware structure in bits.

In Table 1, we show an informative example that calculates the reliability of an entire CPU that consists of six hardware structures (first column) of different sizes (third column), which are fabricated on different technologies (second column). After estimating the AVF for all these structures (fourth column), we calculate the FIT for each of these structures (last column) according to equation (5) and the FIT of the entire CPU that is the sum of the FIT of each individual structure according to equation (3) and is equal to 38.88 FIT in this example.

Table 1: Example of estimating reliability of an entire CPU in FIT.

Structure	Raw FIT/bit	Size (bits)	$AVF_{structure}$	$FIT_{structure}$
A	0.001	1K	0.08	$FIT_A = 0.08$
B	0.001	64K	0.12	$FIT_B = 7.68$
C	0.001	32K	0.02	$FIT_C = 0.64$
D	0.001	8K	0.17	$FIT_D = 1.36$
E	0.001	4K	0.24	$FIT_E = 0.96$
F	0.001	256K	0.11	$FIT_F = 28.16$
				$FIT_{CPU} = 38.88$

2.1.2 Generalized transient fault effect classification

After the occurrence of a transient fault, it is unknown if it is going to affect the program execution. For example, the fault may not eventually be read, or can be overwritten by another access on the same hardware entry, or can be even detected and corrected by a protection mechanism (if one is implemented on the system). All these cases are formalized in the flowchart that is presented in Figure 9 that presents the generalized concept of the transient fault effect classification that is used in the reliability studies [40].

If the faulty bit is not read, then it cannot cause an error and therefore is a benign fault (case 1 in the figure). However, if the faulty bit is read, then one needs to ask whether the bit has error protection. If the bit has error detection and correction (e.g., like ECC), then the fault is corrected, causing no user-visible error (case 2). If the bit has no error protection (neither error detection nor correction) then one needs to ask whether the bit flip affected the program outcome. If the answer is no, then the bit does not matter and that leads to case 3; again a benign fault. However, if the bit flip does affect the program outcome, then it causes what is known as an SDC (Silent Data Corruption) event (case 4). Now, if the bit only has error detection (e.g., parity bit without the ability to recover from an error), then it prevents data corruption but can still cause the program to crash. Then, irrespective of whether the bit matters or not, the program will be usually halted and crashed as soon as the error is detected. Such error detection events, typically visible to the user, are called DUE (Detected Unrecoverable Errors). If an error is declared to be a DUE, it cannot lead to an SDC (since SDC implies no detection – i.e. silence). Thus, the definition of DUE has the implicit notion of a fail-stop system.

Figure 9 shows that DUE events can be further broken down into false and true DUE events. False DUE events (case 5) are those DUE that could have been avoided if there was no error detection mechanism to begin with. For example, certain bits of a wrong-path instruction may not cause an error. In the absence of an error detection mechanism, a flip in such a bit would have gone unnoticed and would not have created any user-visible error. However, because the error detection mechanism detects the error and possibly reports it, the program or the system may be unnecessarily brought down. A bit flip that matters (meaning that actually affects the output of the program) and is detected by the system is a true DUE event (case 6). Protecting a bit with an error detection mechanism moves category 3 to 5 and 4 to 6.

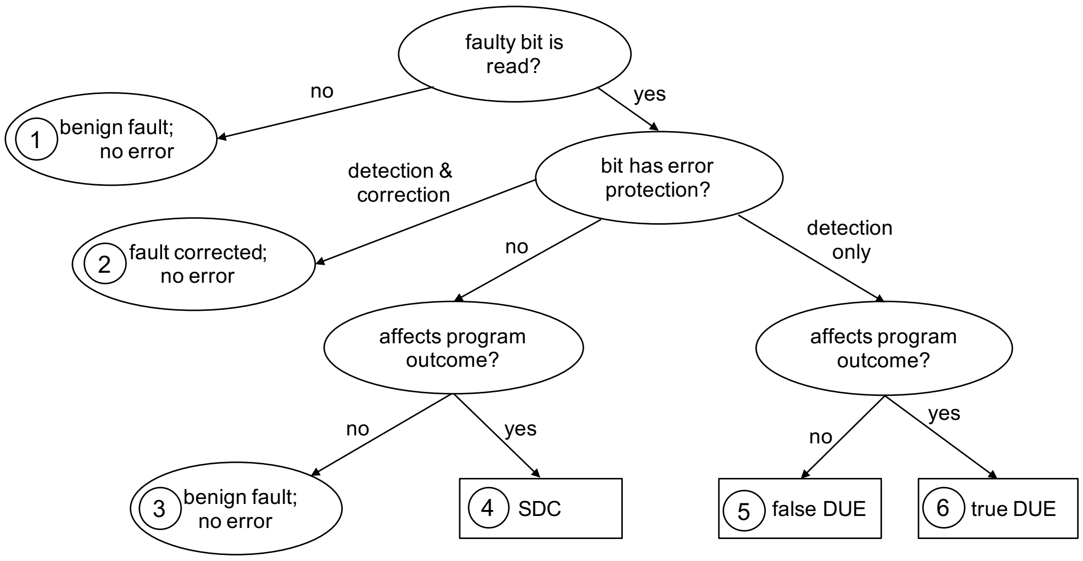


Figure 9: Generalized concept of transient fault effect classification that is used in reliability studies.

2.1.3 Tools used for early reliability assessments

Tolerance mechanisms against any fault model must be decided as early as possible to avoid costly re-design cycles for the 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 important parameters are unknown: hardware components sizes, microarchitectural policies, and the workloads.

It is widely recognized that microarchitecture (or performance) simulators, apart from their importance for performance studies, offer an opportunity for an effective combination of *early*, *accurate* and *fast* reliability estimations, because:

1. They are available in early design phases.

2. They are significantly faster than simulators at more detailed levels of abstraction (RTL, gate-level) [44] and thus allow studies using large, realistic and complex workloads.
3. They accurately model important array-based microarchitecture components: storage arrays which occupy the majority of a chip's area (e.g. on-chip caches, register files, buffers, queues) and thus largely determine the vulnerability of the entire processor to faults.
4. They accurately resemble the behavior of the entire system stack, starting from the microarchitecture level until the higher levels of the operating system- and the application-level. This gives the opportunity to the reliability engineers to analyze the propagation of the faults from the hardware level throughout the entire system stack.

2.1.4 Methods used for reliability assessments

There are four popular approaches to estimate the reliability of hardware components: (i) *RTL injection* [45] [46] [47], (ii) *microarchitecture level injection* [29] [30] [31] [48] [49], (iii) *ACE (Architecturally Correct Execution) analysis* [50] [51] [52] [53] and (iv) *probabilistic models* [54] [55] [56] [57].

RTL injection allows very accurate studies of the fault effects in all hardware structures but these studies are performed too late in the design cycle to facilitate effective decision-making for error protection. Moreover, RTL injection has excessively low simulation throughput (2 to 4 orders of magnitude lower than the throughput of the microarchitectural simulation) [44] which prevents detailed reliability evaluation of components with statistically significant number of injections and large workloads. Finally, RTL design usually is not available at the very early stages of the design cycle.

Microarchitecture level injection is less detailed than RTL injection concerning the logic components but it is very accurate in the implementation of the array-based structures (caches, memories, queues, latches, etc.) that occupy the majority of chip's area. Moreover, it is used for accurate full system studies of fault effects in early design stages, while it is orders of magnitude faster than RTL injection [44].

ACE analysis and *probabilistic models* are significantly faster than the two injection methods because they require only a single or few fault-free runs to report reliability estimations. They provide a very useful but conservative lower bound of the reliability (upper bound of the vulnerability) of hardware components [47] [58] [59]. In particular, [58] reports 7X and [47] reports 3X over-estimation of the AVF that ACE analysis provides compared to fault injection. As an example, [60] reports about 30% AVF for the physical integer register file of the out-of-order Alpha 21264 microprocessor with 80 registers using ACE analysis; however, [32] reports that a comprehensive fault injection campaign of high statistical significance (60,000 transient faults) targeting the same structure for the same benchmarks on the out-of-order x86-64 microarchitecture measures only 2.56%, 4.81%, and 8.92% AVF for 256, 128 and 64 physical registers, respectively. Consequently, the AVF for 80 registers provided by the injection-based measurement will be about 6%, which shows that ACE analysis leads to a 5X over-estimation. Finally, ACE analysis is not suitable to evaluate fault tolerant mechanisms that are based on soft error symptoms, in contrast to injection-based techniques [47] [61]. Despite its disadvantages, ACE analysis merit in early reliability assessments is indisputable because it gives the opportunity to estimate the upper bound of vulnerability for different design options (component sizes, policies, etc.) in very short time.

Table 2 illustrates a qualitative comparison among the methods that are used for reliability evaluation in terms of simulation time, fault model accuracy, estimation accuracy, implementation complexity and availability in the early design phases. ACE analysis and probabilistic models lead to conservative lower bound of reliability. Fault injection in the RTL is very accurate but suffers from low simulation throughput, while RTL designs are not available in the early design phases. On the other hand, microarchitecture level fault injection is very accurate with relative high simulation throughput, while performance models exist early in the design phase and microarchitecture simulators provide many benefits when they are used in reliability studies as was described in Section 2.1.3. Finally, fault injection methodology is less complex than: (a) the ACE analysis that requires a lot of effort to make several modifications in the simulator to achieve better accuracy, and (b) the probabilistic models that need a more complex mathematical background. To conclude, fault injection method at the microarchitecture level is the simplest and most suitable approach for *accurate*, *fast* and *early* reliability assessments.

Table 2: Qualitative comparison among the methods used for reliability evaluation.

	RT-Level injection	Microarchitecture level injection	ACE analysis	Probabilistic models
Simulation Time	High	Medium	Low	Low
Fault Model Accuracy	High	High	None	None
Estimation Accuracy	High	High	Low	Low
Complexity	Low	Low	High	High
Availability in the early design phases	No	Yes	Yes	Yes

2.1.5 Statistical fault injection at the microarchitecture level

The most accurate approach to evaluate the reliability of a hardware structure at the microarchitecture level is to run the *exhaustive* fault injection campaign that consists of all flips for every bit of a hardware structure and for every program execution cycle. This should be repeated multiple times for different hardware structures and different programs to ensure a safe final reliability assessment for the entire microprocessor by using a large representative set of programs and input datasets. This massive repetition of fault injection runs during the exhaustive fault injection in conjunction with the simulation throughput of a detailed simulation run at the microarchitecture level (10^5 cycles/sec [15]) make the execution of a complete exhaustive fault injection campaign infeasible. For instance, according to [32] the exhaustive fault injection to evaluate the reliability at the microarchitecture level of only three hardware structures (a L1 data cache of 32KB, a store queue of 16 entries and a physical register file of 64 registers) for a single benchmark of 1 billion cycles takes about $3x10^9$ years, assuming that a single hardware thread is used.

Thus, statistical fault sampling is unavoidable in order to make the reliability estimation at the microarchitecture level feasible without compromising accuracy [32]. For all the studies concerning the fault injection of transient faults at the microarchitecture level that are presented in this thesis, we used the statistical fault sampling formula that was described in [62]. This formula is presented below:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \quad (7)$$

where,

- **n**: The total population of faults that are needed after the fault sampling to provide the final reliability assessment of a fault injection campaign
- **N**: The initial (exhaustive) population of the faults that comes from the product of the *workload duration* in cycles (**d**) and the *size of the hardware structure* in bits (**s**). Thus,

$$N = d \times s \quad (8)$$

- **e**: The predefined error margin of the final estimation
- **t**: The predefined confidence level of the final estimation
- **p**: The estimated proportion *p* of individuals in the population having a given characteristic (e.g. the estimated probability of faults resulting in a failure). This parameter defines the standard error and basically corresponds to an estimate of the true value being searched (e.g. percentage of errors resulting in a failure). Since this value is a priori unknown (but between 0 and 1), a conservative approach is to use the value that will maximize the sample size. In other words, the sample size will be chosen so that it will be sufficient to ensure the expected margin of error with the expected confidence level, no matter the actual value of the proportion. It has been demonstrated that this is achieved for p=0.5; it is therefore sufficient to use this value in all cases [62]. If the expected proportion is very small or very large, refining this estimate would lead to a reduced sample size for a given margin of error but the goal is to avoid any a priori assumption on the results. Instead, after the fault injection campaign to estimate the final reliability, the refinement targets the error margin, while all the other parameters of the formula remain the same; this finally leads to the real error margin of the estimation.

  For instance, assuming that two fault injection campaigns take place to evaluate the reliability of a 64-entry integer register file of 64-bit registers (the size of the structure is 4096 bits in total) running two different benchmarks of 1 million cycles. The initial fault list population for the case of the *exhaustive* fault list injection campaign is equal to $4096\text{x}10^6$ for both cases; thus, the statistical fault sampling is unavoidable. If the statistical significance of the sampling is predefined at 3% error margin and 95% confidence level, then the total population of faults will be 1067 faults according to [62]. Note that, as the final reliability estimation is a priori unknown, the *p* parameter must be predefined at 0.5 to ensure the expected margin of error with the expected confidence level. If the final reliability estimations of the campaigns after injecting the 1067 faults are different than 50% (that was described to be the expected by defining *p* parameter to 0.50), for example 30% vulnerability estimation for the first benchmark and only 5% for the second one, then we need to refine the formula of [62] to get the real error margin of the final estimations. After this refinement, the final error margin will be 2.75% and 1.31% (instead of 3% that was initially defined) for the first and the second campaigns, respectively. The confidence level for both the two final estimations remains stable at 95%.

To achieve high statistical significance, the initial fault list should consist of tens or hundreds of thousands of faults. For instance, an injection campaign targeting a 256-entry integer register file of 64-bit registers with error margin 2.88%, confidence level 99% and 100 million cycles of program execution time, requires 2000 fault injection runs. If a higher statistical significance is needed (i.e. 0.63% error margin and 99.8% confidence level), the total number of injection runs explodes to 60,000 (an unacceptably large number of injections even for relatively short benchmarks).

To better understand these sizes, we assume a single fault injection campaign of 60,000 faults that runs the *sha* benchmark as workload under test (a common benchmark used in reliability studies). A single fault free run of this benchmark lasts about 66 seconds. Thus, a campaign of 60,000 injections needs about 46 days to complete using a single thread or about 5.75 days in a machine of 8 threads. Note that for the reliability assessment of the entire CPU, multiple fault injection campaigns should be launched to evaluate different benchmarks and microarchitectural structures for different sizes and policies etc.

According to [62], for estimations of high statistical significance the confidence level and the error margin dominate in the calculation of the initial fault list population instead of the duration of the benchmark or the size of the hardware structure. For example, for a fault injection campaign that targets a hardware structure of 32K bits and runs a benchmark of 1 billion cycles, Figure 10 presents the increase in the population of faults when the error margin decreases and the confidence level remains stable at 99% and 99.8%, respectively. For the same example, Figure 11 illustrates the increase of the fault population, when the confidence level increases, while the error margin remains stable at 3% and 1%, respectively. From these three figures, it is obvious that the population of faults explodes when the statistical significance increases (when the confidence level increases and/or the error margin decreases).

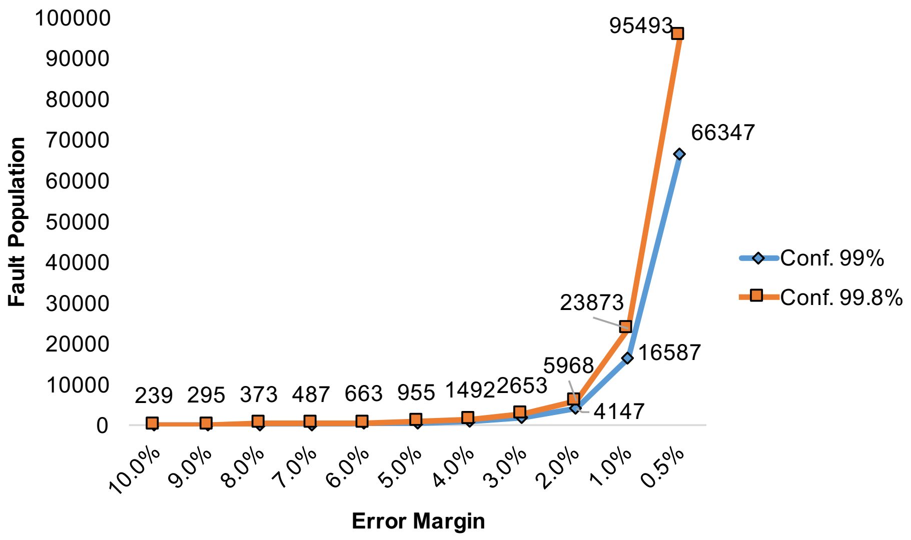


Figure 10: Fault Population vs. Error Margin for Confidence Level 99% and 99.8%.

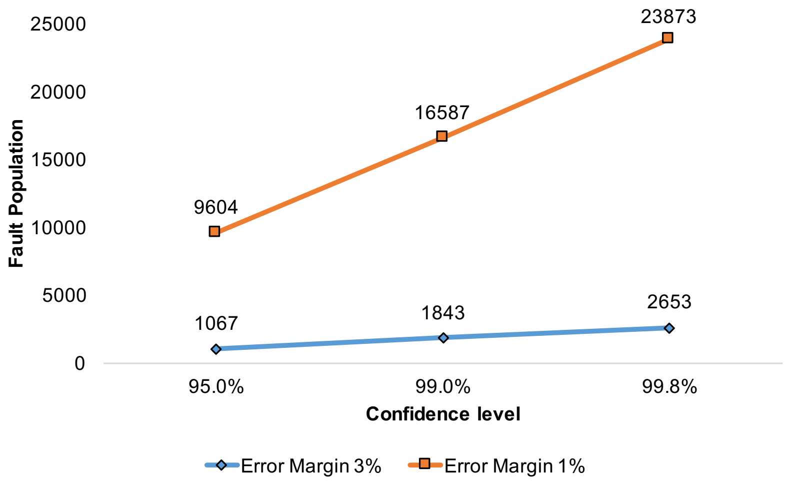


Figure 11: Fault Population vs. Confidence Level for Error Margin 3% and 1%.

To conclude, *exhaustive* fault injection at the microarchitecture level is infeasible for a large combination of benchmarks, input datasets and hardware structures. So, reliability engineers resort to statistical fault sampling to overcome this issue. However, the reliability estimations of high statistical significance to ensure the final accuracy still require very time consuming fault injection campaigns that last from days to months of execution, in particular for long benchmarks. This thesis proposes techniques to reduce the execution time of these campaigns without affecting the final accuracy.

2.2 MaFIN tool for early microarchitecture level reliability assessments

Accurate identification of the vulnerabilities of a microprocessor product, early in design time, assists designers to carefully plan for reliability enhancements with low cost and high power efficiency. On the contrary, inaccurate reliability estimation often results on over-designed microprocessors and negatively impacts time-to-market (TTM) and product costs. Thus, computer architects require tools for fast and accurate assessment of a component's reliability, so that they can decide for high level architectural trade-offs early in the design process without resorting to worst-case and over-protecting approaches.

As was previously presented in Sections 2.1.3 and 2.1.4, fault injection at the microarchitecture level using performance simulators is a very efficient method for *fast*, *early* and *accurate* reliability assessments. Unfortunately, there are not public available tools to estimate reliability at the microarchitecture level using fault injection. Thus, we have developed a tool called MaFIN (MARSSx86 Fault INjector) [27] [29] that is based on a state-of-the-art x86-64 performance simulator (called MARSSx86 [14]).

MARSSx86 is widely used for performance measurements [63] [64] [65] and utilizes PTLsim [13] to simulate the internal details of an x86-64 microprocessor model. PTLsim has been used for many reliability measurements [49] [57] [66], as well as silicon validation [67]. MARSSx86 is a full system, cycle-accurate simulator capable of simulating a multicore processor with a detailed implementation of the front-end and the back-end pipeline stages of a modern x86-64 architecture. In addition, MARSSx86 simulates the cache hierarchy, which we extended with the data arrays (to allow realistic fault injections at all different cache levels L1, L2, L3), and implements several cache coherency protocols. To provide full system capabilities MARSSx86 is coupled with QEMU emulator [68].

Before selecting MARSSx86, we have considered a number of publicly available full system simulators. A recent study [69] on the sources of modeling errors in full system simulators summarizes the publicly available tools and their advantages: Flexus [70], Gem5 [15], GEMS [71], MARSSx86 [14], OVPsim [72], PTLsim [13], Simics [73]. Among these full system simulators MARSSx86 [14] and Gem5 [15] are: cycle-accurate (thus can allow per cycle granularity of fault injections at any modeled hardware component), publicly available, and regularly maintained today by their developers. By themselves, these properties can justify selecting MARSSx86 and Gem5 the most suitable for our reliability studies.

As a first choice, we selected MARSSx86 as the kernel of our framework due to the following reasons: (a) it accurately simulates an x86-64 microprocessor model, (b) its full system simulation operation provides us with the capability to trace the propagation of a low-level hardware fault, till its manifestation on the operating system- or on application-level output, (c) the x86 functional model of MARSSx86 is more accurate than other publicly available simulators and its memory system better models real systems [65], (d) it uses QEMU that makes its models to resemble the functionality of a real full system as close as possible compared to Gem5, and (e) its predecessor (PTLsim) was previously used in many reliability and performance studies [49] [66].

In the next subsections, we will describe in detail the features and the implementation of MaFIN (Section 2.2.1), as well as several reliability studies that were launched during this thesis and used MaFIN as experimental tool (Section 2.2.2 to Section 2.2.4).

2.2.1 MaFIN features and implementation

In this section, we discuss in detail the MaFIN fault injection framework that was developed during this thesis. Firstly, we will present the fault models and the fault effect classification that are supported by MaFIN. Next, we will present in detail the implementation of the fault injection framework.

MaFIN models exactly the three different fault types on microarchitectural array components: transient, intermittent and permanent faults as well as their combinations. These three types of fault models allow a wide analysis of the effect of different factors that affect reliability: fabrication defects, environmental conditions, early-life failures, device degradation and voltage scaling. Table 3 describes the three basic single bit fault models.

Table 3: Fault models supported by MaFIN.

Fault model	Description of fault model
transient	a storage element’s bit value is flipped in a clock cycle of the program execution; the bit position and the clock cycle can be set arbitrarily (randomly or directed). Note that, MaFIN does not consider transient faults in combinational logic because microarchitecture simulators do not model such logic accurately; however, their effects would propagate to storage elements and thus can be also implicitly studied with our tools
intermittent	a storage element’s bit value is set to ‘0’ or to ‘1’ starting at a clock cycle and for an arbitrary number of clock cycles; the bit position, the start time and the duration of the fault can be set arbitrarily (randomly or directed)
permanent	a storage element’s bit value is permanently set to ‘0’ or to ‘1’; the bit position can be set arbitrarily (randomly or directed)

Moreover, MaFIN supports fault injection experiments for multiple faults in many different combinations to match both the temporal and the spatial behavior of faults in hardware structures. Such combinations can include injection of (a) multiple faults of any type and any duration in a single structure, (b) multiple faults on different structures. Obviously, the type, the multiplicity and the locations of the faults used in a certain injection campaign depend on the study that a user of MaFIN wishes to perform.

MaFIN injector classifies the outcomes of each fault injection simulation based on the impact of the fault on the simulated system. The fault classification is fully configurable and a user of the injector can modify the classes of the fault effects by changing the parser of the injection logging information.

For all the reliability studies of this thesis, we modified the flowchart that was presented in Figure 9 to analyze the results of the fault injection campaigns that are executed on a microarchitectural simulator. Thus, we concluded to six categories of fault effects (Masked, SDC, DUE, Timeout, Crash, Assert) that represent typical classes (and corresponding terminology) used in the reliability literature. The definition for each of these six classes is presented in Table 4. Finally, the decision tree that was used for all the reliability estimation studies of this thesis in order to classify an injection run to one of the six aforementioned categories is presented in Figure 12.

Note that after parsing the results of each fault injection campaign, the sum of the non-masked categories (SDC, DUE, Timeout, Crash, Assert) represents the AVF that is the final vulnerability estimation of the fault injection campaign (or equivalently the 100% – Masked category). For example, if a fault injection campaign led to 64% Masked, 13% SDC, 7% DUE, 11% Timeout, 2% Crash and 3% Assert, then the AVF estimation is equal to 36% (the sum of the non-masked categories).

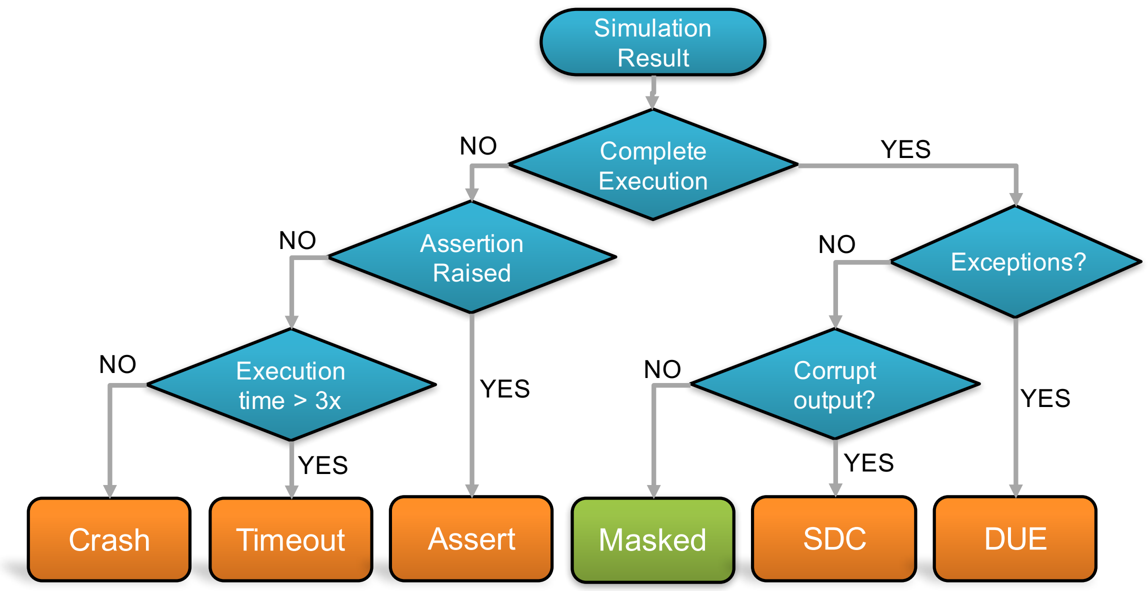


Figure 12: Flowchart of fault effect classification used in our reliability estimation studies.

Table 4: Fault effect classification.

Fault effect	Description of fault effect
Masked	Fault injection runs in which the fault does not affect the execution of the application (which is executed to its end). The result of an injection with a masked fault is identical to that of a “golden” (fault-free) simulation. This practically means that both the output of the application and the population of exceptions generated during execution of the fault injection run are identical to the “golden” run.
Silent Data Corruption (SDC)	Fault injection runs for which the final output of the program that is written to an output file is corrupted (differs from the output of the fault-free execution) and no other indication of the fault has been recorded (an abnormal event such as an exception, etc.).
Detected Unrecoverable Error (DUE)	Includes cases in which the simulated process completes successfully, but with indications of errors. The baseline microprocessor models do not include any error detection or protection mechanisms and therefore, the only indication of an error is the raising of ISA exceptions. Typically, reliability reports in the literature divide DUEs in two sub-categories: false DUE (the output is correct despite the error indication) and true DUE (output is corrupted).
Timeout	Includes all of the cases that lead to either a Deadlock or a Livelock. A Deadlock describes the condition in which the program flow has been trapped (due to the injected fault) and cannot commit any further instructions. A Livelock, on the other hand, describes a situation where the program flow has been redirected and continues the execution of instructions on random code areas (again due to the fault). In order to monitor these cases, a configurable execution timeout limit is used. In our experimental results, the limit is three times the fault-free execution time of each benchmark.
Crash	Includes any case that results in an unrecoverable situation and stops the simulated program. Crashes involve all three levels of the simulation, including a process crash, where the simulated program was abnormally terminated, a system crash, where the simulated full-system was unable to recover (typical cases of kernel panic) as well as a simulator crash, where the simulator process itself was abnormally terminated.
Assert	Includes all cases where the simulator reached, due to injected fault, a (high level) condition which was unable to handle and an assertion was raised stopping the simulation.

MaFIN injector is built on three main modules which form the backbone of any fault injection campaign run on it. Figure 13 visualizes the flow of operation of MaFIN injector, while the three main modules are defined below:

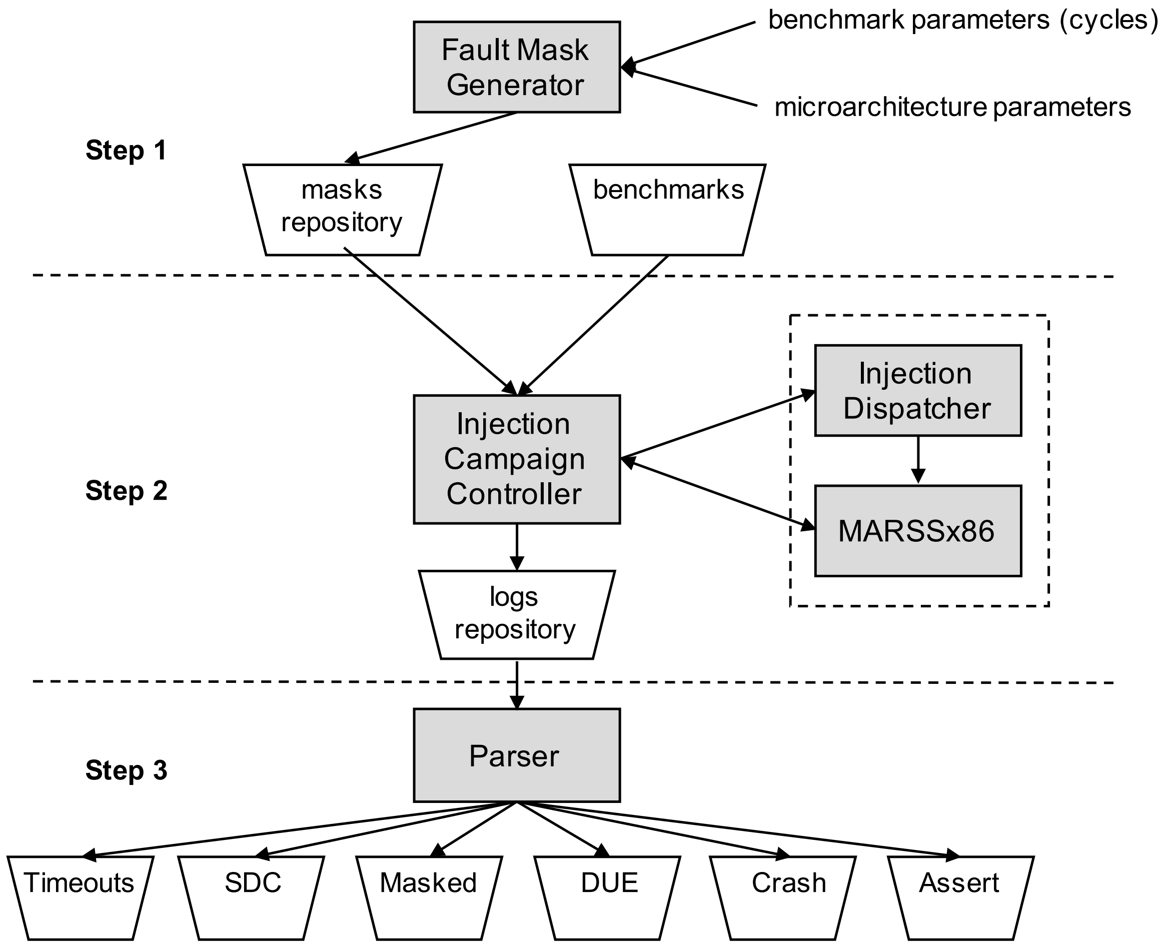


Figure 13: MaFIN injection framework.

- **1st module:** In the first step, the Fault Mask Generator module produces the fault masks that are used during the injection campaign. This is a one-step process for each fault injection campaign that corresponds to a specific combination of hardware structure and benchmark. The Fault Mask Generator can produce (by user defined parameters) a random set of fault masks for any type of fault (transient, intermittent, permanent) for the entire simulation time of the benchmark.

  A fault mask contains information about: (i) the processor core where the fault is going to be injected (can be used in a multicore architecture), (ii) the microarchitecture structure on which the fault will be injected, (iii) the exact bit position of the injection, (iv) the exact simulation cycle or exact instruction on which injection happens (for transient or intermittent), (v) the type of fault, and finally (vi) the population of faults (single or multiple). All the generated fault masks are stored in a “masks repository” from which the Injection Campaign Controller picks fault masks to apply.

- **2nd module:** Provided the “mask repository”, the actual fault injection campaign can begin. The Injection Campaign Controller reads the masks from the repository and sends injection requests to the Injector Dispatcher which is the module that directly communicates with the MARSSx86 simulator. The interface between the Injection Campaign Controller and the individual Injection Dispatcher contains the transfer of user defined parameters concerning the injection to the microarchitectural simulators and the transfer of the results of the fault injection experiments from the microarchitectural simulator back to the

Injection Campaign Controller. The last task of the Injection Campaign Controller is to store the results of the injection in a “logs repository” which contains all log files for further processing by the Parser.

- **3rd module:** The third and last step of the fault injection campaign is the processing of the injection results and the generation of the fault effects classification. The processing of the fault injection results is performed by the use of a Parser. The Parser is an easily reconfigurable script that classifies the faults into the six final categories described in Table 4: Masked, SDC, DUE, Timeout, Crash, and Assert. The classification results can be easily modified through small changes of the Parser code according to the user’s needs as the input of Parser for an alternative classification is not changed and is already stored into the log files repository (no new fault injection campaign is required). For example, a more course-grain classification can be used just separating “Masked” from “Non-Masked” behavior. On the other hand, a more fine-grain classification may break down the DUE category in false-DUE and true-DUE (a usual separation in the reliability literature). Moreover, the user could move the results from the Simulator Crash subcategory to the Assert category to group together faulty behaviors attributed to simulator malfunctions due to the injected faults.

Microarchitectural simulators are developed for performance measurements of the simulated model and their main objective is to save simulation time without modeling details that are not necessary for performance assessments. As a result, performance simulators may lack certain functionality necessary to perform accurate fault injection experiments. For example, the functional and the control logic components are not implemented in a way that resembles actual hardware structures. Therefore, injectors like MaFIN focus on reliability studies in hardware structures which are modeled as arrays in a performance simulator and thus the effect of faults on them can be accurately measured. The injection of transient, intermittent or permanent fault on a modeled storage bit of a microarchitectural simulator is largely equivalent to injecting it on the actual hardware.

Unfortunately, some simulators do not model data arrays of caches (and other structures such as queues, buffers); MARSSx86 is such a simulator. It models the control information of cache memories (tags and control bits) but only keeps the actual data and instructions at the main memory model of the simulation. Without the implementation of the actual arrays for the data and the instructions on caches of the different levels, fault injection is not feasible. To address this issue, we enhanced the initial model of MARSSx86 with the data arrays in all cache levels (L1 data and instruction cache, unified L2 cache and L3) to allow realistic fault injections at all different cache levels. This modification of MARSSx86 introduced an approximate ~40% throughput degradation which depends on the memory intensiveness of a program.

The development of MaFIN went through the following major tasks:

- Identification of existing structures; integration of the fault injector on these structures.
- Modification of structures that lack of accuracy to perform a fault injection study (missing bit arrays); integration of the fault injector on these structures.
- Enhancement of the x86 model of MARSSx86 with new components (performance related) to fully resemble a modern design; integration of the fault injector on these new structures. Table 5 summarizes all enhancements made on MARSSx86 for accurate measurements of the reliability of the hardware structures of x86

architectures. Note, that MaFIN supports fault injection to all hardware components of Table 5.

Table 5: MaFIN enhancements and supported components.

Components	MaFIN
Existing	Load/Store Queue Issue Queue Integer Register File FP Register File Caches – Tag Data TLB – Valid, Tag Instr. TLB – Valid, Tag Branch Target Buffer – Uncond. indirect branches
Modified	L1D cache – Data arrays L1I cache – Instruction arrays L2 cache – Data arrays L1I cache – Valid bit L1D cache – Valid bit L2 cache – Valid bit Branch Target Buffer – Uncond. / Cond. direct branches
New	Prefetcher in L1D cache Prefetcher in L1I cache

To conclude, MaFIN tool is an accurate framework for early reliability assessments of x86-64 microarchitectures using fault injection; it supports a large population of array-based hardware structures that occupy the majority of chips' area. For these reasons, MaFIN was extensively used in many reliability related studies that are presented in detail in the following subsections. More specifically, in Section 2.2.2 we evaluate the reliability and performance tradeoffs for major hardware components of an x86-64 microprocessor across several important parameters of their design (size, associativity, write policy, etc.). In Section 2.2.3, we use MaFIN in conjunction with GeFIN tool [30] to evaluate in a differential way: (a) the reliability sensitivity of several microarchitecture structures for the same ISA (x86-64) implemented on two different simulators, and (b) the reliability of workloads and microarchitectures for two popular ISAs (ARM vs. x86-64). Finally, in Section 2.2.4 we present some methods to accelerate the fault injection campaigns using MaFIN as experimental vehicle based on the faults lifetime.

2.2.2 Reliability – Performance tradeoffs assessment study

Early decisions in microprocessor design require a careful consideration of the corresponding performance and reliability implications of transient faults. The size and organization of important on-chip hardware components such as caches, register files and buffers have a direct impact on both the microprocessor resilience to soft errors and the execution time of the applications. There are some critical design decisions for an architect concerning the tradeoff of performance and reliability. For instance, how is performance and reliability affected when the cache size is doubled? Are they going to the same direction or are they diverging (e.g. performance gets better and reliability gets worse)?

In [28], we present a complete fault injection analysis of transient faults which jointly considers the interplay between reliability and performance of several important design parameters on a modern out-of-order x86-64 architecture. To achieve this, we also propose a simple and flexible *fitness function* that measures the aggregate effect of such design changes on the reliability and the performance of the studied workload.

In this case study, we focus on five storage arrays of the microprocessor: the physical register file, the cache memories (split L1 data and L1 instruction cache and the unified L2 cache) and the load/store queue. We use MaFIN to execute statistical fault injection and we modify a single parameter of the baseline microarchitecture (presented in Table 6) at a time, to monitor the impact of individual changes on the reliability of the structure in conjunction with the performance of the application. All the variants of the baseline model used in our experiments are illustrated in Table 7. In the first two columns, we refer to the target structures and their microarchitectural features while in the last column we list the alternative values that each feature had in our experiments. Summarizing all our experiments, for all the components of our study we modify the sizes of the hardware structures and assess the impact on reliability and performance. Also, we study the impact of the different write policies (write back vs. write though) for the first level and second level caches, while for the L1 caches (both instruction and data), we extensively evaluate the impact of different associativity points as well as their behavior in the presence or absence of L1 prefetchers.

We analyze the reliability of each different design point using statistical significant fault injection campaigns on MaFIN. For each fault injection campaign, we ran 2000 fault injection experiments corresponding to 2.88% error margin and 99% confidence level according to [62] as was presented in Section 2.1.5. We classified the fault effects in six categories (Masked, SDC, DUE, Timeout, Crash, Assert) that were defined in Table 4, while the final reliability estimation was measured in FIT (failures in time) for all the cases according to equation (5). Finally, we measured the performance for different design points using the IPC (instructions per cycle) metric.

Table 6: Baseline configuration of study [28].

Structure		Baseline Model
Pipeline		OoO
Issue Queue		32 entries
Reorder Buffer		64 entries
Phys. Int. Reg File		256 registers
L1D cache		32KB, 64B, 4-way, write-back, stride pref.
L1I cache		32KB, 64B, 4-way, write-back, sequential pref.
Unified L2 cache		1MB, 64B, 16-way, write-back, w/o pref.
Unified LSQ		32 entries (16 load queue, 16 store queue)
Branch Predictor		Tournament Predictor
Branch Target Buffer	Direct branches	1K entries, 4-way
	Indirect branches	512 entries, 4-way
RAS		16 entries
Functional Units		2 Integer ALUs, 2 FP ALUs, 4 AGU

Table 7: Experimental setup used in study [28].

Structure	Parameter	Values
Phys. Register File	Number of Registers	64/128/256
L1D cache	Size	16KB/32KB/64KB
	Associativity	1/2/4/8
	Write Policy	WB/WT
	Prefetcher	enabled/disabled
L1I cache	Size	16KB/32KB/64KB
	Associativity	1/2/4/8
	Write Policy	WB/WT
	Prefetcher	enabled/disabled
L2 cache	Write Policy	WB/WT
LSQ	Number of entries	32/64/96

The reliability information along with the performance information of each design point can help a microprocessor designer in making informed decisions about the hardware protection mechanisms required for a particular configuration and workloads [53]. To assist design decisions, we also define a simple yet flexible *fitness function* which describes the combined effect on reliability and performance that certain design parameters have:

$$fitness = a \times \frac{1}{FIT} + (1 - a) \times IPC \tag{9}$$

In equation (9), FIT is the fraction of the failures in time (on average for all benchmarks) that a hardware component with a specific configuration has over the one of the baseline model ($FIT=FIT_{conf}/FIT_{base}$); therefore, FIT >1 means that the studied configuration has a higher FIT rate (smaller reliability) than the baseline configuration. Similarly, IPC is the fault free committed instructions per cycle (on average for all benchmarks) with a specific hardware configuration over the one with the baseline hardware configuration ($IPC=IPC_{conf}/IPC_{base}$); therefore, IPC>1 means that the studied configuration is faster than the baseline. Parameter *a* is designer-defined (taking values from 0 to 1) and represents a wide range of designs that put more emphasis on the reliability or on the performance or balances both. The smaller the value of *a*, the more importance is given to performance (IPC). On the contrary, the larger the value of *a*, the more importance is given to reliability. If *a* equals 0.5, then the same importance is given to both performance and reliability. Consequently, every fitness value of equation (9) represents a design point corresponding to an experimental setup that can be either better (in terms of reliability and performance) than the baseline configuration ($fitness_{conf}$ > $fitness_{base}$) or worse ($fitness_{conf}$ < $fitness_{base}$), where $fitness_{base}$ = 1.00.

Performance and reliability evaluations are workload dependent [43] [53] and a careful selection of benchmarks is vital for the accuracy of a study. For this study, we carried out fault injection campaigns during the execution of 7 benchmarks (*djpeg*, *search*, *corners*, *edges*, *sha*, *qsort*, *smooth*) from MiBench suite. We selected MiBench benchmarks suite for our evaluation because it consists of programs from different application domains that are very similar in their instruction mixes and instruction throughput with SPEC2006 benchmarks [74]. Their shorter execution times compared to SPEC2006 [74] [75] (standard benchmarks for performance studies) make them very suitable for fault injection and reliability studies and for this reason they have been extensively used in such a context [58], [60], [76], [77].

Firstly, we present the reliability estimation results of the analysis of [28] using MaFIN as experimental tool, when we change different microarchitectural features; these features represent different design decisions. In the bars from Figure 14 to Figure 24, we present the fault effect classification on average and per benchmark for each parameter of the hardware components (size in all components, associativity of L1 caches, write policy of all caches and behavior of L1 caches with or without prefetcher). Figure 14 to Figure 17 show the faulty behavior classification for these four structures per benchmark and on average for the 7 benchmarks. In the two L1 caches (Figure 14 and Figure 15), there are some benchmarks with opposite behavior but the average trend shows an increase of the percentage of masked class for larger sizes: the average masked class of L1 Data cache increases by 7 percentile points and the one of L1 Instruction cache increases by 5 percentile points from 16KB to 64KB. In the register file (Figure 16), all benchmarks follow the same trend featuring higher percentage of masked class when the register file contains more registers and the average percentage of masked class of register file increases by 7 percentile points from 64 to 256 registers. The LSQ (Figure 17) features a smaller but still important 2 percentile points increase in the percentage of masked category from 32 to 96 entries.

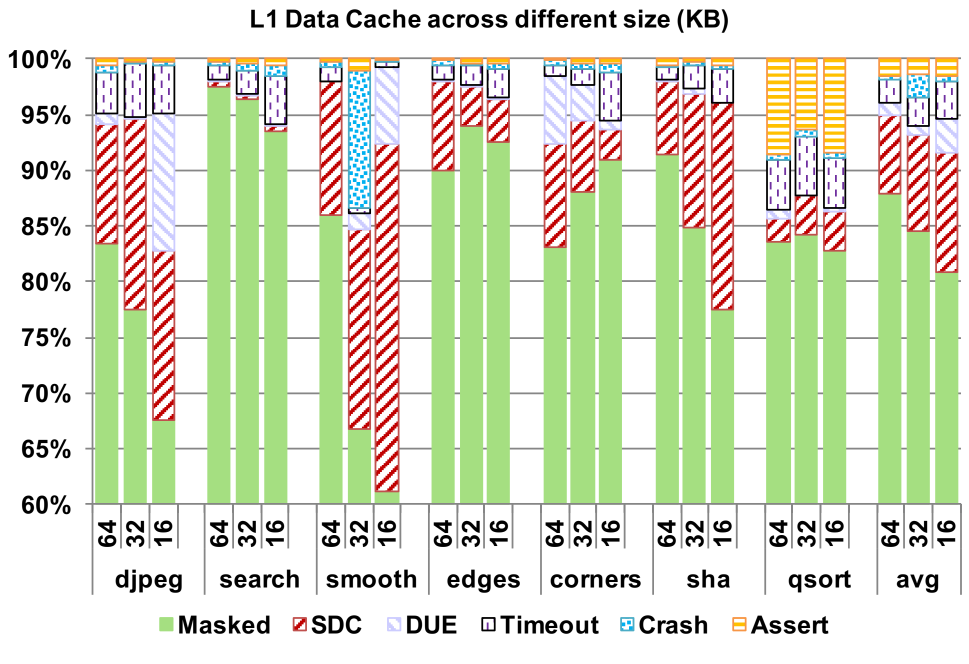


Figure 14: Faults classification in L1 Data cache (sizes).

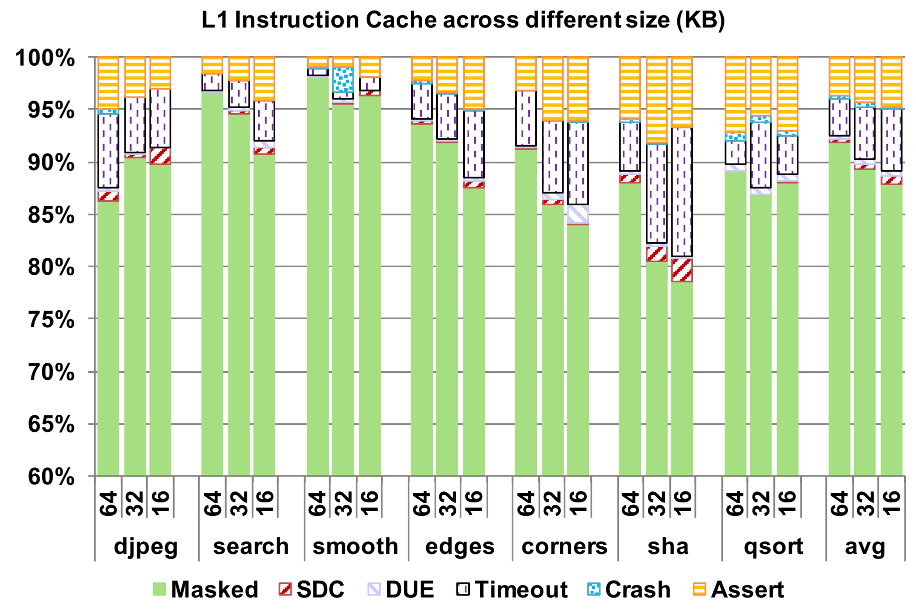


Figure 15: Faults classification in L1 Instruction cache (sizes).

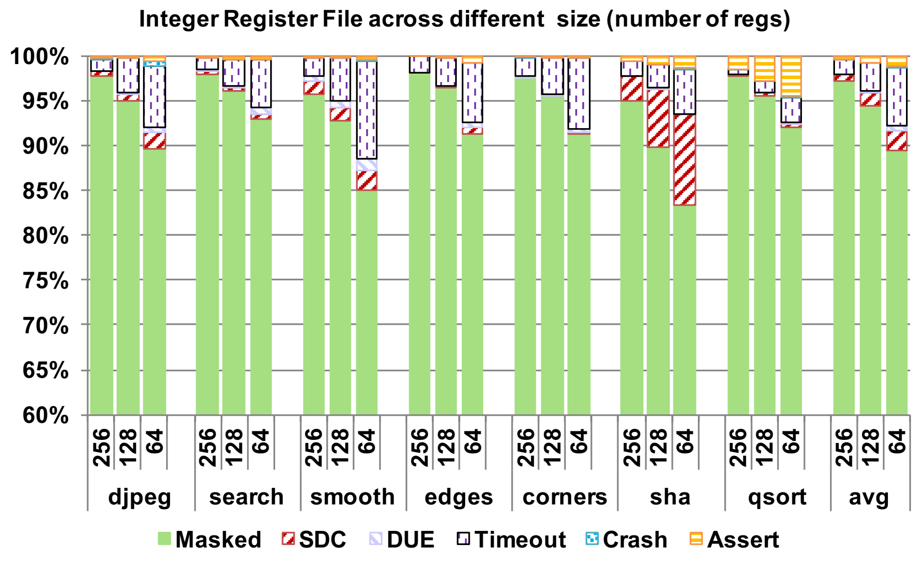


Figure 16: Faults classification in Physical Register File (sizes).

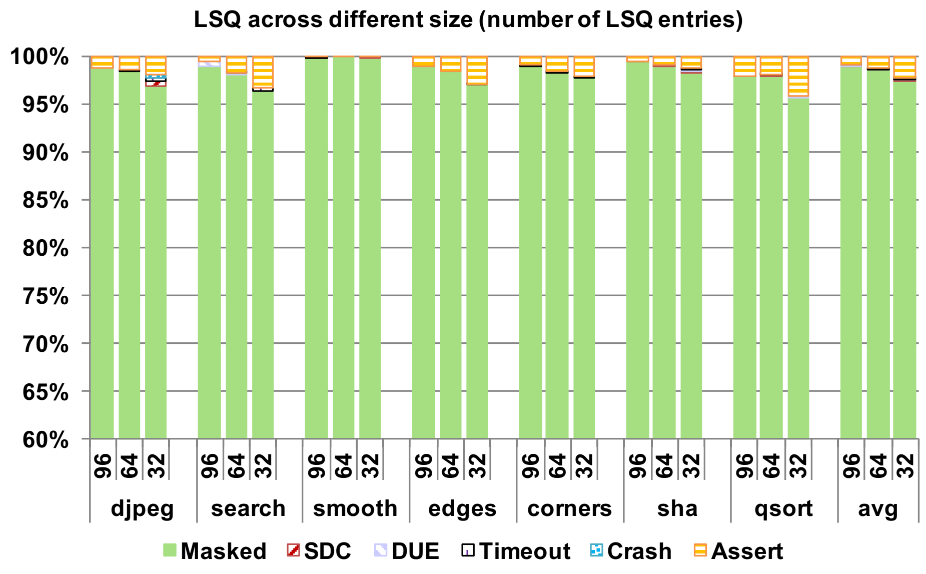


Figure 17: Faults classification in LSQ (sizes).

In the same concept, Figure 18 and Figure 19 present the reliability results when we change the associativity parameter of the L1 Data and L1 Instruction cache, respectively. In Figure 18, the average percentage of masked category of L1 Data cache is almost insensitive to associativity while the percentage of masked category of only two benchmarks (djpeg and smooth) features changes for different associativity. On the other hand in Figure 19, the masked category of L1 Instruction cache increases by 6 percentile points from a direct-mapped to an 8-way set associative cache (for the same size of 32KB).

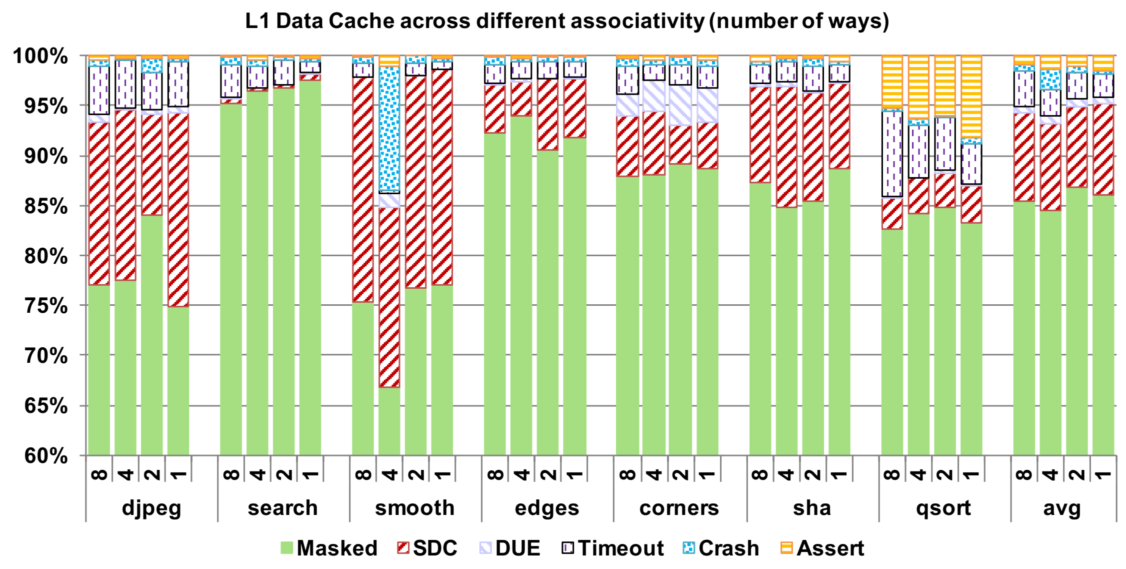


Figure 18: Faults classification in L1 Data cache (associativity).

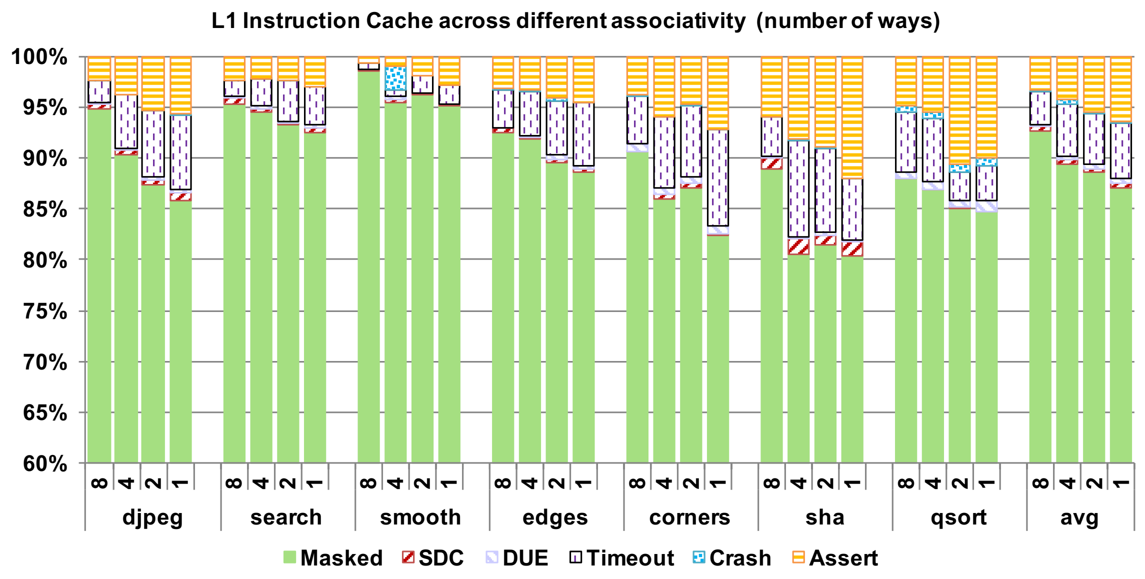


Figure 19: Faults classification in L1 Instruction cache (associativity)

Figure 20 to Figure 22 show the results of all the caches, when we change their policy (write back and write through). In Figure 20, the average percentage of masked category in L1 Data cache increases by 6 percentile points when the write through policy is used instead of the write back. In Figure 21, the L1 Instruction cache features almost the same behavior (equivalent percentage of masked category) for both policies since blocks that are evicted by instruction caches are never dirty and thus cannot propagate the fault to lower levels of memory hierarchy. In Figure 22, the write through unified L2 cache has higher masking probability than the write back L2 cache (about 2 percentile points on average). A faulty cache line in L2 Cache that is exclusively allocated to data (not instruction) may propagate the fault to the lower level of memory hierarchy only if a write back policy is used.

Finally, in Figure 23 the average percentage of masked class of L1 Data cache with prefetcher is very close to the one without prefetcher. In Figure 24, the average percentage of masked class of L1 Instruction cache increases by 5 percentile points when prefetecher is enabled.

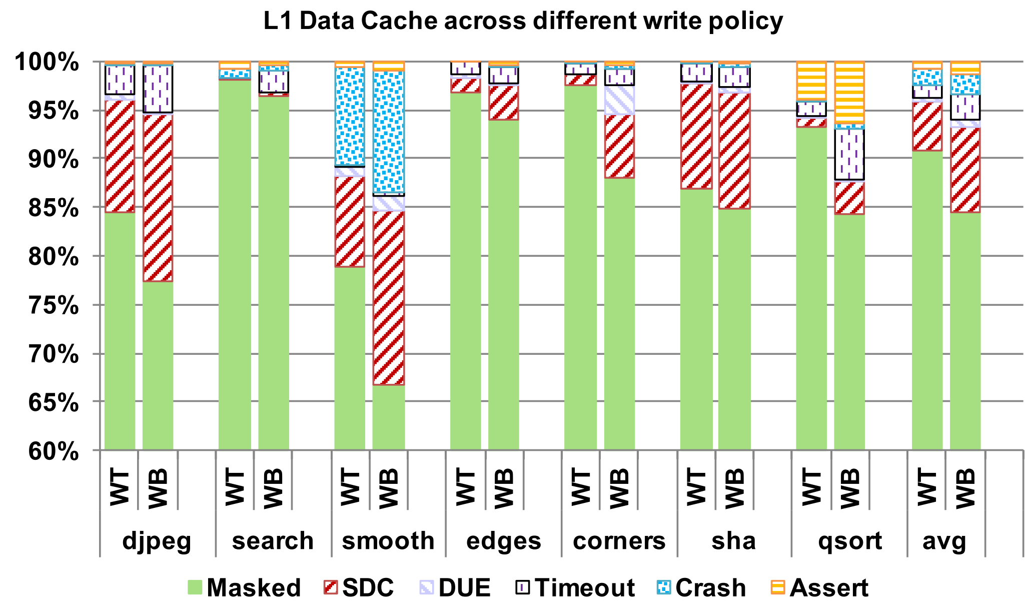


Figure 20: Faults classification in L1 Data cache (write policies).

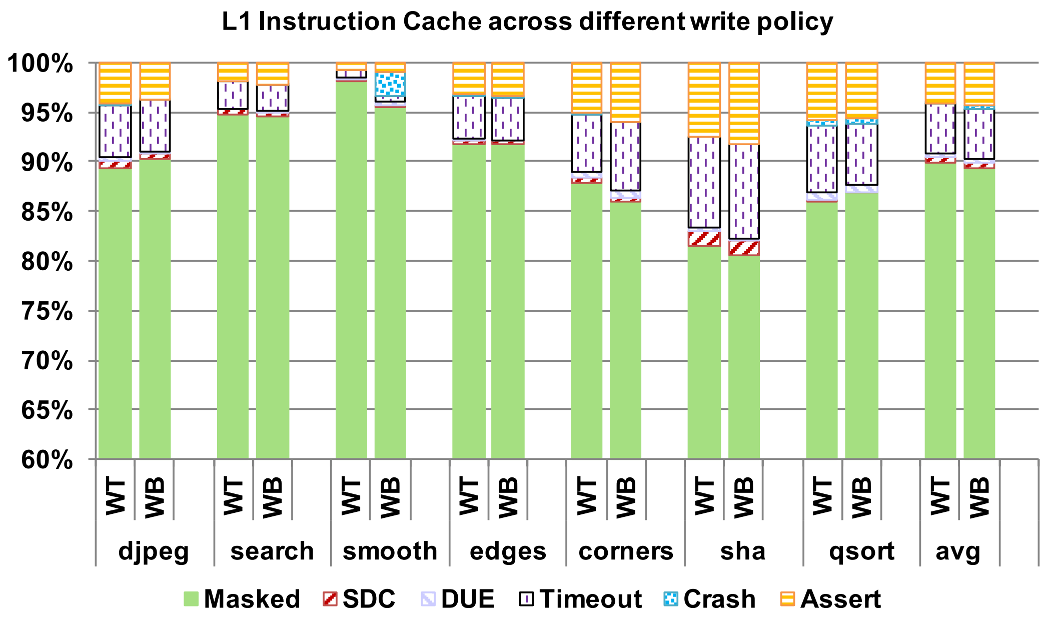


Figure 21: Faults classification in L1 Instruction cache (write policies).

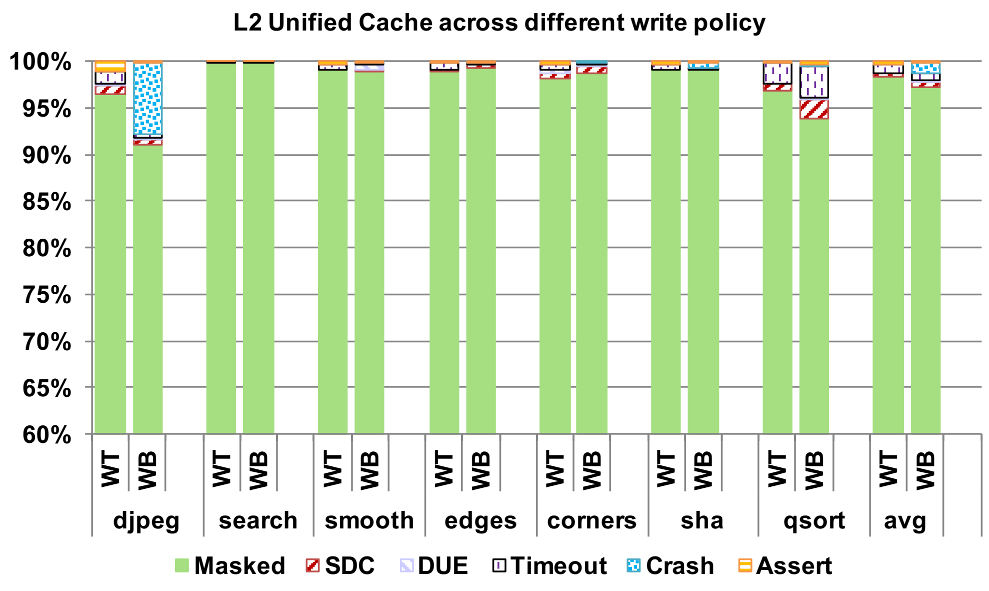


Figure 22: Faults classification in L2 Unified cache (write policies).

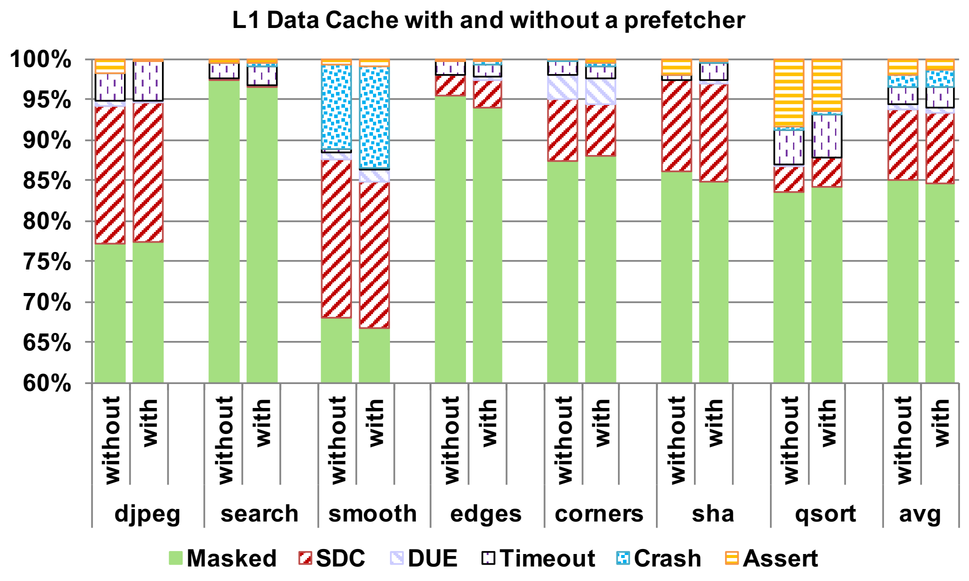


Figure 23: Faults classification in L1 Data cache (prefetcher).

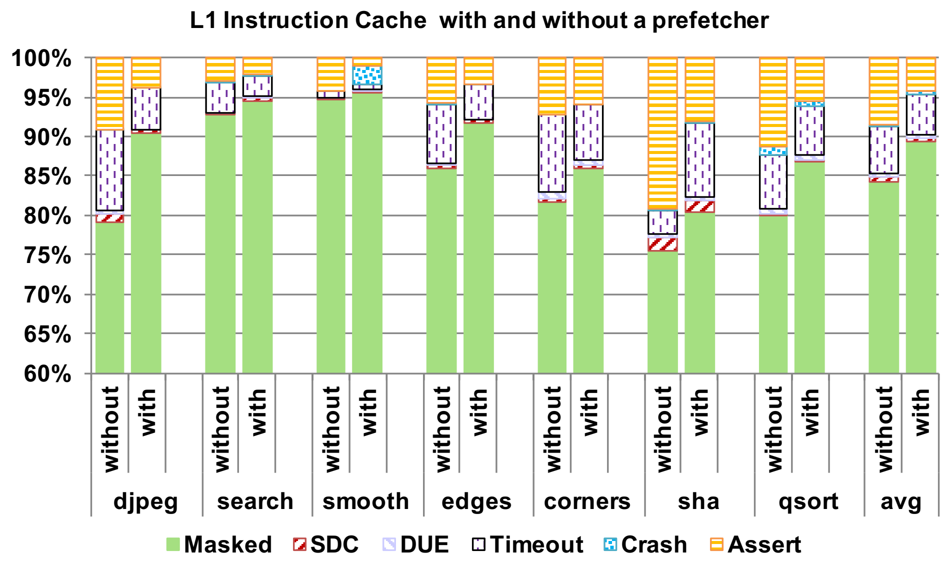


Figure 24: Faults classification in L1 Instruction cache (prefetcher).

Next, we present the results of our proposed *fitness function* to quantify the effect of modifications in microarchitectural parameters both in terms of performance and reliability. Assigning different values to parameter *a*, we adjust the impact of reliability and performance in the design decisions. Table 8 to Table 12 present the values of the fitness function for all the components of our study and for three different values of parameter *a*: 0.25 (design focused on performance), 0.50 (design balanced between reliability and performance), 0.75 (design focused on reliability). Moreover, for each microarchitectural configuration under study, Table 8 to Table 12 also show the FIT and IPC on average for all benchmarks (second row on each table). Individual benchmarks can be similarly studied. The fitness function values are normalized to the baseline configuration fitness (which has fitness value equal to 1.00). The “best” fitness values for different design priorities (the three *a* values) are highlighted with shaded cells. Fitness values greater than 1 indicate a design point which improves the fitness compared to the baseline model.

Finally, from the calculated FIT (Table 8 to Table 12) we observe that the most vulnerable component is the L2 cache with 2318.9 FIT for the baseline model (Table 12) and the most reliable is the LSQ with 0.5 FIT for the baseline model (Table 11). These results can be explained by the large and the small size of the L2 cache and the LSQ, respectively. Moreover, we can conclude to some general observations concerning the dependence of the reliability with the different microarchitectural features:

- **Size**: All structures follow the trend to feature less FIT rates (more reliable) for smaller sizes because the size of a structure is more dominant than the vulnerability factor (percentage of not masked categories) in the computation of FIT. Especially, the highest reliability (the smaller FIT rate) is observed for 16KB L1 Data cache (251.1 FIT), 16KB L1 Instruction cache (159.4 FIT), physical register file of 64 registers (4.3 FIT) and LSQ of 32 entries (0.5 FIT). The trend of both first level caches is similar to the reported findings in [53], which are based on ACE analysis and use SPEC2000 benchmarks [75].
- **Caches Associativity**: In general, the most reliable is the 2-way set associative L1 Data cache (346.4 FIT) and the 8-way set associative L1 Instruction cache (193.6 FIT).
- **Caches Write Policy**: Write through caches are more reliable than write back caches in all levels of memory hierarchy. In case of write through cache, a transient fault hitting a bit in the cache can only be propagated to the processor when the block is read before its eviction. However, in a write back cache the fault can be propagated to the processor or to the lower levels of memory hierarchy when the block is evicted and contains dirty data. The write through caches feature less FIT than write back caches: 240.4 FIT for L1 Data, 264.0 FIT for L1 Instruction and 1390.1 FIT for unified L2.
- **First Level Cache Prefetchers**: The L1 Data cache with prefetcher (405.4 FIT) and the one without prefetcher (392.3 FIT) have almost the same reliability since their FIT are close. The L1 Instruction cache is more reliable with enabled prefetcher because it features less failures in time (278.8 to 413.1 FIT with and without prefetcher respectively). A prefetcher can occasionally reduce the residency time of a cache line (by replacing it) or fill a cache line with useless data that is not used by the processor. Our results show that the presence of a prefetcher enhances only L1 Instruction cache’s reliability.

Similar to [28], in [94] the authors presented a detailed study to evaluate the reliability – performance tradeoff assessments between three different commercial CPUs (different ISAs and microarchitectures) and one GPU that execute the same benchmarks.

Table 8: FIT, IPC and Fitness values for the L1 Data cache.

L1D	Baseline WB, 4-way prefetcher,32KB	policy	Associativity			prefetcher	size	
		WT	1-way	2-way	8-way	w/o prefetcher	16KB	64KB
FIT *IPC*	405.4 0.7932	240.4 0.7606	366.3 0.7554	346.4 0.7682	383.7 0.7541	392.3 0.8569	251.1 0.7590	637.4 0.8043
a	*Fitness*							
0.25	1.000	1.141	0.991	1.019	0.977	1.069	1.121	0.919
0.5	1.000	1.323	1.030	1.069	1.004	1.057	1.286	0.825
0.75	1.000	1.504	1.068	1.120	1.030	1.045	1.450	0.731

Table 9: FIT, IPC and Fitness values for the L1 Instruction cache.

L1I	Baseline WB, 4-way prefetcher,32KB	policy	Associativity			prefetcher	size	
		WT	1-way	2-way	8-way	w/o prefetcher	16KB	64KB
FIT *IPC*	278.8 0.7932	264.0 0.7606	339.7 0.7554	300.5 0.7682	193.6 0.7541	413.1 0.8569	159.4 0.7590	427.3 0.8043
a	*Fitness*							
0.25	1.000	0.983	0.919	0.958	1.073	0.979	1.155	0.924
0.5	1.000	1.007	0.887	0.948	1.195	0.878	1.353	0.833
0.75	1.000	1.032	0.854	0.938	1.318	0.776	1.551	0.743

Table 10: FIT, IPC and Fitness values for the Integer Physical Register File.

Reg. File	Baseline 256 regs.	size 64regs.	128regs.
FIT *IPC*	4.7 0.7932	4.3 0.7057	4.6 0.8097
a	*Fitness*		
0.25	1.000	0.937	1.023
0.5	1.000	0.985	1.026
0.75	1.000	1.032	1.028

Table 11: FIT, IPC and Fitness values for the LSQ.

LSQ	Baseline 32 entries	size 64 entries	96 entries
FIT *IPC*	0.5 0.7932	0.6 0.7889	0.7 0.8025
a	*Fitness*		
0.25	1.000	0.973	0.962
0.5	1.000	0.951	0.912
0.75	1.000	0.929	0.863

Table 12: FIT, IPC and Fitness values for the L2 cache.

L2	Baseline WB	Policy WT
FIT *IPC*	2318.9 0.7932	1390.1 0.7694
a	*Fitness*	
0.25	1.000	1.145
0.5	1.000	1.319
0.75	1.000	1.494

2.2.3 Differential studies on microarchitectural fault injectors

Microarchitecture level fault injection tools that are developed on performance simulators (like MaFIN) are suitable for early reliability estimations as presented in detail in Section 2.1.3. During the last years, the development and the use of microarchitectural simulators in computer architecture exploded, giving computer architects the opportunity to model a great variety of different ISAs or even different microarchitectures for the same ISA. However, these tools are developed by programmers using high level description languages to describe all the details of the real hardware chips for performance studies, without modeling details that can influence the simulation time and are not necessary for performance assessments. As a result, some performance simulators may lack certain details necessary to perform accurate fault injection experiments.

Consequently, some important missing aspects concerning the sensitivity of the reliability assessments depending on the microarchitecture, the ISA and the simulator implementation still exist. In [29], we reveal some important insights concerning the reliability estimations using microarchitectural fault injection. The findings of this study can give answers to the following important questions:

1. What are the characteristics of a microarchitectural simulator that make it more suitable as a substrate for fault injection studies?
2. How sensitive is the vulnerability of hardware structures to the ISA as well as the microarchitecture (simulator model or hardware structures configurations) for a given workload?

The concept of study [29] is to investigate the limits of microarchitecture level fault injection for x86 and ARM ISAs conducting a *differential analysis* on two comprehensive fault injector tools supporting the same fault models and running the same workloads. Such a differential analysis can bring insights on the sensitivity of the vulnerability of hardware structures and workloads to the underlying microarchitecture as well as the ISA of the microprocessor. It can also identify common trends and diverging reliability reports in the two tools which can lead to informed design decisions for error protection. We explain the common trends and the sources of difference when diverging reliability reports are provided by the tools using benchmarks runtime statistics.

The combination of the two fault injection frameworks can also serve many different studies in the same differential context by injecting hardware faults on actual microarchitecture structures (all storage arrays: caches, register files, buffers, queues – not only on architecturally visible points) to better assist design decisions for error protection of individual components.

The microarchitecture-level fault injection tools that were used in this differential study [29], called MaFIN [27] [29] and GeFIN [30] (for MARSSx86-based and Gem5-based Fault Injector, respectively), are built on the two most popular microarchitectural simulators (MARSSx86 [14] and Gem5 [15]) and the two popular ISAs (x86 and ARM). We selected MARSSx86 and Gem5 because they are: *cycle-accurate* (thus can allow per cycle granularity of fault injections at any modeled hardware component), *publicly available*, and *regularly maintained* today by their developers. By themselves, these properties can justify the selection of MARSSx86 and Gem5 suitable for reliability studies, but also:

- They are widely adopted by the computer architecture community. Both simulators are recent and very popular. Their increased popularity is mainly due to their accurate support of important ISAs, their detailed and configurable model of the memory system [65] and check-pointing support.

- Their combination supports differential reliability studies. The combination of MARSSx86 and Gem5 supports the purposes of our work – reliability studies on different ISAs and reliability studies on the same ISA on different simulators. In particular:
    a. Both MARSSx86 and Gem5 support the x86 ISA and thus facilitate comparison of microarchitectural fault injections in the hardware components of an x86 microprocessor.
    b. Gem5 supports several ISAs; ARM and x86 are among the best supported and thus a comparative study of these two popular ISAs can be performed.
    c. Both MARSSx86 and Gem5 have a fully configurable model (pipeline depths and widths, structures sizes and organizations, etc.)
    d. MARSSx86 models both a high-performance OoO pipeline and a simple in-order (Atom-like) pipeline; a reliability assessment study between these two models can be implemented (the study of [29] focuses on the OoO model to compare with the corresponding one of Gem5).

An important difference between MARSSx86 and Gem5 is that they require different development efforts to support fault injection at the microarchitecture level. Gem5 already includes all key microarchitecture components that model hardware arrays on which faults of any duration and severity can be injected. MARSSx86, on the other hand, does not contain important arrays needed for fault injection: data/instruction arrays of caches at all levels. Details about the modifications that were made on the original model of MARSSx86 to implement MaFIN were presented in Section 2.2.1.

Both MaFIN and GeFIN injectors have been developed modularly using exactly the same principles and employ the check-pointing features of the simulators to ensure that faults affect only the execution of the benchmark being studied as well as to speed up the injection campaigns. The backbone of both simulators consists of the same three modules as were presented in Figure 13: a fault mask generator, an injection campaign controller and a parser of the logged information. The tools allow studies on the full range of fault models: transient, intermittent and permanent, as well as studies with multiple faults injected in: (i) different bits of the same entry of a hardware structure, (ii) different entries of a structure, (iii) different hardware structures simultaneously, (iv) all combinations of the above. Finally, the parsers of both injectors (for the purposes of our differential study) were modified to classify the fault effects in the same six fault effect categories (Masked, SDC, DUE, Timeout, Crash, Assert) as were defined in Table 4.

Table 13 summarizes the state in microarchitectural fault injectors and puts the new contributions of [29] study in this context. The combination of the two new microarchitectural fault injectors used for the needs of our differential study cover several important missing aspects of the research area.

Table 13: State-of-the-art and contributions of [29] in fault injection techniques on microarchitectural simulators.

Aspect	State-of-the-art	Our differential work [29]
Injection framework that targets all major microarchitecture structures	None[1]	Both MaFIN and GeFIN: all major structures
Comparison between ISAs (x86 vs. ARM)	None	GeFIN (x86 vs. ARM ISA)
Comparison between Out-of-Order microarchitectures	None	MaFIN and GeFIN
Comparison between simulators for same ISA	None	MaFIN and GeFIN (for x86 ISA)
Full system fault injection	[15]: Gem5; [48]: M5; [78] [79]: GEMS	Both MaFIN and GeFIN are full system injectors
New microarchitectural structures added	None	MaFIN
Transient, intermittent, permanent fault models	[48] (not all hardware structures)	MaFIN and GeFIN: all fault models

The three different configurations of MARSSx86 and Gem5 on which we performed our experimental study and analysis are summarized in Table 14. MARSSx86 simulates x86 ISA while the x86 and ARM ISAs of Gem5 have been used. Both MaFIN and GeFIN injectors can be easily modified for other values of the parameters shown in Table 14. Our main focus in setting the parameters was to keep the sizes and organizations of the hardware structures the same (or as close as possible) in the two simulators.

For all the fault injection campaigns of our differential study, we injected 2000 transient faults that correspond to 2.88% error margin and 99% confidence level (according to [62] as described in Section 2.1.5). Furthermore, we used 10 benchmarks from the MiBench suite [74] (*djpeg*, *search*, *smooth*, *edge*, *corner*, *sha*, *fft*, *qsort*, *cjpeg*, *caes*).

From Figure 25 to Figure 29, we present the results of all the fault injection campaigns of [29] targeting the Integer physical register file (Figure 25), the Load/Store Queue (Figure 26), the L1D cache (Figure 27), the L1I cache (Figure 28) and the second level cache (Figure 29). Each graph shows for a particular component the faulty behavior classification (using the classes of Table 4) for each of the 10 benchmarks and on the average. For each benchmark the graphs show three stacked bars (each bar corresponds to a fault injection campaign): one for the execution on the MaFIN-x86 injector (M-x86 bar), one on the GeFIN-x86 configuration (G-x86) and one on the GeFIN-ARM configuration (G-ARM). For the average case, the same three bars are shown at the rightmost end of each diagram.

[1][58]: integer register file and ROB only; [48]: no injections supported in any cache level.

Table 14: Simulators configurations for [29] study.

Parameter	Simulator / ISA		
	MARSS/x86	**Gem5/x86**	**Gem5/ARM**
Pipeline	OoO	OoO	OoO
Physical register file	256 int; 256 FP; 16 store; 24 branch	256 int; 128 FP	256 int; 128 FP
Issue Queue entries	32	32	32
Load/Store Queue entries	32 (unified)	16 (load)/ 16 (store)	16 (load)/ 16 (store)
ROB entries	64	40	40
Functional units	2 int ALUs; 2 FP ALUs; 4 AGUs	6 int ALUs; 2 complex int ALUs; 4 FP ALUs, 2 FP mul/div, 4 SIMD	2 int ALUs; 1 complex int ALUs; 2 FP & SIMD
L1 Instruction Cache	32KB, 64B line, 128 sets, 4-way, write back	32KB, 64B line, 128 sets, 4-way, write back	32KB, 64B line, 128 sets, 4-way, write back
L1 Data Cache	32KB, 64B line, 128 sets, 4-ways, write back	32KB, 64B line, 128 sets, 4-ways, write back	32KB, 64B line, 128 sets, 4-ways, write back
L2 Cache	1MB, 64B line, 1024 sets, 16-way, write back	1 MB, 64B line, 1024 sets, 16-way, write back	1 MB, 64B line, 1024 sets, 16-way, write back
Branch Predictor	Tournament predictor	Tournament predictor	Tournament predictor
Branch Target Buffer	direct branches BTB (4-way, 1K entries), indirect branches BTB (4-way, 512 entries)	conditional and unconditional branches BTB (direct-mapped, 2K entries)	conditional and unconditional branches BTB (direct-mapped, 2K entries)
RAS	16 entries	16 entries	16 entries

The first observations from the average vulnerability reports at the rightmost bars of each diagram reveal the following:

- The largest average case vulnerability differences are observed between the two x86-based configurations (MaFIN-x86 and GeFIN-x86): 7.20 percentile points in the L1D cache, 3.61 percentile points in the L1I cache, and 1.36 percentile points in the L2 cache.
- On the contrary, the vulnerability differences between the two ISAs (x86 and ARM) on GeFIN are much smaller in all components. In the L1D and L2 cache

the differences between GeFIN-x86 and GeFIN-ARM configurations are only 0.55 and 0.13 percentile points respectively, while in the L1I cache the average difference is 2.03 percentile points.

Next, we analyze in more detail the results of the classification shown in the diagrams both on a per-component basis to identify consistent trends and on a per-benchmark basis to interpret diverging behaviors. We discuss potential (microarchitecture or ISA related) reasons that explain the differences between the two tools providing execution statistics for the benchmarks.

<u>Integer Register File and LSQ:</u>

The Integer Register File (Figure 25) and the LSQ (Figure 26) are the least vulnerable components in all cases (benchmark, ISA and microarchitecture configuration). The vulnerability (sum of all non-masked classes) of the Register File and the LSQ for each individual benchmark and on the average across all benchmarks is almost always less than 3% for all three configurations. This is a consistent behavior that is also compatible to previous literature reports. The two components hold data of relatively short lifetime which explains the small vulnerability to transient faults.

- <u>Remark 1</u> – There is a consistent small difference of ~1 percentile point between the MaFIN and GeFIN report for the LSQ vulnerability (LSQ in MaFIN is always slightly more vulnerable than the GeFIN's LSQ). The reason for this slight difference is that MARSSx86 implements a unified queue for loads and stores while Gem5 implements different queues and only the store queue holds data. Therefore, our injections on GeFIN's LSQ affect only stores while in MaFIN both queues are affected by faults.
- <u>Remark 2</u> – Both the Integer Register File and the LSQ have mixed faulty behaviors in the non-masked classes. Faults in both components in most cases can lead to any of the five non-masked faulty behaviors (SDC, DUE, Timeout, Crash, and Assert). The exact numbers in each class of course depend on the benchmark.

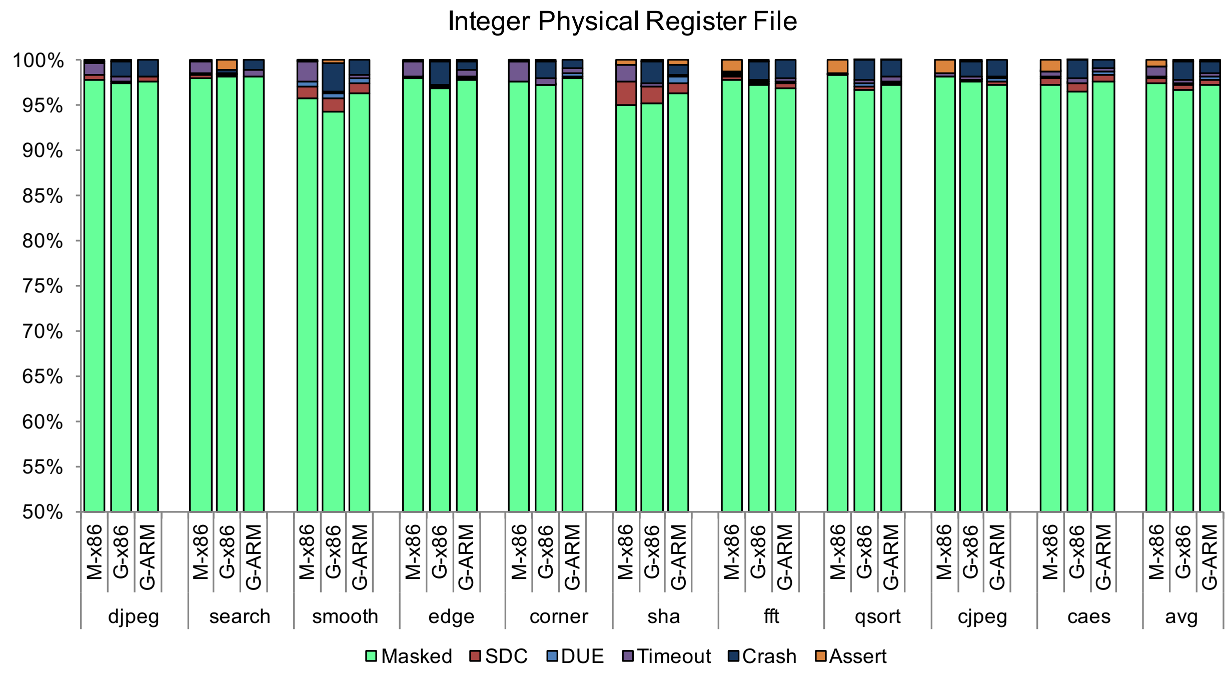


Figure 25: Faulty behavior classification for the integer physical register file.

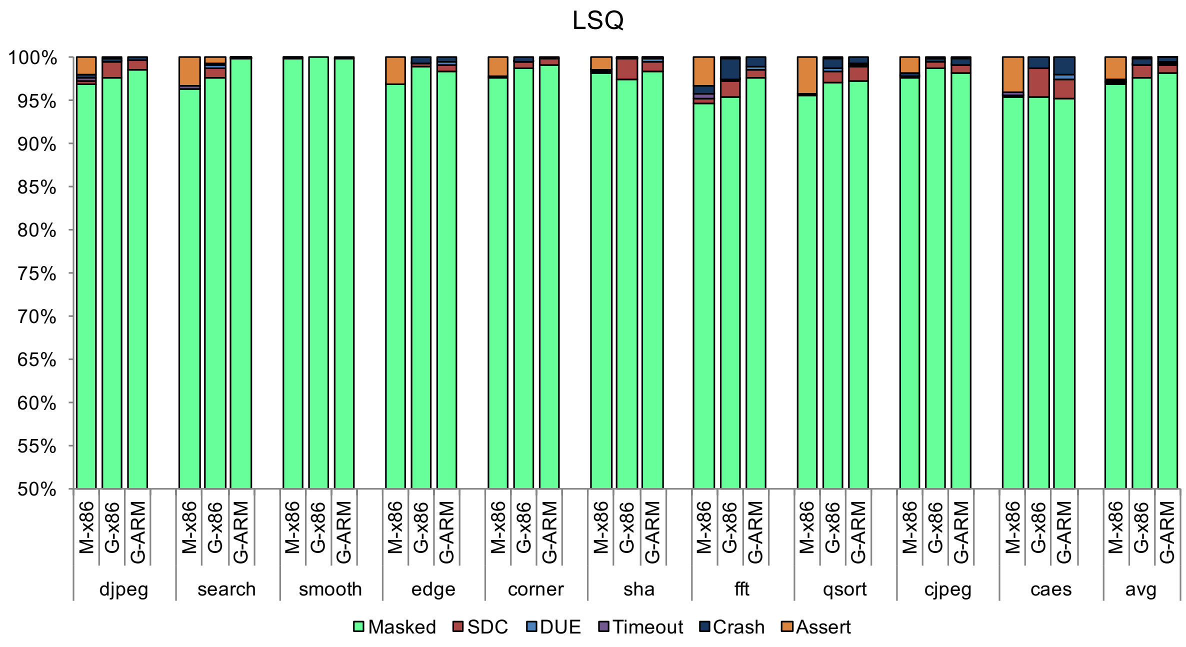


Figure 26: Faulty behavior classification for Load/Store Queue (data field).

L1 Data Cache:

In general, the first-level cache memories (L1D cache in Figure 27 and L1I cache in Figure 28) are the most vulnerable components in all cases (benchmark, ISA and microarchitecture configuration).

The L1D cache vulnerability varies significantly among benchmarks and between ISAs and microarchitectures. Its vulnerability can be as low as 2.5% (*search* benchmark in the MaFIN-x86 setup) and as high as 47.3% (*cjpeg* benchmark in the GeFIN-x86 setup). On average across benchmarks the L1D cache vulnerability is less than 15% in MaFIN-x86 while in both ISAs of GeFIN (GeFIN-x86 and GeFIN-ARM) it is more than 22%. The general trend in most (but not all) individual benchmarks is that MaFIN reports a less vulnerable L1D cache than GeFIN.

- *Remark 3* – The significant ~7 percentile point difference between MaFIN and GeFIN vulnerability reports on the L1D cache can be attributed to two main differences between the two microarchitectural simulators:

  The MARSSx86 CPU model uses more aggressive approaches than Gem5 (and other simulators) for loads issue. Load instructions are issued as soon as possible and before aliasing with earlier stores is determined. For this reason, the number of executed loads in MaFIN is significantly larger than in GeFIN although (for each benchmark) the number of committed loads is very close to each other. This significant difference leads to extra masking of the faults in L1D on MaFIN and along with the previous point consistently explains the L1D cache vulnerability differences between the two tools. For example, in *fft*, *cjpeg*, *caes* (the benchmarks with largest difference in L1D between MaFIN-x86 and GeFIN-x86) MaFIN issues 2.6x, 4.7x, 2.0x more loads than GeFIN; this confirms the general trend.

  MARSSx86 employs the QEMU hypervisor for system functions as well as for unimplemented instructions. When QEMU is invoked, the cache of the microarchitecture is not accessed (memory accesses go to the main memory) – for this reason faults in the L1D cache are masked and do not affect the operation when QEMU runs (this is not the case in the L1I cache; see below). Gem5 on the other hand handles the complete system operation internally and does not employ a hypervisor so this type of masking does not happen.

However, in *qsort* and *smooth* the expected higher masking in MaFIN is not observed. For *qsort* GeFIN-x86 has smaller L1D read hit rate than in MaFIN-x86 (by 0.64x), while GeFIN-x86 has higher L1D write hit rate than MaFIN-x86 (1.91x in qsort and 1.57x in smooth); this means that in MaFIN-x86 for these benchmarks faults in L1D are less likely to be over-written and thus MaFIN-x86 is more vulnerable than GeFIN-x86.

- Remark 4 – The prevailing faulty behavior in the L1D cache is the SDC class (intuitively expected) which leads to corrupted benchmark final output. In all benchmarks and the average case the SDC class is from 3x to 5x larger than the sum of all four other non-masked classes.
- Remark 5 – The most remarkable differences between the different ISAs (GeFIN-x86 and GeFIN-ARM) for the L1D cache are observed in *fft*, *qsort* and *cjpeg*. The ARM model has 2x more store instructions than that of x86 in the *fft* benchmark, while in *cjpeg* the L1D write misses of the ARM model are 6x more than x86 model, which naturally leads to more vulnerability for the x86 model for these two benchmarks. The GeFIN-x86 model in *qsort* follows a completely different memory access pattern which reports significantly more L1D replacements (4x) than GeFIN-ARM. This indicates that the ARM model is more vulnerable for *qsort* than the x86 model.

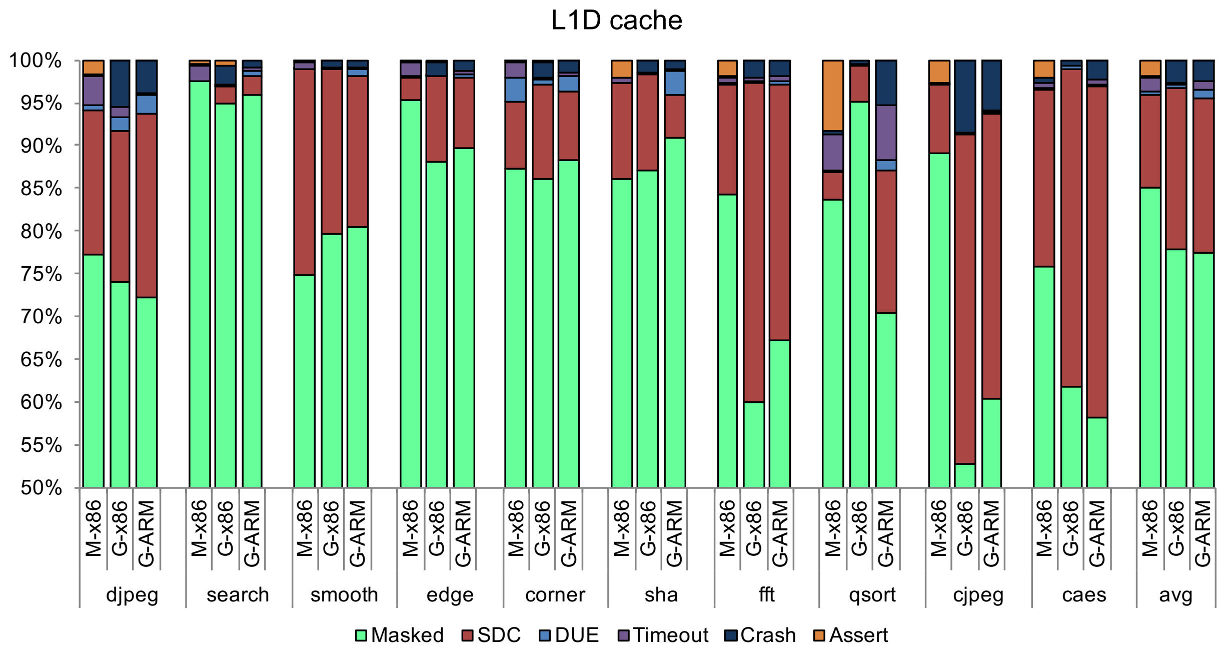


Figure 27: Faulty behavior classification for L1D cache (data arrays).

L1 Instruction Cache:

The L1I cache vulnerability on the other hand (Figure 28), is less variable across benchmarks than the L1D cache but still it can be as low as 5.3% (*smooth* benchmark in the MaFIN-x86 setup) and as high as 34.5% (*caes* benchmark in the MaFIN-x86 setup). On average across benchmarks, L1I cache vulnerability is around 19% in MaFIN-x86 while in both ISAs of GeFIN (GeFIN-x86 and GeFIN-ARM) it is more than 14%. Here, the general trend in most (but not all) individual benchmarks is that MaFIN reports a more vulnerable L1I cache than GeFIN (the opposite trend to L1D reports).

- Remark 6 – Unlike L1D, the QEMU hypervisor does not affect the behavior of L1I cache. QEMU may be invoked during decode stage only, which is after fetching (and accessing of L1I). This means that any faults residing in the L1I cache can be propagated without disturbance by the hypervisor. On the other hand, MARSSx86 and Gem5 have differences in the implementation of their front-end that can lead to different prediction accuracy. Both simulators implement a Tournament predictor, consisting of a local and a global predictor. A meta-predictor takes the final decision based on the accuracy of the local and global ones. The most noticeable difference between MARSSx86 and Gem5 is that the final prediction is bound to the branch address in the case of MARSSx86 and to the global branch history in the case of Gem5. Branch address is not taken into account at all on the decision of Gem5 global predictor as well. This prediction scheme difference leads to different memory access patterns and L1I cache state; this can explain the small differences in the masked category between MaFIN-x86 and GeFIN-x86. Unfortunately, there is no consistent trend for all benchmarks. For instance, in *edge*, *corner* and *sha* benchmarks MaFIN-x86 has by 0.83x, 0.82x, 0.68x less mispredictions than GeFIN-x86 which implies that GeFIN-x86 brings more L1I blocks from lower levels, increasing the probability to overwrite faults.
- Remark 7 – The *fft*, *qsort*, *caes* are the benchmarks with difference more than 5 percentile points between GeFIN-x86 and GeFIN-ARM. For these benchmarks the replacements of L1I blocks in ARM model are 4.2x, 2.0x, and 7.2x more than in the x86; this can explain a more vulnerable x86 behavior than the ARM model.
- Remark 8 – Figure 28 shows that SDCs in the L1I cache are less frequent than in the L1D cache. The prevailing non-masked behavior in L1I cache in the MaFIN injector is the Assert class, while in the GeFIN injector the Crash class prevails. This difference is because MARSSx86 simulator includes a significantly larger number of assert instructions checking points in its code which are raised during faulty executions of the benchmarks and stop the simulation abnormally. On the other hand, assertion checking in Gem5 is compact and less frequent and for this reason injected faults eventually lead to crashes.

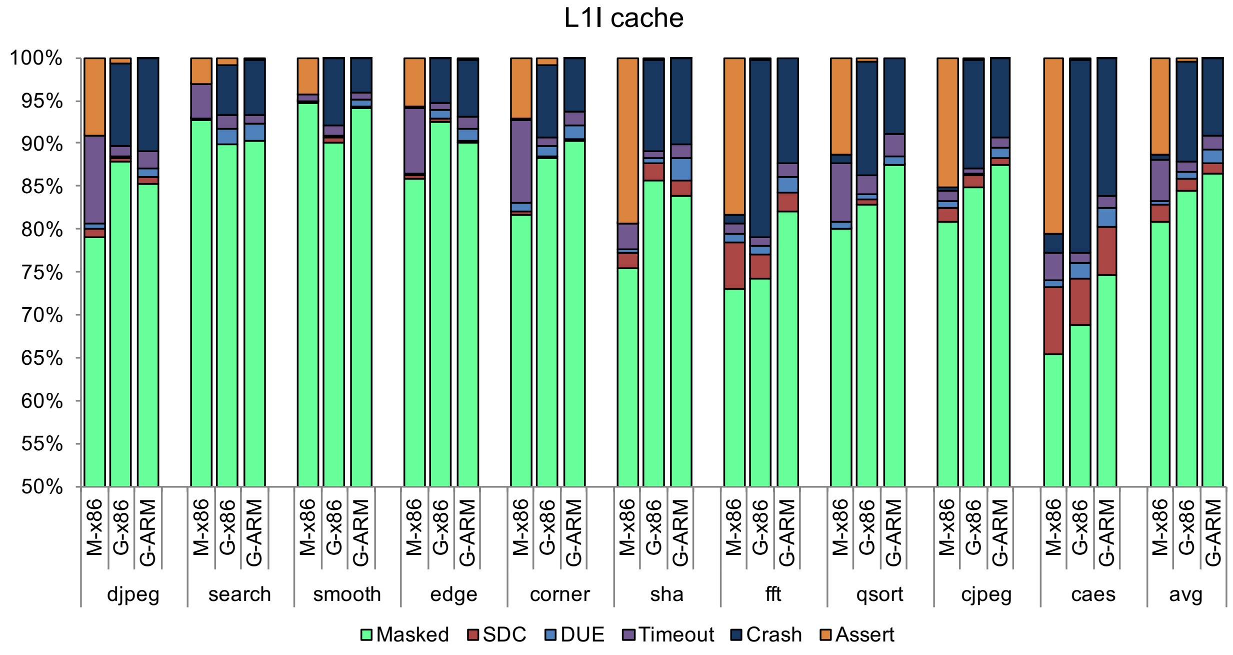


Figure 28: Faulty behavior classification for L1I cache (instruction arrays).

<u>Second-Level Cache (L2):</u>

The L2 cache memory vulnerability (Figure 29) is in all cases (benchmark, ISA and microarchitecture configuration) a few percentile points higher than the Register File and LSQ and significantly lower than both first-level caches. On average, it ranges between 6% and 7% for the three ISA and microarchitecture combinations. The difference in the L2 cache vulnerability between MaFIN and GeFIN is only about 1 percentile point which shows a consistent behavior between the two tools.

- *<u>Remark 9</u>* – Since L2 is unified the vulnerability reports show a balance between SDCs and other abnormal classes (Crashes etc.).
- *<u>Remark 10</u>* – Vulnerability differences larger than 5 percentile points are observed in *cjpeg* and *caes* benchmarks between MaFIN-x86 and GeFIN-x86 for the L2 cache. In *cjpeg* GeFIN-x86 has 1.2x more L2 write misses than MaFIN-x86, while in *caes* GeFIN-x86 has 1.54x more write hits than MaFIN-x86 increasing the probability that a fault is overwritten.
- *<u>Remark 11</u>* – Concerning the ISA differences between GeFIN-x86 and GeFIN-ARM, *djpeg* is the only benchmark with difference larger than 5 percentile points. In this case, the x86 model has 0.5x less L2 read hits and 6.8x more L2 write misses than the ARM model, making this benchmark less vulnerable for the x86 architecture.

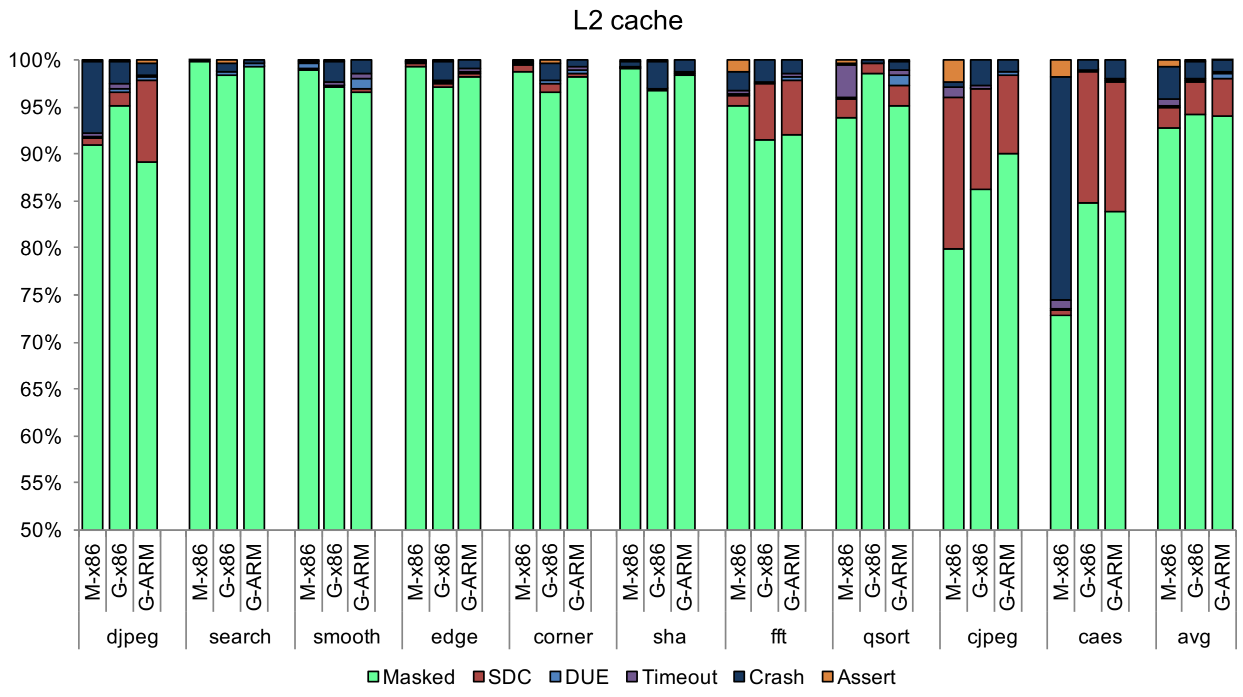


Figure 29: Faulty behavior classification for L2 cache (data arrays).

2.2.4 Acceleration of fault injection campaigns based on the faults lifetime

Despite the fault injection accuracy, execution time remains the major drawback of this technique considering that many campaigns must be completed early in design phase, for different components, microarchitectural characteristics, protection mechanisms and workloads. Except for leveraging parallelism of modern computing systems to run simultaneously multiple campaigns in multiple threads and workstations, statistical fault injection is also used to deliver quick and accurate estimations. Using lower confidence level and higher error margin, far less experiments could be executed in expense of accuracy.

In [31], we extend the baseline mode of an out-of-order cycle accurate full-system x86-64 fault injection framework (MaFIN) with two extra modes of operation in order to speed up the statistical fault injection campaigns at the microarchitecture level. The common characteristic of the two proposed techniques of [31] is that they are implemented after the actual injection of the fault in the hardware structure during its lifetime. In the first mode, an injection experiment is forced to completion when the fault is overwritten before it is read and thus we classify it early and accurately as Masked. In the second mode, an injection experiment is forced to completion before the end of the application in two cases: (a) when the fault is overwritten before it is read, or (b) when an x86 instruction reads the fault from the faulty entry and reaches the commit stage. The second method provides a tradeoff between speedup and accuracy in order to deliver a fast but less accurate solution in the early reliability estimation problem. Next, we will describe in details the proposed methods to accelerate fault injection runs after the actual injection of the fault.

For the needs of our experiments we used MaFIN framework [29] and we modified it to support fault injection in three different operation modes: Full Execution (***Baseline***), Early Stop on Overwrite (***ESO***), Early Stop on Overwrite or first Read (***ESOR***). All modes adopt different criteria and categories to assess fault injection's outcome that are presented below:

- ***Full Execution – Baseline***: In this mode, we run the application to the end and classify the outcome of a fault injection experiment in comparison with the outcome of a golden run. The classes used in this mode are the common six classes used in our reliability estimation studies (Masked, SDC, DUE, Timeout, Crash, Assert) that were presented in Table 4.
- ***Early Stop on Overwrite – ESO***: In this mode, we run the application to the end except for the cases that the fault is overwritten before it is read or the fault is injected in an invalid entry. The former mode (*Baseline*) has been extended to identify if the fault is masked prior to its use. In such cases, we can safely characterize the fault injection experiment as masked and thus stop it early, before the simulation's completion. The extra logic raises safely an assertion message handled by the parser. Thus, the former classification (*Baseline*) was extended with the following sub-classes for the masked category that are presented in Table 15.

Table 15: Fault effect classification of Masked category for ESO mode in [31].

Fault Class	Definition
Write After Injection (WAI)	The fault was overwritten or injected in an invalid entry
not Write After Injection (not WAI)	The fault was finally masked but it was not detected as overwritten or injected in an invalid entry

Early Stop on Overwrite or first Read – ESOR: In this mode, we run the application to the end except for the cases that the fault is overwritten or is read by a committed instruction. The former mode (*ESO*) has been extended to identify when corrupted data (in presence of fault) are read. The extra logic raises an assertion when an instruction reads the faulty bit and passes through the commit pipeline stage, meaning that the architectural state has been corrupted by the fault. This mode of operation consists of four classes of fault effects that are presented in Table 16. Note that, the sum of *WAI* and *Unused* categories defines a conservative lower boundary of the overall masking probability of a structure, while the sum of the *RAI* and *Unknown* categories represents the overestimated vulnerability of the structure.

Table 16: Fault effect classification of ESOR mode of study [31].

Fault Class	Definition
Read After Injection (RAI)	The faulty bit is read in the execution stage by an instruction and then this instruction reaches commit stage. Undoubtedly, the early stop of experiments right after the use of corrupted data limits our potential for accurate reliability characterization because we ignore any masking on the software level.
Write After Injection (WAI)	The faulty bit is written or the fault is injected in an invalid entry. Thus, the fault experiment is safely classified as masked in both cases.
Unknown	The corrupted data (due to the fault) are moved from the target structure to another structure and are still potentially harmful for the system. These cases appear in L1 data cache and in the unified L2 cache of our studied structures as they both have write-back policy. For instance, a cache line of a lower cache level or memory updates its data with a corrupted block of data that was evicted by a higher level cache. This block could influence the correct execution of the program during the rest of its execution, but we do not trace the fault propagation through the memory system.
Unused	The outcome of the experiment cannot be classified to any of the above categories i.e. the moment of injection is close to the completion of the experiment and its impact never manifests or the fault is injected in an unused by the program entry.

We enhanced MaFIN to support *ESO* and *ESOR* modes in order to speed up fault injection campaigns. Fault injection on *ESOR* mode is a tradeoff between speedup and accurate reliability estimation because the overestimation of vulnerability is inevitable. Table 17 summarizes the vulnerability functions used in all modes of operation.

Figure 30, Figure 31 and Figure 32 illustrate the three modes of operation presented in [31] (*Baseline*, *ESOR* and *ESOR* modes respectively), along with the speedup that *ESOR* and *ESOR* modes provide compared to the *Baseline* mode.

Table 17: Vulnerability across modes of operations of study [31].

Operation Mode	Vulnerability Function
Baseline	1 – Masked
ESO	1 – Masked
ESOR	1 – WAI – Unused

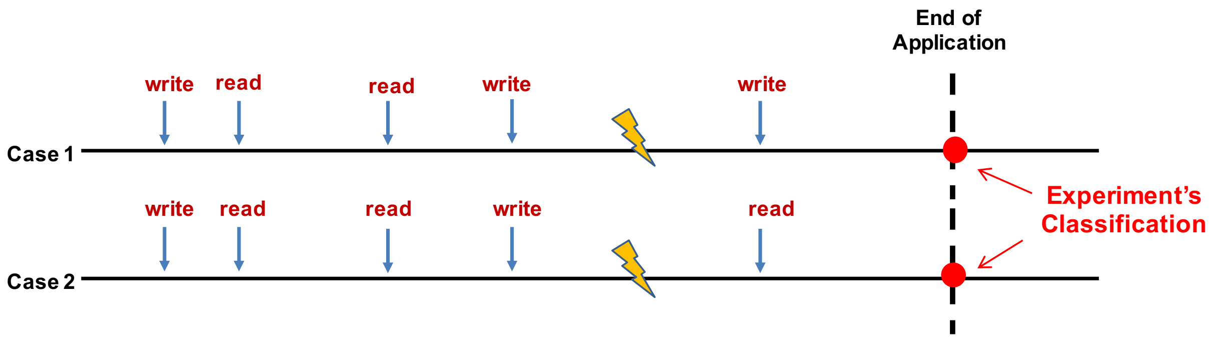


Figure 30: Baseline mode of operation presented in [31].

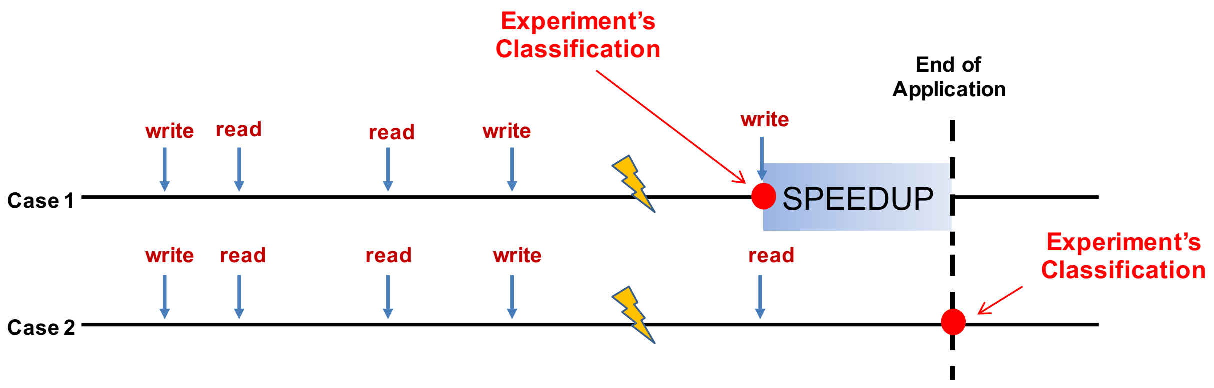


Figure 31: ESO mode of operation presented in [31].

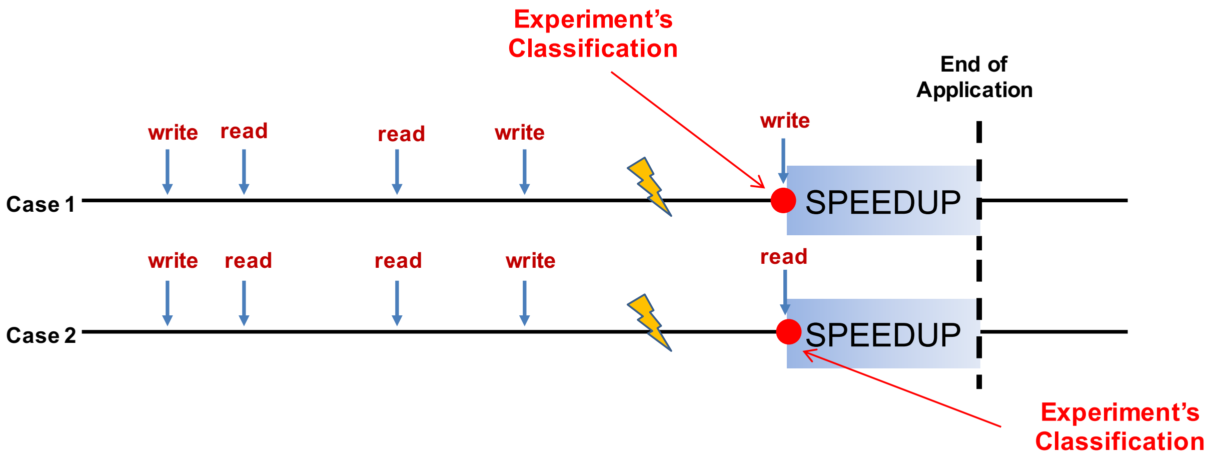


Figure 32: ESOR mode of operation presented in [31].

Figure 33 illustrates the correlation of categories among the three modes. The categories of *Baseline* and *ESO* mode are equivalent but this is not the case for the categories of *ESOR* mode. The *WAI* and *Unused* categories of the *ESOR* mode certainly lead to *Masked* but the *Unknown* and *RAI* categories may result in any category of the *Baseline* or *ESO* modes.

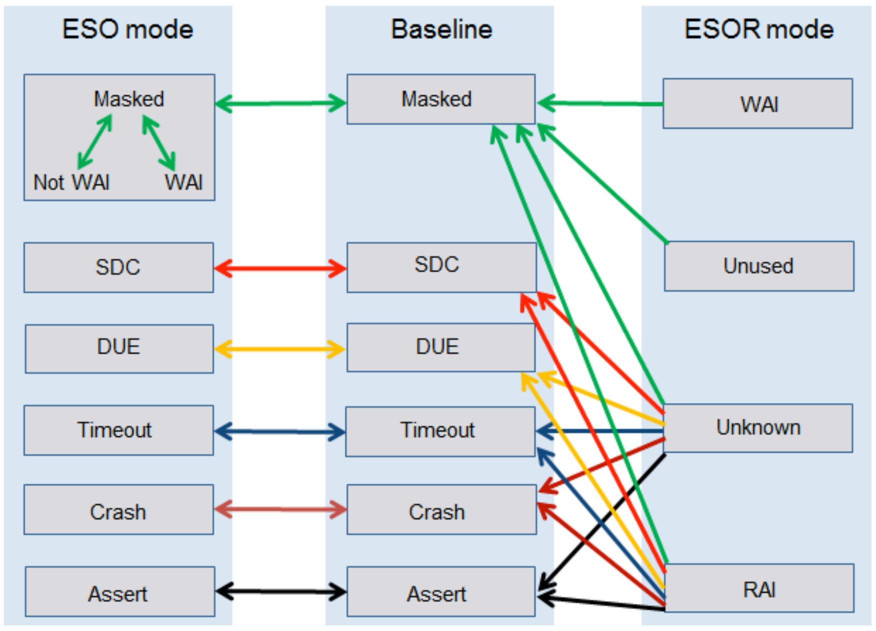


Figure 33: Correlation of classes among the three modes of presented in [31].

In this study, we used MaFIN to carry out extensive fault injection campaigns of transient faults in six structures of the microprocessor that hold the majority of chip's area: L1 Data cache, L1 Instruction cache, L2 unified cache, Physical Integer Register File, LSQ (data field) and LSQ (address field). We used seven benchmarks from the MiBench suite (*djpeg*, *search*, *smoothing*, *edges*, *corners*, *sha*, *qsort*) [74]. The microprocessor's baseline configuration used in all the modes of framework's operation, resembles a modern out-of-order x86-64 microprocessor and is illustrated in Table 18.

Table 18: Baseline configuration of study [31].

Component	Characteristics of Baseline Model
Pipeline	OoO
Physical Integer Register File	256 registers
L1 Data cache	32KB, write-back, 4-way set associative, 64B block size
L1 Instruction cache	32KB, write-back, 4-way set associative, 64B block size
L2 cache (unified)	1MB, write-back, 16-way set associative, 64B block size
LSQ (unified)	32 entries (16 load and 16 store entries)

Each statistical fault injection campaign consists of 2000 experiments and one golden run. A separate injection campaign was performed for each hardware structure, each benchmark and each of the three modes of operation. This number of experiments corresponds to 2.88% error margin and 99% confidence level according to [62]. A total number of 252,126 injection runs (7 benchmarks x 6 structures x 2001 injections x 3 operation modes) were performed.

Next, we present and analyze the results of the aforementioned three modes of operation (*Baseline*, *ESO* and *ESOR*), in terms of accuracy of the final reliability estimation and speedup compared to the *Baseline* full-execution mode. Each column in the following graphs corresponds to one microarchitectural structure, representing the average values obtained from executing statistical fault injection campaigns running the 7 aforementioned MiBench benchmarks of this study.

<u>Full Execution – Baseline mode:</u>

The first part of our analysis concerns the *Baseline* mode, where the fault injection experiments run to the end. The results for the six structures of our study are presented in Figure 34. It is observed that the first level caches are the most vulnerable among the structures, while the unified L2 cache is the most reliable. The vulnerability of L1D, L1I and L2 cache is 14.95%, 9.95%, 2.08% respectively. Despite the fact that data in L2 cache may present more residency than in first level caches, the result of more reliable L2 cache can be explained because cache blocks are more often used from first level caches than L2 cache, increasing the probability of fault propagation to the core. Furthermore, the evicted blocks from the write-back L1D cache can probably over-write the faults that were injected on L2.

Moreover, in L1D cache the SDC category dominates as it hosts data that can corrupt silently the output of the program, while L1I cache holds mostly data that their corruption can probably lead to crashes or assertions and termination of the experiment. In Integer Physical Register File and L2 cache the not-masked categories are well-balanced, while in LSQ the Assert class dominates among the not-masked categories. The vulnerability of Register File, LSQ (data field) and LSQ (address field) is 2.86%, 2.60% and 3.78%, respectively.

The prevailing category is the Masked across all structures and ranges from 85.05% to 97.92%. In general, Masked category consists of faults that are overwritten at microarchitecture level or application level. Especially, the speedup of the proposed modes stems mainly from microarchitectural level masking.

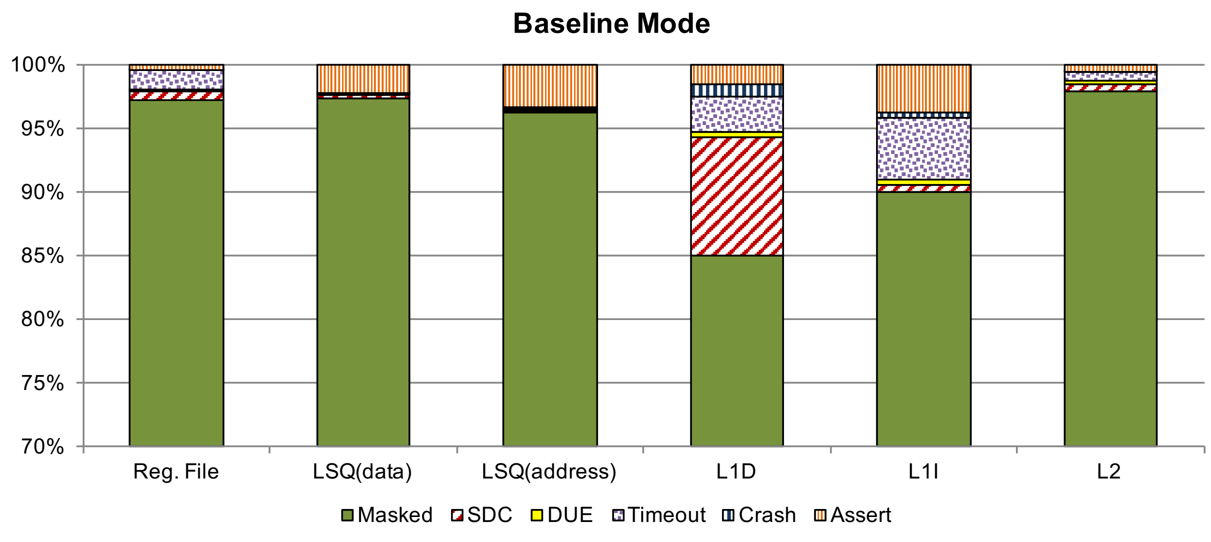


Figure 34: Faulty Behaviors classification of Baseline and ESO mode.

Early Stop on Overwrite – ESO mode:

In this mode of operation the experiment is stopped only when the fault is over-written before it is read or the injection is targeted on an invalid entry. Consequently, *ESO* mode decreases the simulation execution time without sacrificing the accuracy of reliability estimation. For this reason, the reliability classification of this mode is exactly the same with that of the *Baseline* mode, illustrated in Figure 34.

In essence, the degree of speedup in our fault injection campaigns using *ESO* mode is strongly related to the percentage of experiments that can be definitely characterized as masked, allowing the termination of the experiment before its completion, without any loss of accuracy. In Figure 35, the *WAI* class represents the experiments that can be safely stopped before the end of the experiment and classified as masked. The worst case is that of L2 cache that has only 11.71% *WAI* experiments, while Integer Physical Register File and LSQ (address field) present the highest amount of *WAI* (91.36% and 93.95% respectively). Moreover, the *WAI* category in LSQ (data field), L1D and L1I is 43.20%, 46.71% and 62.84% respectively. The low percentage of *WAI* category observed in the caches is related to the access patterns of the workload. In our experiments, L1D and L1I have more over-written faults as they both have more blocks coming from the lower level of memory hierarchy and L1D has more store hits than L2 cache. The Register File has a high percentage of over-written faults, because many injections took place in a period of time when the data of the register were not useful. Finally, the *WAI* class of LSQ (data field) is higher than that of LSQ (address field), due to the forwarding of data from store entries to load entries when a dependency exists between them.

The major conclusion coming up from Figure 35 is that for all the structures except for L2 cache there is great potential of injection campaign speedup using *ESO* mode of operation. This is based on the fact that the more the over-written faults, the more speedup will be gained during the fault injection campaign.

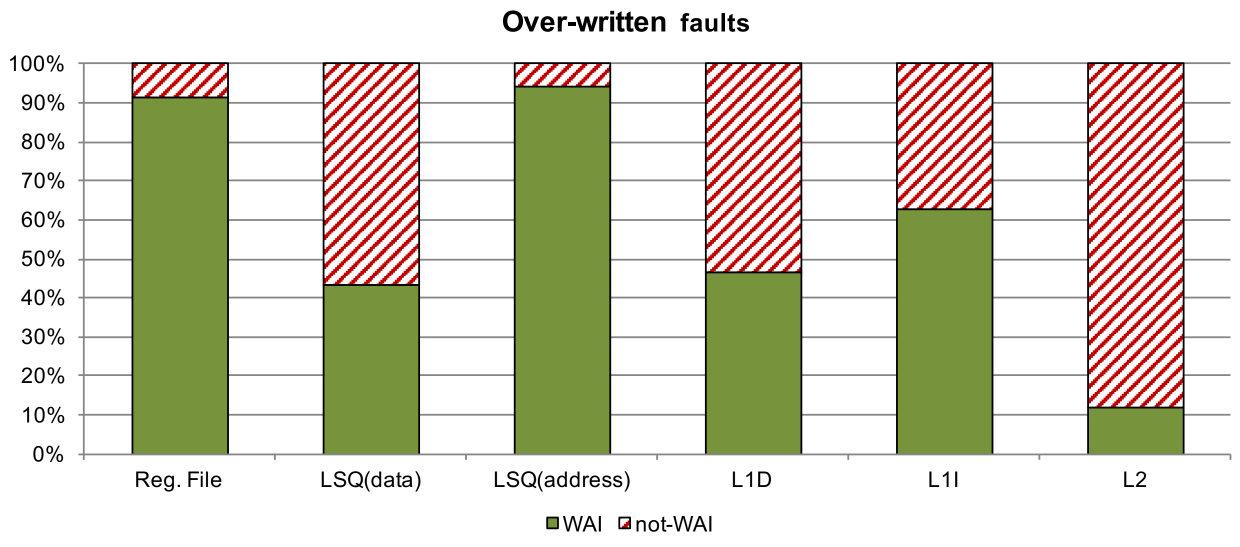


Figure 35: Percentage of over-written or injected on invalid entry faults.

Early Stop on Overwrite or first Read – ESOR mode:

The third mode of operation balances between reliability estimation accuracy and speedup of the injection campaign. Thus, this mode of operation must be evaluated in terms of accuracy and speedup compared to the full execution of the entire campaign (*Baseline* mode). Figure 36 presents the reliability classification of the *ESOR* mode for all structures and benchmarks. Note that the reliability of a structure in this mode consists of the percentage of *WAI* and *Unused* category, while the *RAI* and *Unknown* categories can be potential not-masked, corrupting the correct execution of the program. Another important note concerning speedup is that *Unused* category ensures the accuracy of the estimation, but it does not contribute to speedup, as the experiment runs to the end for this case.

Moreover in Figure 36, it is observed that the three caches consist of a high percentage of *Unused* category (~24% for the first level caches and 79.29% for the L2), omening that the speedup of caches will be limited by this factor. Conversely, the high percentage of *WAI* omens a good speedup for the cases of Integer Physical Register File and LSQ, which present a percentage of more than 86% in their *WAI* category. The percentage of *RAI* for all structures is less than 10%, except for L1D that has 11.83% and L1I cache with 23.69%. Finally, L1D cache presents a high percentage (~23%) of the potentially not-masked category (*Unknown*), as there are many evictions of dirty blocks from the L1D cache to the lower memory levels, but this is less intense in L2 cache with only 9.85% *Unknown*.

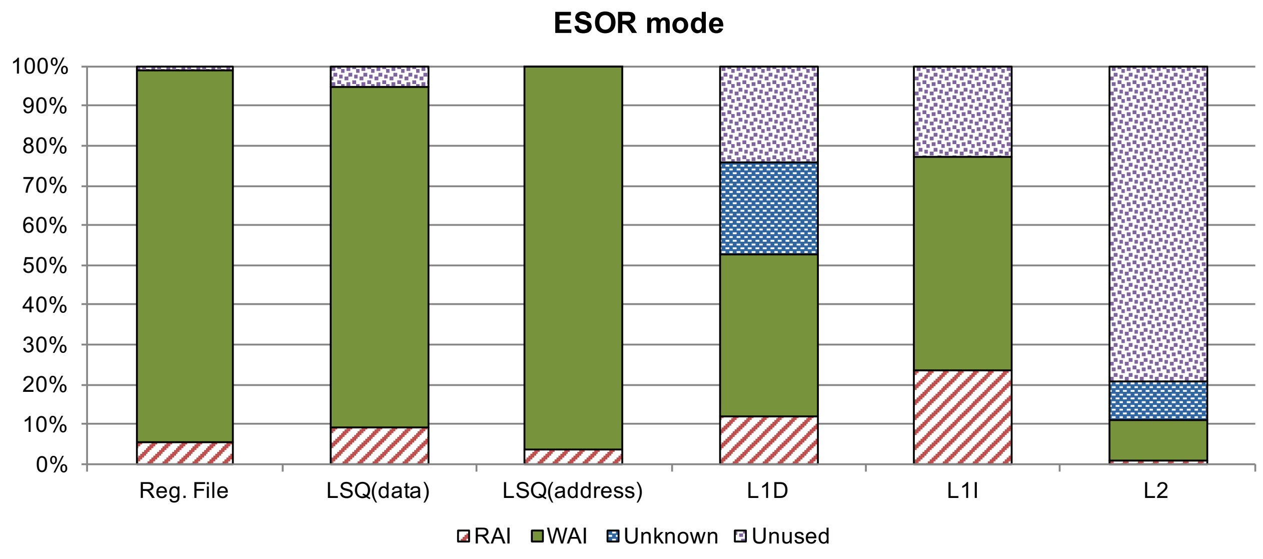


Figure 36: Faulty Behaviors classification of ESOR mode.

Based on the definition of vulnerability for the three modes of operation as illustrated in Table 17 we evaluate the inaccuracy of the *ESOR* mode of operation as presented in Figure 37. The inaccuracy between *Baseline* and *ESOR* mode is only 2.66 percentile untits in the Integer Physical Register File, 0.10 in the address field of LSQ, 6.58 in the data field of LSQ and 8.47 in L2 cache. On the contrary, the inaccuracy in the L1D is 20.13 percentile units and in the L1I is 13.74.

Figure 38 summarizes the speedup of the three operation modes. We can observe that speedup scales from *Baseline* mode to *ESO* mode and from *ESO* mode to *ESOR* mode in all cases. The address field of LSQ presents the best scaling and the best speedup among all structures (2.92X in *ESO* mode and 4.06X in *ESOR* mode) and this is justified by the highest *WAI* rate that it features according to Figure 35 and Figure 36. The worst scaling is presented in L2 cache (~1.06% for both *ESO* and *ESOR* modes), but this can be explained by the high percentage of *Unused* category illustrated in Figure 36. In general, all the caches that present a high percentage of the *Unused* category do not speedup so well as the structures with low percentage of *Unused* category.

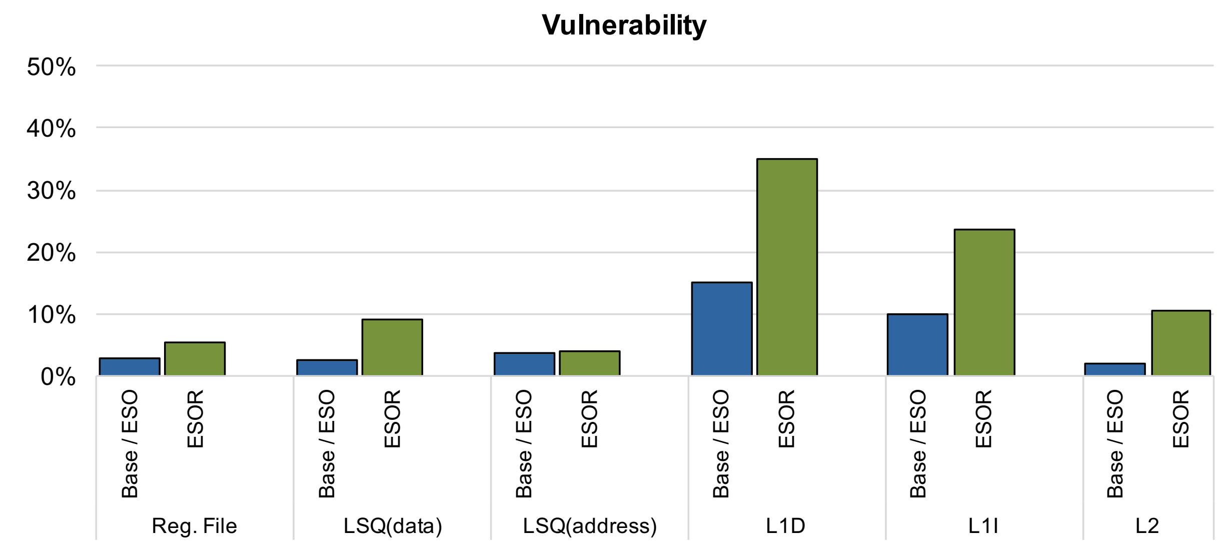


Figure 37: Structures vulnerability reported by the three operation modes. There is no loss of accuracy in the vulnerability reports between the baseline mode and the *ESO* mode, while *ESOR* mode reports higher vulnerability in all cases.

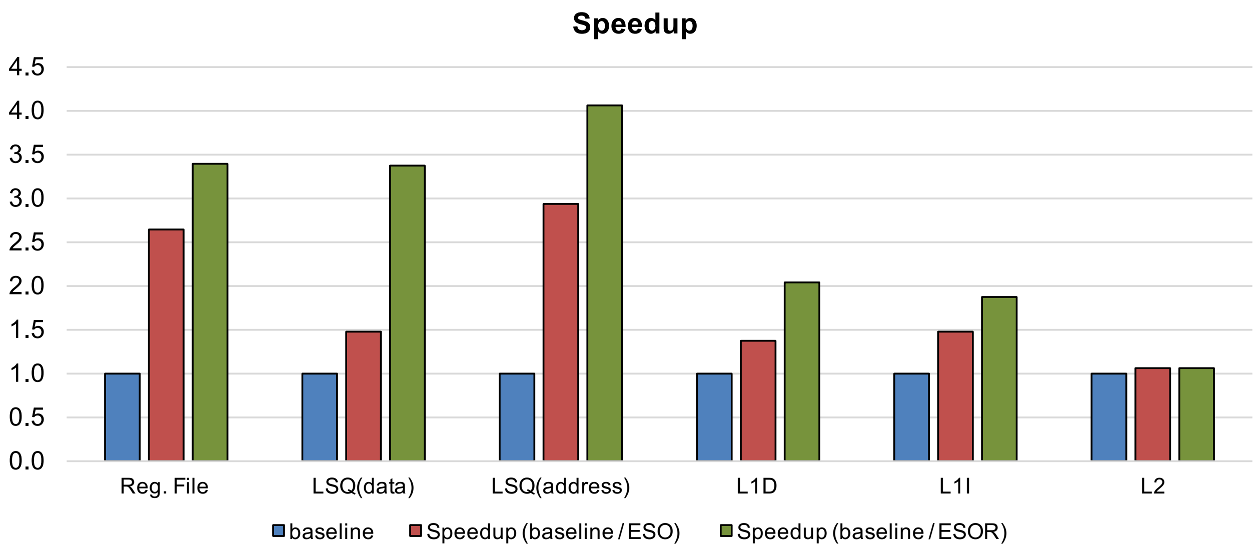


Figure 38: Speedup of the three operation modes of study [31].

Combining the results presented in Figure 37 and Figure 38, we conclude that for the intra core structures (Physical Integer Register File, address and data fields of LSQ), the best solution to speed up the statistical fault injection campaign is the third mode of framework's operation (*ESOR* mode) with negligible loss of estimation accuracy, getting a high speedup of 3.38X, 4.06X and 3.37X respectively. Except for *ESOR* mode, the *ESO* mode could be also used for the same structures without any accuracy loss getting a speedup of 2.63X, 2.92X and 1.46X respectively.

On the other hand, the best choice for an architect to estimate the reliability of caches is the *ESO* mode of framework's operation. This conclusion comes from the fact that the inaccuracy of caches' reliability assessment using *ESOR* mode is not negligible (from 8.47 percentile units for L2 cache to 20.13 units for L1D cache) and the speedup is not as high as in the intra core structures (for instance only 1.06% for L2 cache). Consequently, *ESO* mode is the best choice for caches to speedup campaign (with 1.37X, 1.48X and 1.05X speedup for the L1D, L1I and L2 respectively) and ensure the estimation accuracy.

3. ACCELERATION OF RELIABILITY ASSESSMENTS USING MeRLiN

In Section 2.2.4, we have described techniques to accelerate fault injection campaigns that are implemented after the actual injection of the fault in the hardware structure. In this chapter, we propose MeRLiN[2] methodology that was presented in [32] to accelerate statistical fault injection campaigns by pruning the faults of the initial fault list *before* their actual injection.

Figure 39 reflects the motivation of our MeRLiN methodology compared to the four state-of-the-art methods used for reliability assessments and presented in Section 2.1.4 in terms of speed and measurement accuracy. An ideal method at the top-right corner of the figure would provide the highest speed (equal to that of the ACE analysis and probabilistic models) and the highest accuracy (equal to that of the injection methods with high statistical significance). MeRLiN approaches the ideal method boosting microarchitecture level injection-based reliability assessment while keeping its measurement accuracy unaffected. The backbone of MeRLiN is built on two major observations:

- A large number of faults in a statistical fault injection campaign are over-written before being read or are injected in dead or invalid entries of the hardware structure [31]. These faults can be easily identified and pruned from the initial fault list in a single run. We call this first part of our method *ACE-like*, because it resembles a simple ACE analysis flow.
- The faults that are injected in the same or different entries of a structure during the same or different vulnerable intervals are *very likely* to have the same effect on program execution if these intervals end up to the same static instruction and the same micro-operation (uop) that reads the faulty entry. MeRLiN groups these faults together and performs fault injection on a small number of representatives. While it preserves the accuracy of the reliability measurements, this grouping drastically reduces the number of required injections because instruction repetition is an extensively inherent property of all programs [80] [81] [82] [83].

MeRLiN's contributions that make it the state-of-the-art method to accelerate the statistical fault injection campaigns of high statistical significance at the microarchitecture level without loss of accuracy are the following:

- It accelerates statistical microarchitecture level fault injection from 1 to 3 orders of magnitude. Our experiments with full runs of 10 MiBench benchmarks [74] show 93X, 225X, 68X and 28X speedup on average for different sizes of the register file, the store queue, the first level data cache and the issue queue, respectively. When applied to 10 SPEC CPU2006 benchmarks, MeRLiN reveals larger average speedups of 1644X, 2018X and 171X for the register file, the store queue and the first level data cache, respectively.
- It reports virtually the same reliability estimations as the baseline microarchitectural fault injection with extremely high statistical significance.
- It delivers fine-grained insights of the fault effects (Silent Data Corruptions-SDC, Detected Unrecoverable Errors-DUE, crashes, locks) unlike ACE analysis which only reports a gross AVF estimate. This can be used to evaluate different protection schemes or to identify benchmarks more prone to SDCs [61] [84].

[2] MeRLiN = **M**icroarchitectural **e**valuation of **R**eliability using statistica**L** fault i**N**jection.

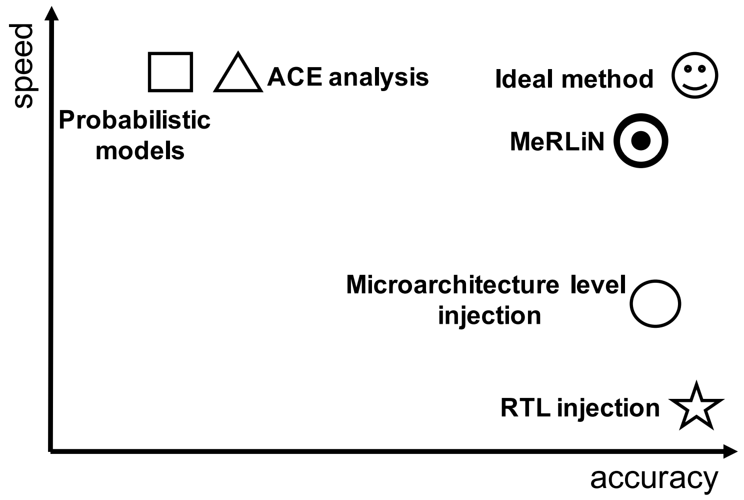


Figure 39: Motivation of MeRLiN methodology compared to the four state-of-the-art methods used for reliability assessments (as described in Section 2.1.4).

MeRLiN methodology consists of three phases: *Preprocessing*, *Fault List Reduction* and *Fault Injection Campaign* as shown in Figure 40. Next, we describe these three phases of MeRLiN methodology in detail:

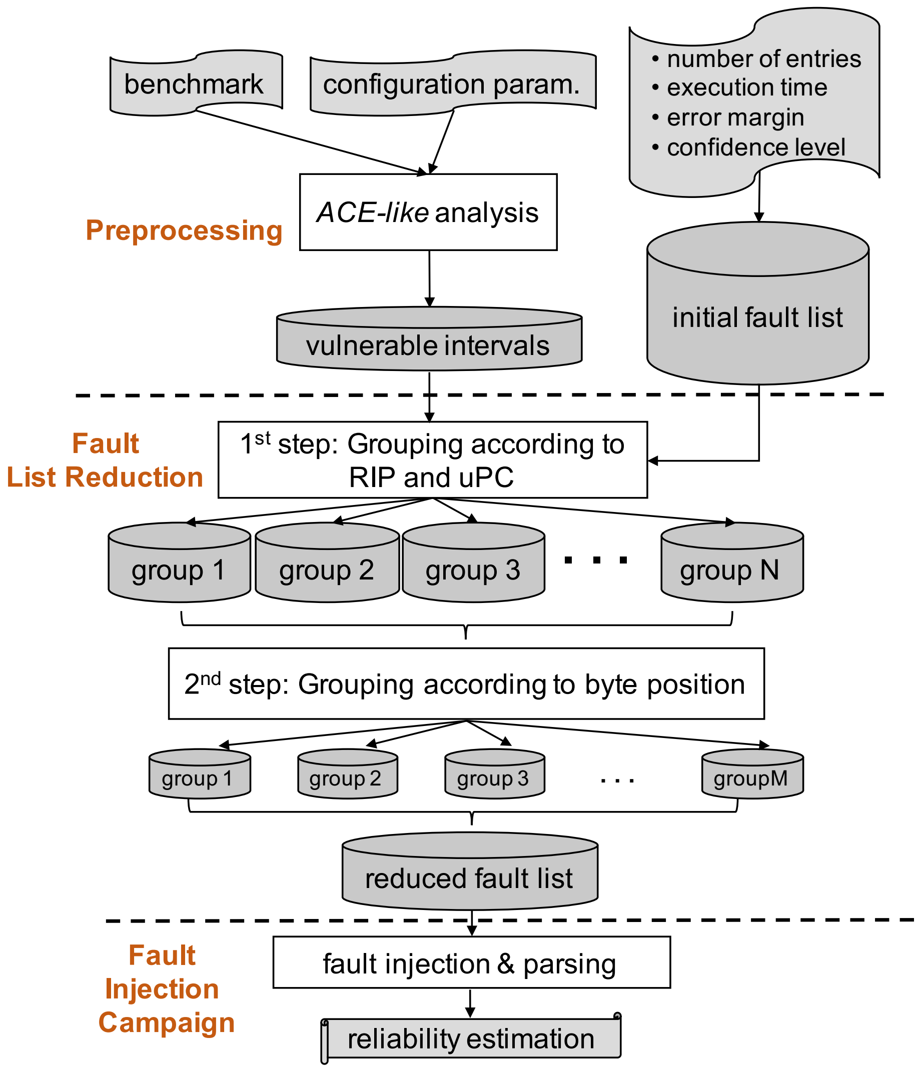


Figure 40: Flowchart of MeRLiN.

- ***Preprocessing:***

  The first phase of MeRLiN methodology includes two tasks. First, MeRLiN records all vulnerable intervals of all entries of a hardware structure during the entire benchmark execution (this task is called *ACE-like* analysis). Then, MeRLiN creates the *initial fault list* repository that consists of a large number of faults for a statistically significant sampling: very low error margin and very high confidence level [62] (this task is called *Initial Fault List Creation*).

  During the *ACE-like* analysis task, the benchmark runs once to completion to profile the vulnerable intervals (in which a bit flip may lead to corruption) of each entry of the target hardware structure (e.g. the registers in a physical register file). For our analysis, a *vulnerable interval* of an entry:

  - Starts with a write operation and ends with a committed read of the same entry;
  - Starts with a committed read and ends with another committed read of the same entry.

This definition differs from the typical definition of ACE intervals [50] [52] (where intermediate reads do not define the end of an interval) but the overall vulnerable time (sum of vulnerable intervals) is the same. Note that, similar to the original ACE analysis wrong-path execution instructions are not considered as part of the vulnerable intervals of MeRLiN. We highlight this difference between the two methods by an example in Figure 41, which represents the lifetime of an entry during the execution of a benchmark. The arrows directed upwards and downwards represent read and write operations, respectively. The read operations at t2, t5 and t6 are finally squashed. MeRLiN divides the interval between t7 and t9 in two individual vulnerable intervals, while ACE analysis considers them as a single interval.

This difference between MeRLiN's first step and classic ACE analysis is very important for the second phase of MeRLiN, where the faults are grouped with respect to the instruction pointer (RIP) and the micro program counter (uPC) of the committed read that accesses the entry at the end of the vulnerable interval. Our analysis requires both the RIP and the uPC to cover cases where an x86-64 instruction consists of different micro-instructions that access the same or different entries of the hardware structure in the same or different cycles. These accesses can lead to different fault effects and are classified separately.

Our *ACE-like* analysis is significantly lighter in terms of storage overhead (10-100MB in our experiments) and more easily implemented than the complete ACE, because it does not trace the transitively dynamically dead (TDD) instructions [50]. The execution time of the *ACE-like* single-run step was less than 5 hours for all our experiments.

At the end of this step, the following information is stored in the *vulnerable intervals* repository for every *ACE-like* vulnerable interval of each entry: (i) start and end of the interval (cycle numbers), (ii) the instruction pointer (RIP) of the static x86-64 instruction that reads an entry at the end of the interval, and (iii) the micro program counter (uPC) of the micro-operation which is part of the x86 instruction and reads an entry at the end of the interval.

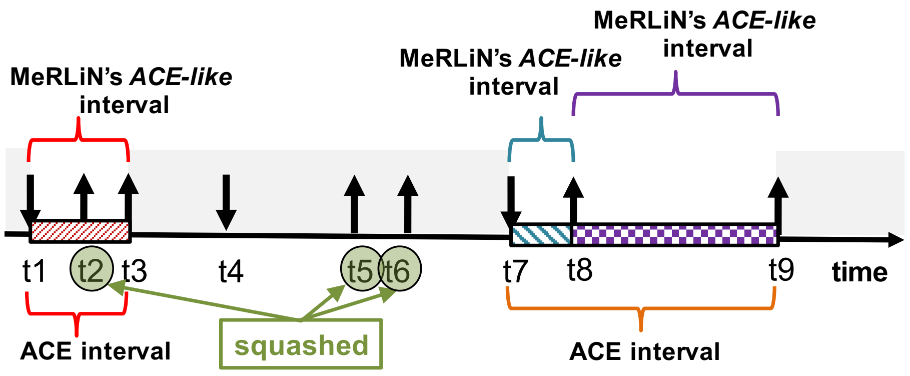

MeRLiN's *ACE-like* interval
MeRLiN's *ACE-like* interval
MeRLiN's *ACE-like* interval
t1
t2
t3
t4
t5
t6
t7
t8
t9
time
ACE interval
squashed
ACE interval

Figure 41: ACE and *ACE-like* intervals definition example.

In the second task of the first phase (the *Initial Fault List Creation* task), MeRLiN creates the *initial fault list* repository according to the statistical sampling described in [62]. The initial faults population is defined by: (1) the size (in bits) of the hardware structure, (2) the total execution time (in cycles) of the benchmark, (3) the statistical confidence level and (4) the statistical error margin. To achieve high statistical significance, the initial fault list should consist of tens or hundreds of thousands of faults. For instance, an injection campaign targeting a 256-entry integer register file of 64-bit registers with error margin 2.88%, confidence level 99% and 100M cycles of program execution time, requires 2000 fault injection runs [62]. If a higher statistical significance is needed (i.e. 0.63% error margin and 99.8% confidence level), the total number of injection runs explodes to 60,000 (an unacceptably large number of injections even for relatively short benchmarks). We use this number of 60K faults to define the baseline injection campaign for each single component, size and benchmark configuration, ensuring the same or even slightly higher statistical significance for all our structures. According to [62], for estimations of high statistical significance the confidence level and the error margin dominate in the calculation of the initial fault list population (see more details in Section 2.1.5).

The outputs of the first phase of MeRLiN are the *vulnerable intervals* repository and the *initial fault list* that feed MeRLiN's second phase (see Figure 40).

- ***<u>Fault List Reduction:</u>***

  This phase of MeRLiN classifies the faults in groups running a two-step grouping algorithm, and creates the *reduced fault list* that is used for the actual injections.

  During the execution of the first step of the algorithm, all faults of the *initial fault list* are examined. All faults that target a non-vulnerable interval are directly classified as *Masked* as no injection is needed for them. The remaining faults that hit *ACE-like* vulnerable intervals are stored in different subdirectories (see Figure 40) according to the RIP and the uPC of the instruction that reads the entry at the end of the interval. Each of the created groups consists of transient faults on the same or different entries of the hardware structure being analyzed, during the same or different *ACE-like* vulnerable intervals that are read by an instruction with the *same* RIP and the *same* uPC.

  Figure 42 shows an informative example of this first step for three entries of a hardware component during the execution of the same benchmark. When this step finishes, four groups are created containing faults that hit different hardware entries at different time intervals. The faults with the same color belong to the same group. The faults belonging to non-vulnerable intervals (gray color) are characterized as *Masked*. For instance, the faults in intervals t4-t6, t10-t13 and t7-t11 are grouped together (red color), because these intervals end up to micro-instructions with the same *ripC* and *uPC3*.

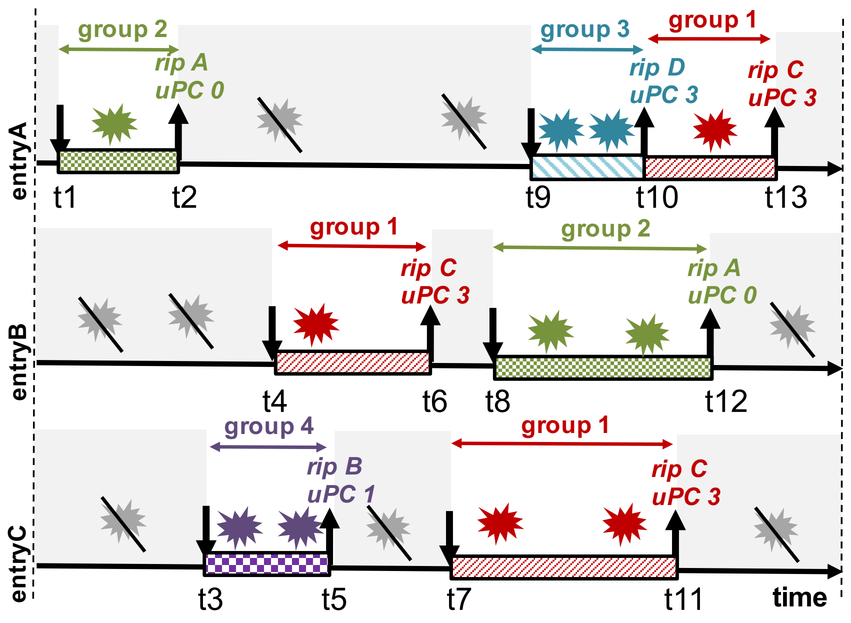


Figure 42: 1st step example of the grouping algorithm.

Due to logical masking, all bits in a given faulty entry may not have the same effect when read by an instruction. To maximize MeRLiN's accuracy, especially for groups with hundreds of faults, we select more than one fault for the actual fault injection runs in cases that faults hit a different byte of the entry. Moreover, faults in different bytes are selected from *different* dynamic instances of the same static instruction to increase time diversity. This can be further extended to separate faults hitting different *nibbles* or *bits*, but our experiments verify that this is not necessary.

MeRLiN ensures that for static instructions that are correlated with large population of faults, several representatives are selected from different dynamic instances of the same instruction, covering all possible byte positions of different entries. This per byte selection leads to smaller final groups ensuring the statistical significance of MeRLiN (see the theoretical analysis in Section 3.4), while it leads to groups of faults that are *extremely likely* to have the same effect. Figure 43 shows an example of the second step of the algorithm for three different hardware entries (*K*, *L*, *M*) during the execution of a benchmark. Note that all these faults were classified in the same group (same rip=*F* and uPC=4) from the first step of the grouping algorithm. The number next to each fault corresponds to the group in which the fault is finally classified at the end of the second step; the faults in circles are stored in the *reduced fault list* repository and are the *only* ones that will be injected. The execution time of the entire MeRLiN's single-run *group creation algorithm* was less than 50 minutes for all our experiments.

At the end of this phase, the *reduced fault list* repository contains all the selected faults. Only these faults are injected using the microarchitecture level fault injector.

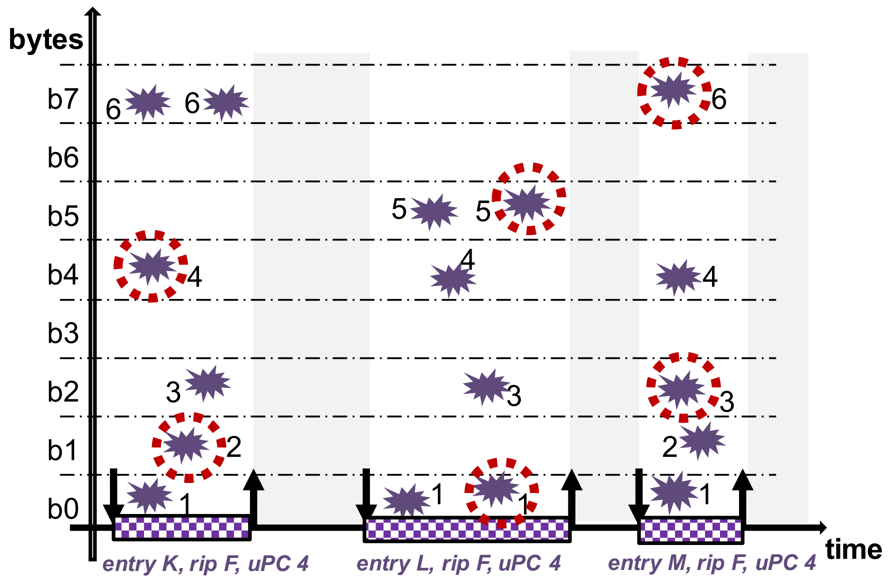


Figure 43: 2nd step example of the grouping algorithm.

- ***Fault Injection Campaign:***

  In the last phase of MeRLiN, the *fault injection* campaign is launched using all faults of the *reduced fault list* repository. During the *parsing* step, the outputs of all the injection runs per reduced group are compared to that of the golden run to identify the fault effect and calculate the final *reliability estimation* of the structure.

To evaluate MeRLiN methodology, we employed GeFIN [29] [30] microarchitectural injector and extend it to implement and evaluate MeRLiN on four structures of an x86-64 out-of-order processor covering both data- and instruction-related structures:

- The physical integer Register File (RF) for three sizes: 256, 128, 64 registers.
- The data field of the Store Queue (SQ) of the Load/Store Queue for three sizes: 64 load and 64 store, 32 load and 32 store, and 16 load and 16 store entries. Gem5 doesn't implement data fields in the Load Queue.
- The data field of L1 data cache (L1D) for three sizes: 64KB, 32KB and 16KB.
- The destination register of the Issue Queue (IQ) for two sizes: 32 and 60 queue entries.

The reason of choosing GeFIN fault injector instead of MaFIN is that GeFIN provides deterministic simulation runs; a feature that is necessary for the ACE-like analysis task of MeRLiN. On the other hand, MaFIN fault injector tool uses QEMU which on one side leads to non-deterministic behavior between two consecutive simulation runs, but on the other side resembles more accurately the full-system stack.

MeRLiN can be also used for: (i) all hardware structures of the CPU (caches, buffers, queues, registers, etc.), (ii) different input sets and benchmarks, (iii) different architectures and ISAs. Table 19 shows the baseline microprocessor configuration that was used for all the injection campaigns that were launched to evaluate MeRLiN methodology targeting the RF, the SQ, the L1D and the IQ with 32 entries. For all the injection campaigns that evaluate MeRLiN targeting the IQ of 60 entries, we used a different configuration that resembles the Intel Haswell-like microprocessor; the major features of this configuration are presented in Table 20.

Table 19: Baseline configuration used to evaluate MeRLiN on RF, SQ, L1D and IQ with 32 entries.

Parameter	x86 microprocessor model
Pipeline	OoO
Physical register file	**256/128/64 int**; 192 FP
Issue Queue entries	**32**
Load/Store Queue	**64/32/16 load & 64/32/16 store entries**
ROB entries	100
Functional units	6 int ALUs; 2 complex int ALUs; 4 FP ALUs, 2 FP mul/div, 4 SIMD
L1 Instruction Cache	32KB,64B line,128 sets,4-way, write back
L1 Data Cache	**16KB/32KB/64KB, 64B line,64/128/256 sets,4-ways, write back**
L2 Cache	1MB,64B line,1024 sets,16-way, write back
Branch Predictor	Tournament predictor
Branch Target Buffer	conditional and unconditional branches BTB (direct-mapped, 4K entries)

Table 20: Haswell-like configuration used to evaluate MeRLiN on IQ with 60 entries.

Parameter	Intel Haswell-like microprocessor model
Pipeline	OoO
Physical register file	168 int; 168 FP
Issue Queue entries	**60**
Load/Store Queue	72 load & 46 store entries
ROB entries	192
Functional units	6 int ALUs; 2 complex int ALUs; 4 FP ALUs, 2 FP mul/div, 4 SIMD
L1 Instruction Cache	32KB,64B line,64 sets,8-ways, write back
L1 Data Cache	32KB,64B line,64 sets,8-ways, write back
L2 Cache	256KB,64B line,512 sets,8-ways, write back
Branch Predictor	Tournament predictor
Branch Target Buffer	conditional and unconditional branches BTB (direct-mapped, 4K entries)

For all the experiments, we used machines with Intel Core i7-4771 at 3.5GHz, 16GBytes of RAM at 1600MHz and 1TByte hard disk. To classify the fault effects, we used the six categories (Masked, SDC, DUE, Timeout, Crash, Assert) that were presented in Table 4. Moreover, the *initial fault list* for each campaign of our evaluation was generated using statistical fault sampling [62] (see Section 2.1.5 for more details) and consists of 60,000 faults that correspond to 99.8% confidence level and 0.63% error margin. To study the scalability of MeRLiN (see experimental results), we increased the initial fault list to 600,000 faults that corresponds to 99.8% confidence level and 0.19% error margin.

For the evaluation, we used 10 benchmarks from the MiBench suite [74] and 10 from the SPEC CPU2006 suite [75]. We ran the MiBench benchmarks to the end to evaluate both MeRLiN's accuracy and speedup. Their execution time ranges from 1 to 55 million cycles, while they are very similar in instruction mixes and throughput with SPECs. In the case of SPEC benchmarks, we evaluate MeRLiN running Simpoint samples of

100M committed instructions with the largest weight [85]. MeRLiN's purpose is not to propose new benchmark intervals sampling approach for reliability evaluation, but any existing approach can be used (e.g. [86] for large caches or Simpoints that were used in many reliability studies [50] [52] [54]).

We selected to evaluate MeRLiN's accuracy executing MiBench benchmarks till the end instead of running entire SPEC benchmarks, because the execution time of each baseline comprehensive injection campaign (60,000 faults for each entire SPEC program, component and configuration) would make the evaluation infeasible. Also, we evaluated the accuracy of MeRLiN at the end of the Simpoint intervals of two selected SPEC CPU2006 benchmarks (bzip2 and gcc).

Next, we present the results of MeRLiN's evaluation when we target either data-related (RF, SQ and L1D) or instruction-related (IQ) structures in Section 3.1 and Section 3.2, respectively. We also present a detailed comparison of MeRLiN methodology with other architecture level fault injection approaches in which the faults are injected at the software level (Section 3.3) and finally in Section 3.4, we present a theoretical analysis of MeRLiN approach.

3.1 MeRLiN's results on data-related structures

In this subsection, we present all the results of MeRLiN's evaluation when we target data-related structures. In our case these structures are: (i) the physical integer register file (RF) with 256, 128, and 64 registers, (ii) the Store Queue (SQ) with 64 load and 64 store, 32 load and 32 store, and 16 load and 16 store entries, and (iii) the L1 Data cache (L1D) for three different sizes: 64KB, 32KB and 16KB.

First, to measure the effectiveness of our grouping algorithm we define the *homogeneity* metric. In equation (10), *N* is the number of the groups that MeRLiN generates and *#faults* is the number of faults of a group. The *dominant* class of a group is defined as the category among those of Table 4 that contains the largest number of faults in the group. Thus, *dominant_class%* is the percentage of faults of the group that are classified in the dominant class. When *dominant_class%* equals 100%, it means that all the faults in that group have the same fault effect. Finally, *#total_faults* is the total population of faults that hit vulnerable intervals. Large values of *homogeneity* close to 1.0, denote that the vast majority of faults across all groups lead to the same effect, and the accuracy of the algorithm is high.

$$\textit{homogeneity} = \frac{\sum_{\text{group1}}^{\text{groupN}} \text{\#faults} \times \text{dominant\_class\%}}{\text{\#total\_faults} \times 100\%} \qquad (10)$$

Figure 44, Figure 45, and Figure 46 show the *homogeneity* of the RF, the SQ and the L1D respectively, for all our experiments running the 10 MiBench. On the average, the highest *homogeneity* for the RF is 0.940, for the SQ is 0.982 and for L1D is 0.920. In general, the homogeneity values are very high for this fine-grained classification (the 6 classes). If homogeneity is calculated in coarser granularity (masked vs. not-masked faults) and all classes that lead to non-masking are combined together, then homogeneity is even larger; see the values at the top of each bar in Figure 47. The value at the bottom of each bar represents the percentage of groups (average for all our experiments with MiBench) that consist of faults with *exactly* the same effect (masked, non-masked) meaning that they have a perfect homogeneity value of 1.0. Finally, homogeneity climbs to 0.99 if we count the faults excluded by the ACE-like, but here we focus only on MeRLiN's grouping part. All these results indicate the extremely high accuracy of MeRLiN methodology.

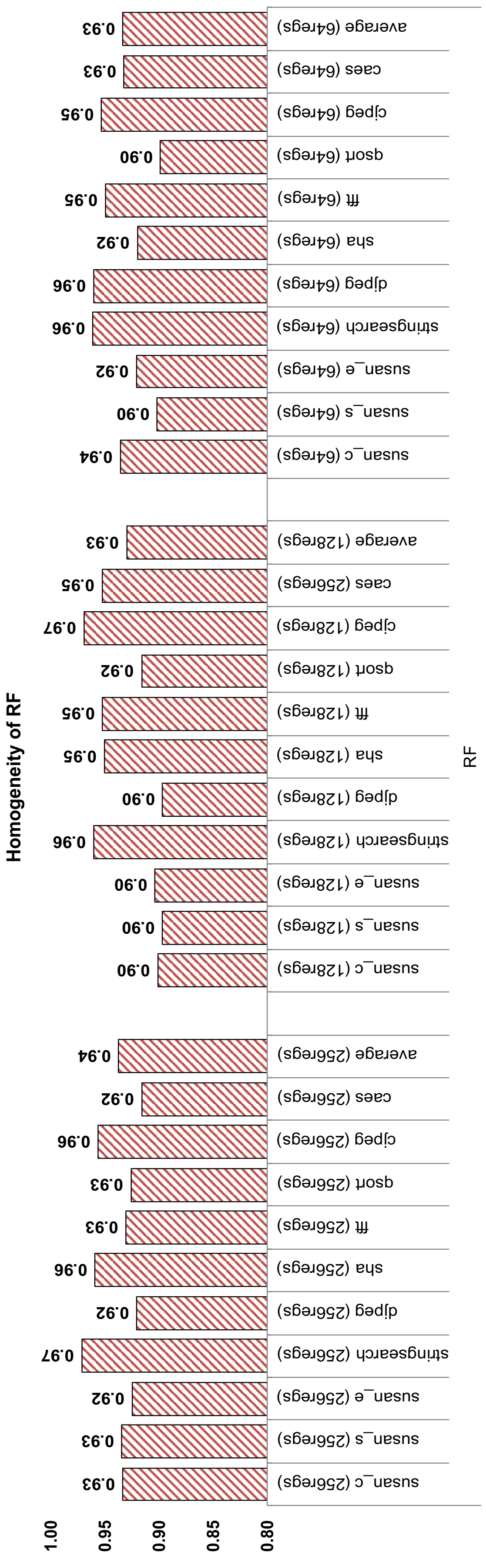


Figure 44: Homogeneity of Physical Register File.

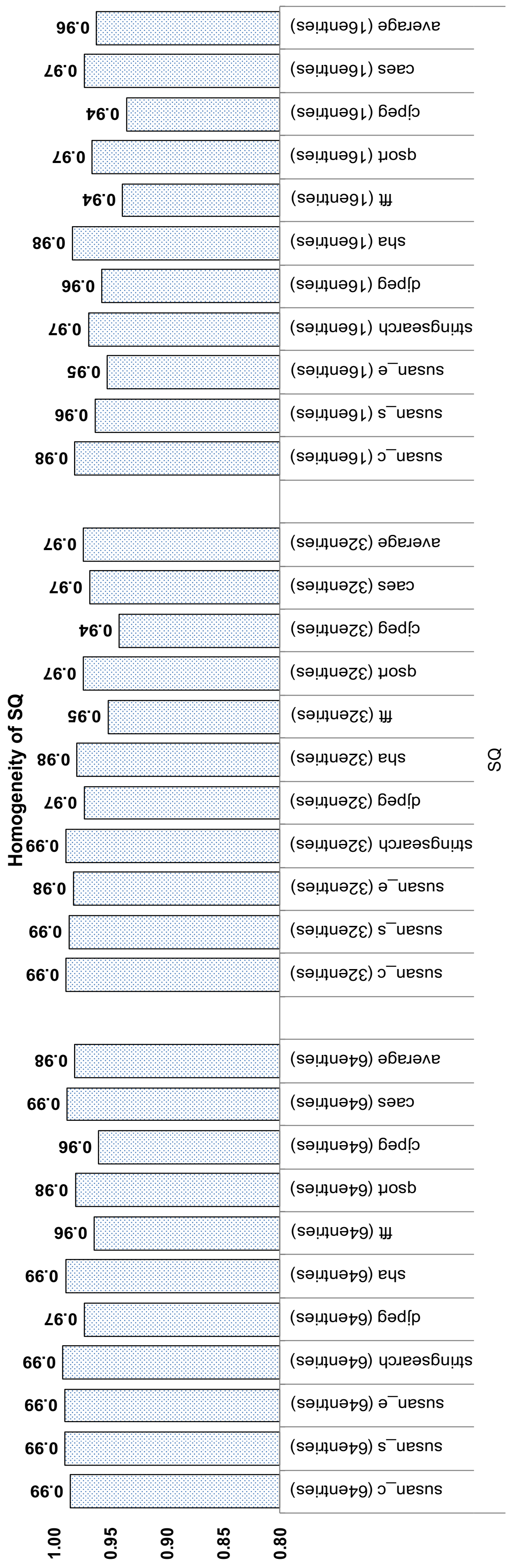


Figure 45: Homogeneity of Store Queue.

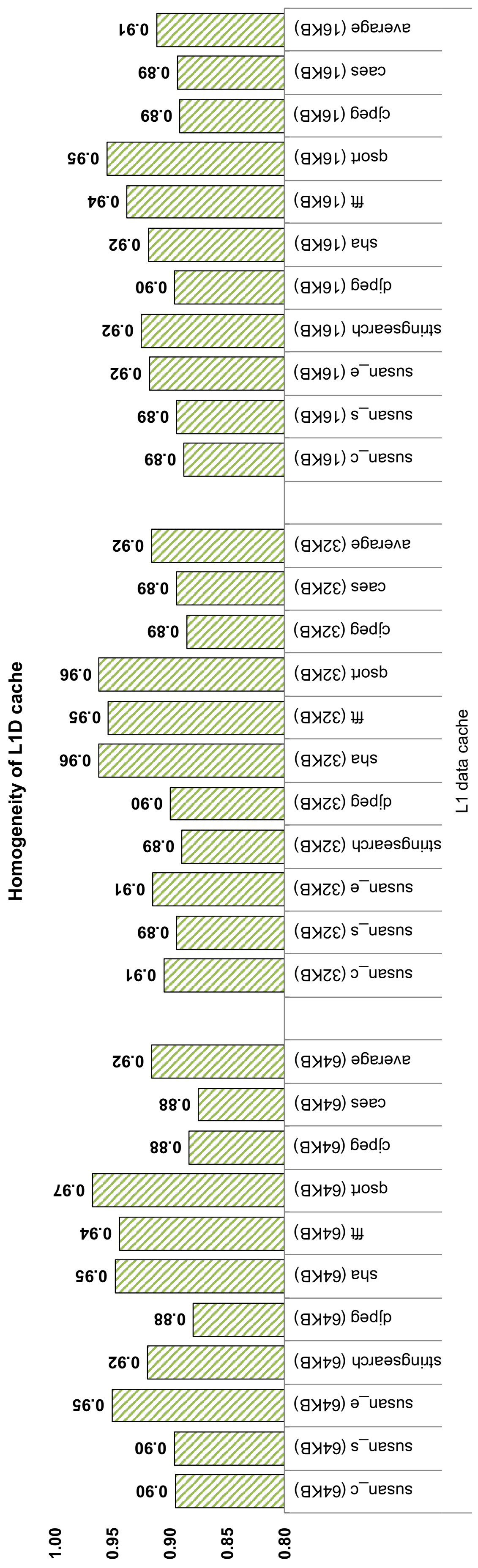


Figure 46: Homogeneity of L1 Data cache.

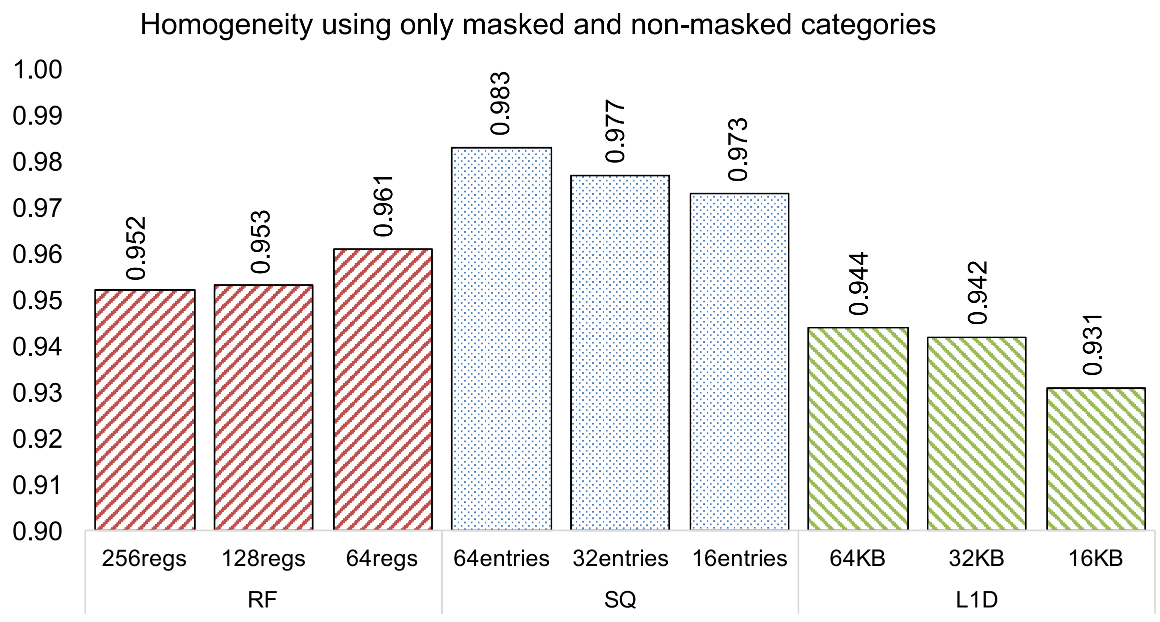


Figure 47: Coarse-grained homogeneity (number on top of the bars) and percentage of groups with perfect homogeneity that is equal to 1.0 (number on the bottom of the bars); average for 10 MiBench.

We measure the accuracy of the reliability estimations of MeRLiN for the three components running 10 MiBench benchmarks till the end. We compare MeRLiN's accuracy against the injection in: (i) the remaining fault list after the exclusion of the faults that target non-vulnerable intervals (identified by the *ACE-like* step of the method), (ii) the comprehensive baseline fault list (60,000 faults). Finally, we evaluate MeRLiN’s accuracy for the RF with 60K faults using Simpoints from the bzip2 and the gcc.

The estimation accuracy of MeRLiN for the physical register file, the store queue and the L1 Data cache against the injection using the remaining fault list *after* the *ACE-like* step is shown in Figure 48, Figure 49, and Figure 50 respectively. Each graph shows the average fault effect classification across the 10 MiBench benchmarks used in our study for the three configurations of each structure. The first bar (blue) in each class corresponds to the results of the fault injection in the remaining fault list *after* the *ACE-like* analysis, while the second bar (red) illustrates the results on the same fault list after applying MeRLiN’s grouping algorithm and injecting only the selected faults. The values on top of each bar represent the measurement per fault effect category. Similar behavior is observed across all benchmarks. For all component configurations, MeRLiN reports negligible differences compared to the injection using all the faults that hit only vulnerable intervals.

Figure 51, Figure 52, and Figure 53 show the bigger picture for MeRLiN’s accuracy for the three data-related structures (RF, SQ and L1D), in which the final fault effect classification of the comprehensive baseline fault injection of 60,000 faults (blue bar) is compared to the final classification of MeRLiN (red bar). Each bar represents the average values across the 10 MiBench benchmarks. Similar behavior is observed across all benchmarks. MeRLiN for all cases is extremely accurate and delivers virtually the same reports with the comprehensive injection, but in orders of magnitude faster.

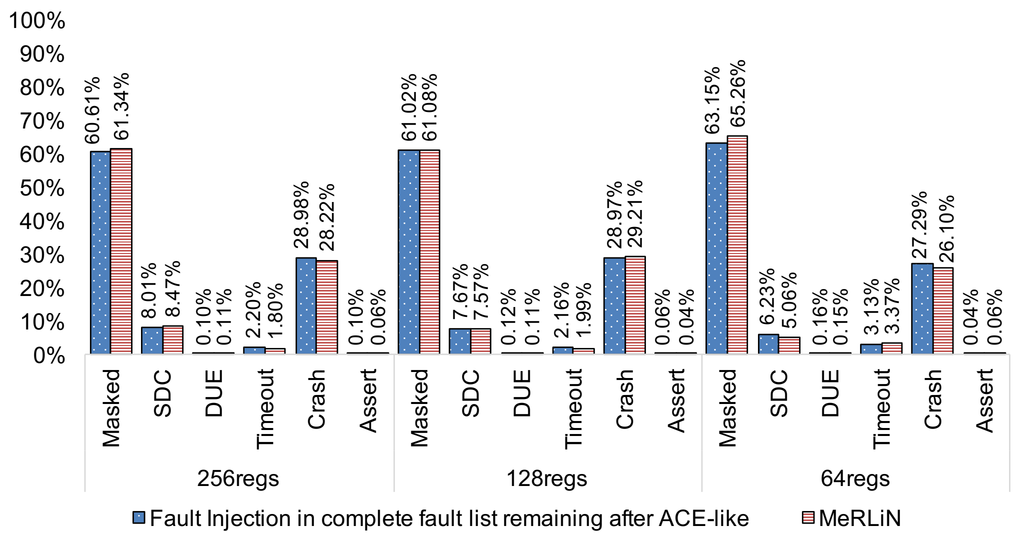


Figure 48: Fault effect classification of MeRLiN against injection with the remaining faults after *ACE-like* step for the Physical Integer Register File; average for 10 MiBench benchmarks.

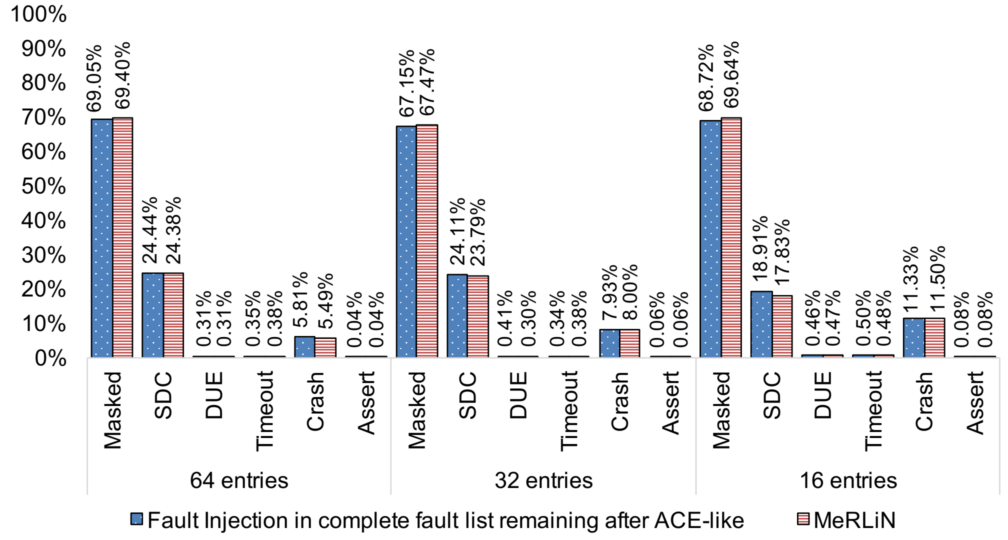


Figure 49: Fault effect classification of MeRLiN against injection with the remaining faults after *ACE-like* step for the Store Queue; average for 10 MiBench benchmarks.

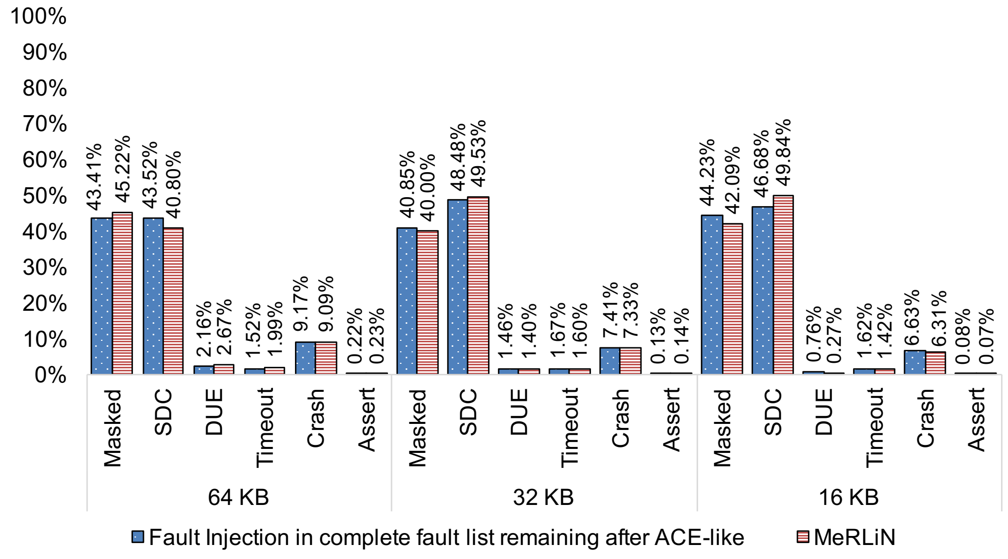


Figure 50: Fault effect classification of MeRLiN against injection with the remaining faults after *ACE-like* step for the L1 Data cache; average for 10 MiBench benchmarks.

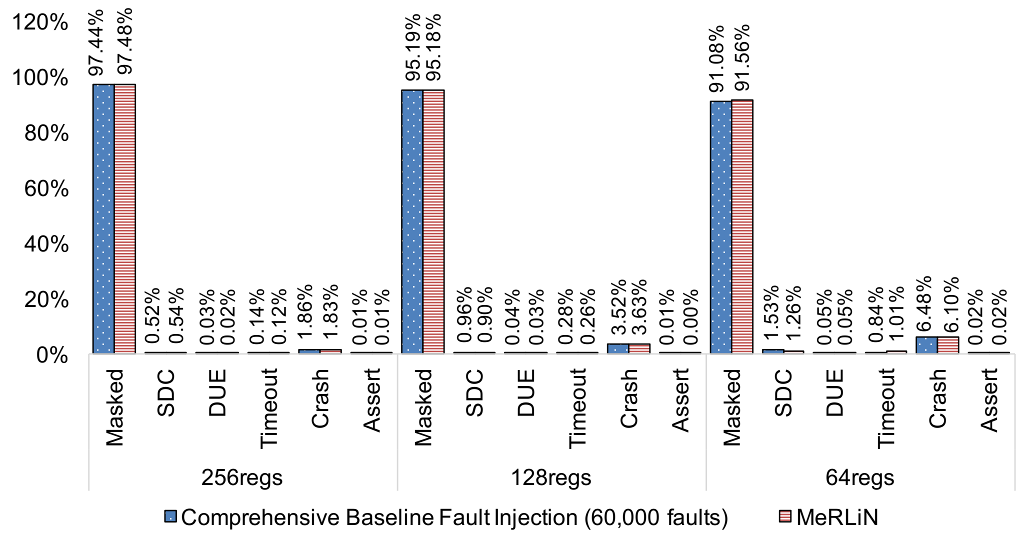


Figure 51: Final fault effect classification of MeRLiN against comprehensive baseline fault injection with 60,000 faults for the Physical Register File; average for 10 MiBench benchmarks.

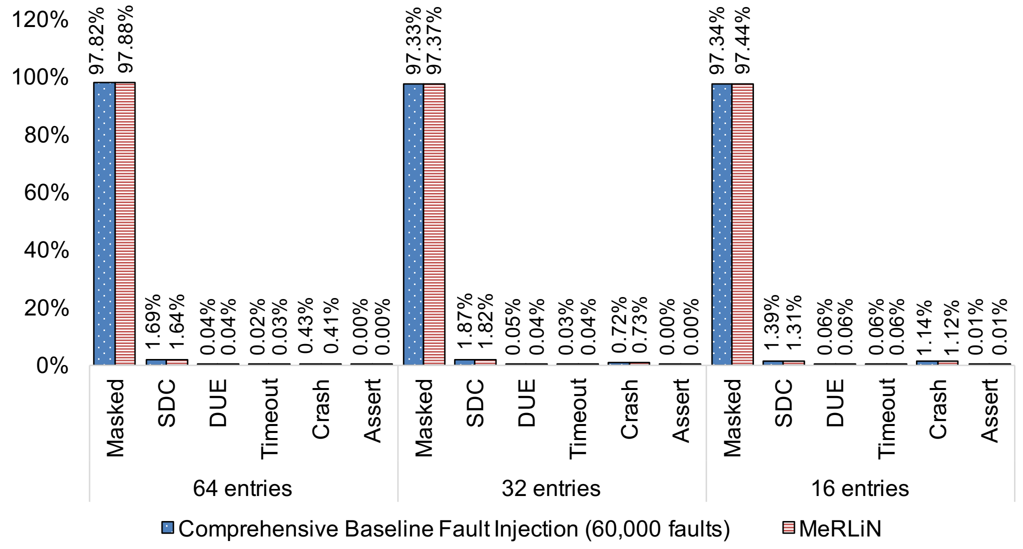


Figure 52: Final fault effect classification of MeRLiN against comprehensive baseline fault injection with 60,000 faults for the Store Queue; average for 10 MiBench benchmarks.

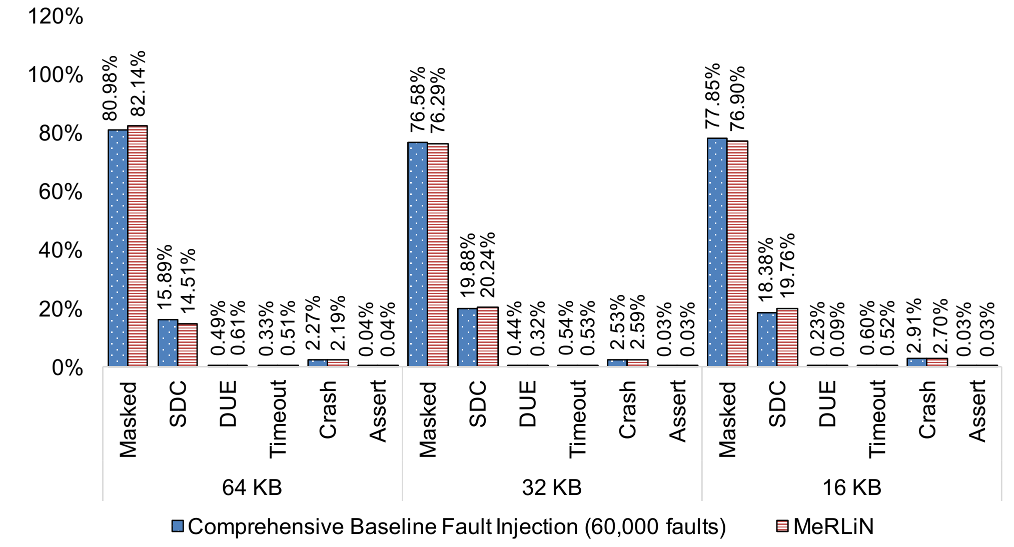


Figure 53: Final fault effect classification of MeRLiN against comprehensive baseline fault injection with 60,000 faults for the L1 Data cache; average for 10 MiBench benchmarks.

Figure 54 demonstrates the final reliability estimation in Failures-in-Time (FIT) rates for the comprehensive baseline campaign (60,000 faults), the MeRLiN method and the *ACE-like* method running the 10 MiBench benchmarks to the end. The reported FIT rates are the products of AVF, raw FIT rate and number of structure's bits. The AVF of the injection-based methods is the ratio of the non-masked injections over the total injections, while the AVF of the *ACE-like* is measured as in [50]. Any raw FIT rate can be used; we use 0.01 FIT per bit. MeRLiN reports negligible differences compared to the comprehensive baseline injection, while the *ACE-like* delivers a pessimistic lower bound of structures' reliability.

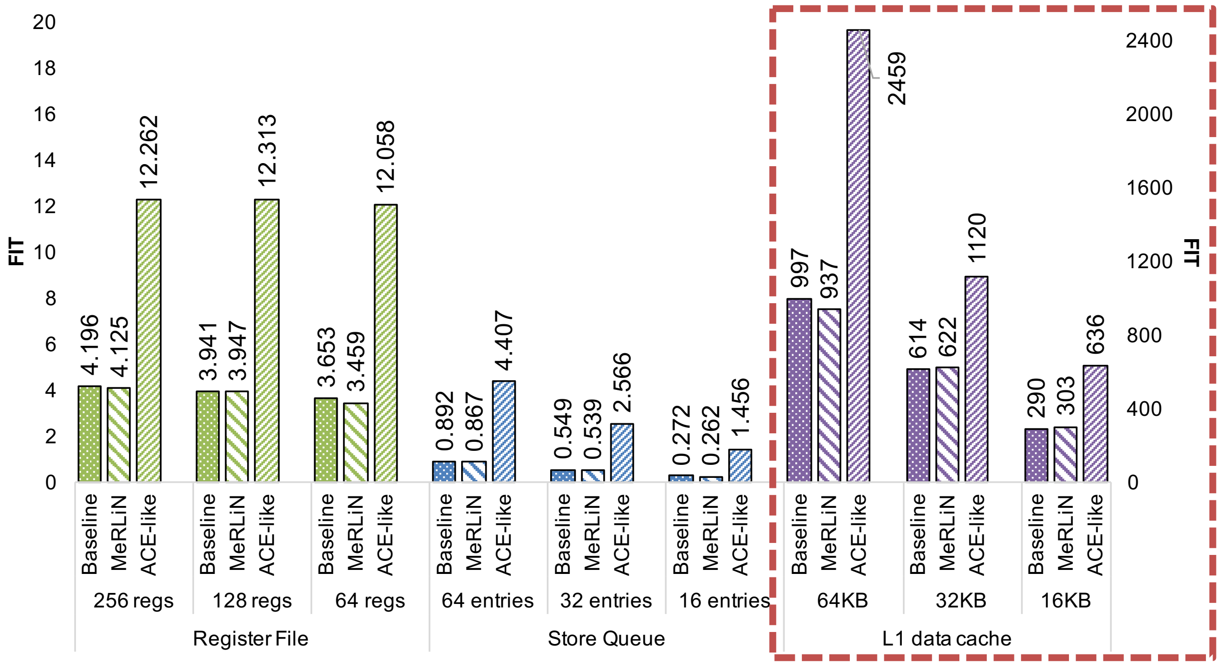


Figure 54: Final reliability assessment (FIT) for Integer Physical Register File, Store Queue, and L1 Data cache (average for 10 MiBench benchmarks).

The evaluation of MeRLiN's accuracy for SPEC CPU2006 benchmarks executed until the end in detailed microarchitectural simulation mode is infeasible as was discussed in Section 2.1.5. To overcome this difficulty and in order to evaluate the accuracy that MeRLiN provides for SPEC CPU2006 benchmarks, we applied MeRLiN injecting faults in the physical register file for the gcc and bzip2 benchmarks and terminating the fault injection runs at the end of the Simpoint interval. The configuration for these experiments is the one of Table 19 with 128 physical registers, 16 store and 16 load queue entries and a 32KB L1 data cache.

As we do not execute the fault injection runs to the end, we are not able to identify SDCs, timeouts or any other abnormal behavior after the end of the Simpoint interval. Thus, only for these experiments we used a different fault effect classification than the classification presented in Table 4. The classification consists of the following categories: (i) Masked; indicates a fault that was not over-written or hit a non-vulnerable interval without affecting program execution, (ii) DUE (as in Table 4), (iii) Crash (as in Table 4), (iv) Assert (as in Table 4), and (v) Unknown; indicates a fault that still exists but at the end of the Simpoint interval it is not known if it will eventually be classified in one of the previous classes or if it will lead to an abnormal behavior.

Table 21 summarizes our measurements per fault effect category using MeRLiN and the comprehensive baseline fault list of 60K faults for the two benchmarks. In both cases, MeRLiN delivers very accurate results per fault effect category compared to the comprehensive baseline method, while the maximum inaccuracy that was observed is only 1.11 percentile points for the Unknown category of the bzip2 benchmark.

Table 21: MeRLiN's accuracy for gcc and bzip2 benchmarks.

Category	gcc (MeRLiN)	gcc (baseline 60K faults)	bzip2 (MeRLiN)	bzip2 (baseline 60K faults)
Masked	85.08%	85.08%	84.98%	84.98%
DUE	0.06%	0.07%	0.29%	0.81%
Crash	3.67%	3.13%	3.50%	4.10%
Assert	0.01%	0.01%	0.03%	0.02%
Unknown	11.18%	11.71%	11.20%	10.09%

Next, we evaluate the speedup of MeRLiN for three data-related targeted structures that are commonly used in reliability assessment studies (RF, SQ and L1D cache) against the comprehensive baseline fault injection campaigns (60,000 faults). Figure 55 presents the speedup of the method for 256, 128 and 64 physical registers for the 10 MiBench benchmarks. The lower (blue) segment and the value on top of it indicate the speedup compared to the comprehensive baseline injection method (60,000 faults) after the first *ACE-like* pass. The higher (red) segment of each bar indicates the speedup achieved by the grouping algorithm on top of the first *ACE-like* step. The value on top of the red bar represents the final speedup achieved by MeRLiN. For example, for 64 registers and the *qsort* benchmark the *ACE-like* step reduces the initial fault list by 4.1X (60,000/14,757). The remaining 14,757 faults are further reduced by the grouping algorithm to 1126 faults that should be actually injected; this totally corresponds to 53.3X (60,000/1126) reduction of the initial fault list. The average speedups are 93.1X, 62.1X and 43.7X for 256, 128 and 64 registers, respectively. Similarly, Figure 56 and Figure 57 present the speedup for the store queue and the data cache, respectively. The average speedups for the store queue are 224.9X, 186.7X and 146.9X for 64, 32 and 16 entries respectively, while for the data cache they are 67.9X, 61.6X and 59.0X for 64KB, 32KB and 16KB respectively.

To evaluate the efficiency of MeRLiN in terms of speedup in larger benchmarks, we ran Simpoint samples of 100M committed instructions with the highest weight from 10 selected integer benchmarks of the SPEC CPU2006 suite assuming an initial fault list of 60,000 faults. We used the configuration of Table 19 with 128 physical integer registers, 16 store and 16 load queue entries and a 32KB L1 data cache. The results of the speedup that MeRLiN delivers are reported in Figure 58. MeRLiN leads to very high final speedups of 1644X, 2018X and 171X on average for the RF, the SQ and the L1D cache, respectively, which are higher than the speedups obtained for MiBench programs since the Simpoint samples we used for SPECs correspond to the most representative part of their execution.

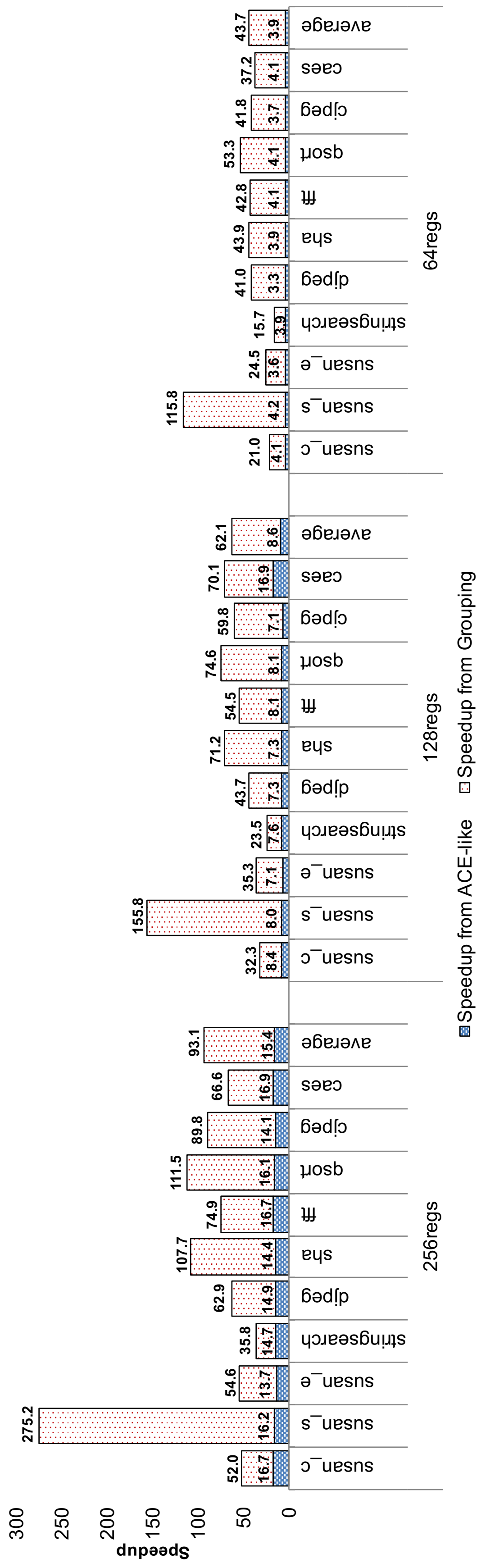


Figure 55: MeRLiN speedup for the three sizes of the Physical Integer Register File running 10 MiBench benchmarks.

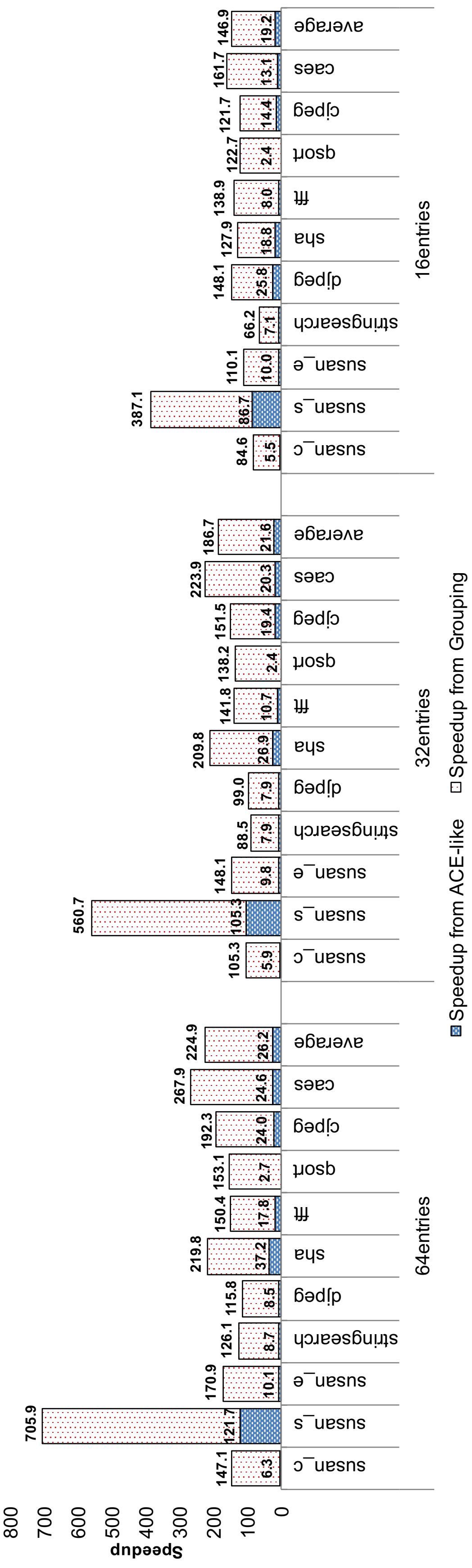
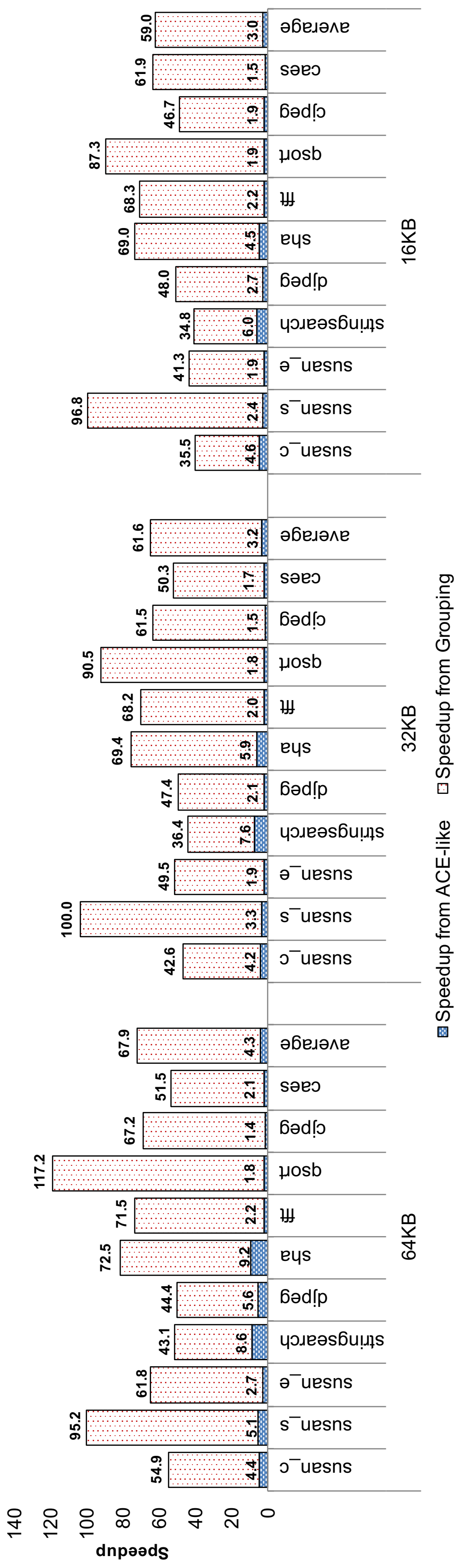


Figure 56: MeRLiN speedup for the three sizes of the Store Queue running 10 MiBench benchmarks.



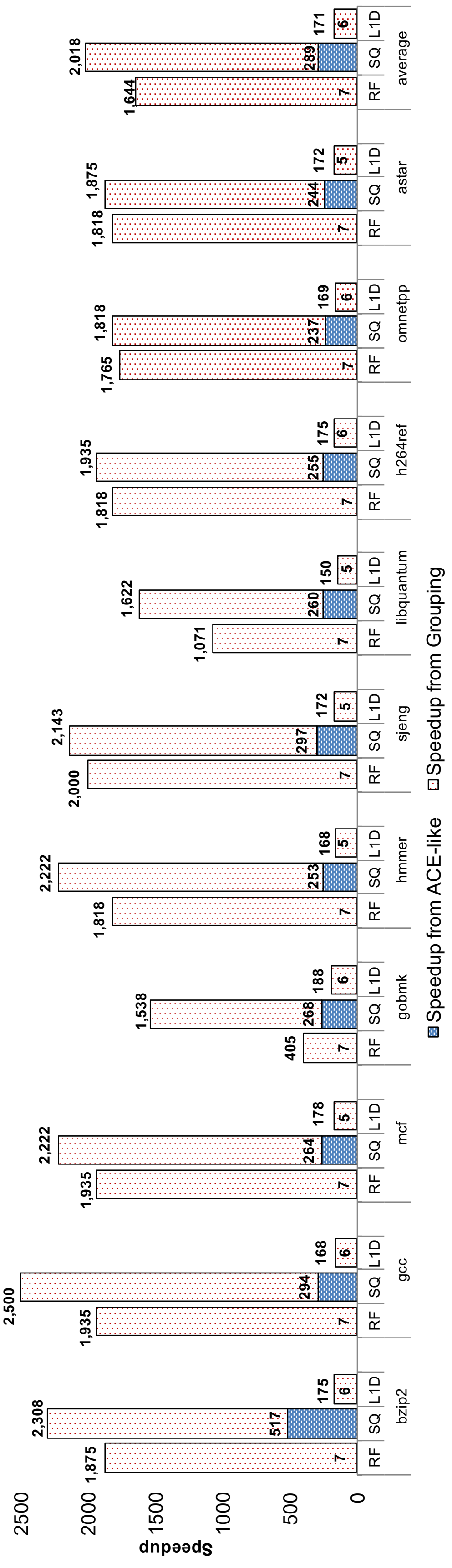


Figure 57: MeRLiN speedup for the three sizes of the L1 Data cache running 10 MiBench benchmarks.

Figure 58: MeRLiN speedup for the RF, SQ, and L1D running 10 Simpoints of 100M committed instructions from SPEC CPU2006.

Figure 59 depicts the actual time (in months) required for the fault injection campaigns in the three structures (RF, SQ and L1D) with the comprehensive fault injection method (60,000 faults per campaign; blue bars) and MeRLiN method (red bars) for all MiBench benchmarks and all component configurations. We assume that all injections run sequentially in the same machine.

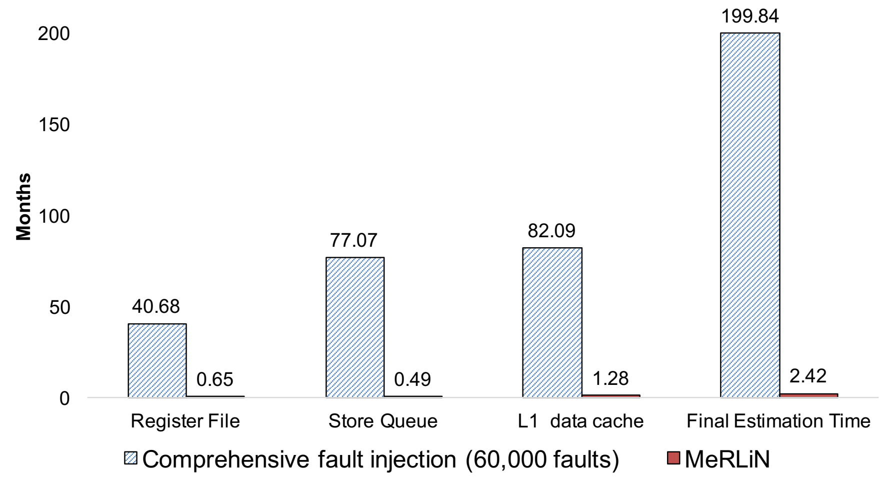


Figure 59: Actual reliability estimation times of the comprehensive baseline injection vs. MeRLiN for all structures configurations of this study running 10 MiBench benchmarks.

The higher the statistical significance of the initial fault list the larger the speedup that MeRLiN offers. In our initial set of campaigns, we ran all MiBench benchmarks using 60,000 faults per campaign (99.8% confidence level and 0.63% error margin). To stress MeRLiN even further, we repeated all these campaigns using a huge 10 times larger initial list of 600,000 faults (99.8% confidence level and 0.19% error margin). Figure 60 presents the average speedup achieved for these two sets of campaigns by the ACE-like (lower purple segment of each bar) and the grouping step (upper white segment) of MeRLiN, as well as the final speedup achieved (value on top of each bar) for each configuration. The final speedup was scaled up 3.46 times on average; practically meaning that for 10 times increase of the initial fault list, MeRLiN finally applies only 2.89 times more faults.

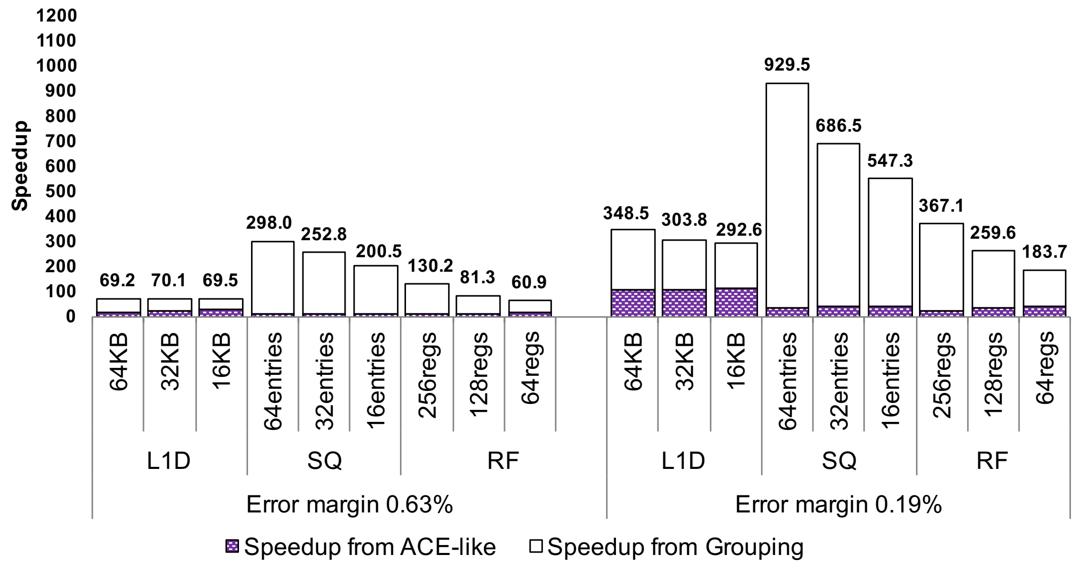


Figure 60: MeRLiN speedup scaling for 0.63% (60K faults) and 0.19% error margin (600K faults); average for 10 MiBench benchmarks.

3.2 MeRLiN's results on instruction-related structures

Apart from the data-related structures (RF, SQ and L1D) that were targeted to evaluate MeRLiN methodology, we also evaluated MeRLiN on an instruction-related structure, the destination register field of the Issue Queue (IQ) with two different sizes: (a) 32 entries using the configuration of Table 19, and (b) 60 entries using the configuration of Table 20 that resembles an Intel Haswell-like microprocessor.

The evaluation of MeRLiN on the Issue Queue has fundamental differences compared to the other data-related structures. The reason is that a fault in this structure can only affect the indexing of the instructions that reside in the pipeline of an OoO processor; this is not the case for the L1D, the SQ and the RF where the data integrity is mainly affected.

In Figure 61 and Figure 62, we illustrate the results concerning the speedup when MeRLiN is applied on an IQ with 32 and 60 entries, respectively. The blue bars and the values on top of them indicate the speedup compared to the comprehensive baseline injection method (60,000 faults) after the first *ACE-like* pass. The red bars and the numbers on top of them indicate the final speedup achieved by MeRLiN. The average speedup that MeRLiN finally provides is 22.2X and 27.5X for 32 and 60 entries, respectively.

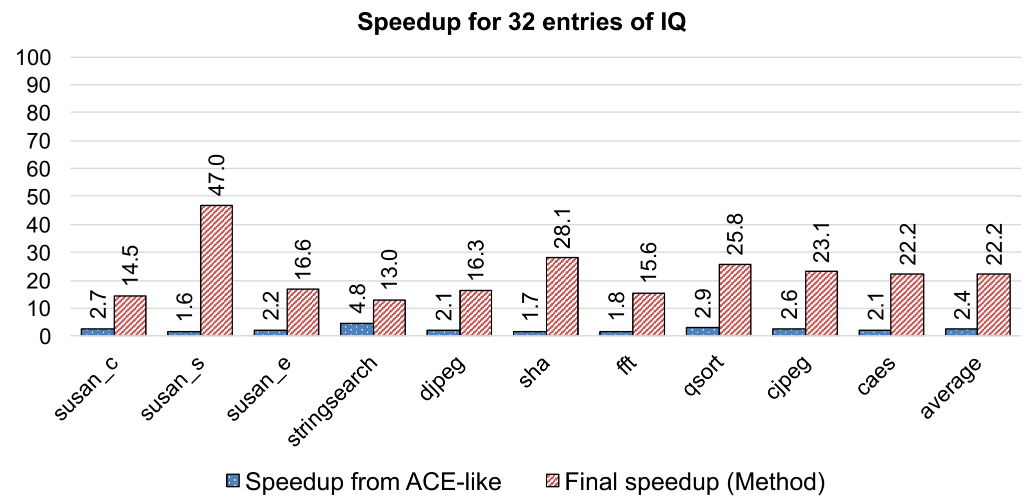


Figure 61: MeRLiN speedup for the Issue Queue with 32 entries running 10 MiBench benchmarks.

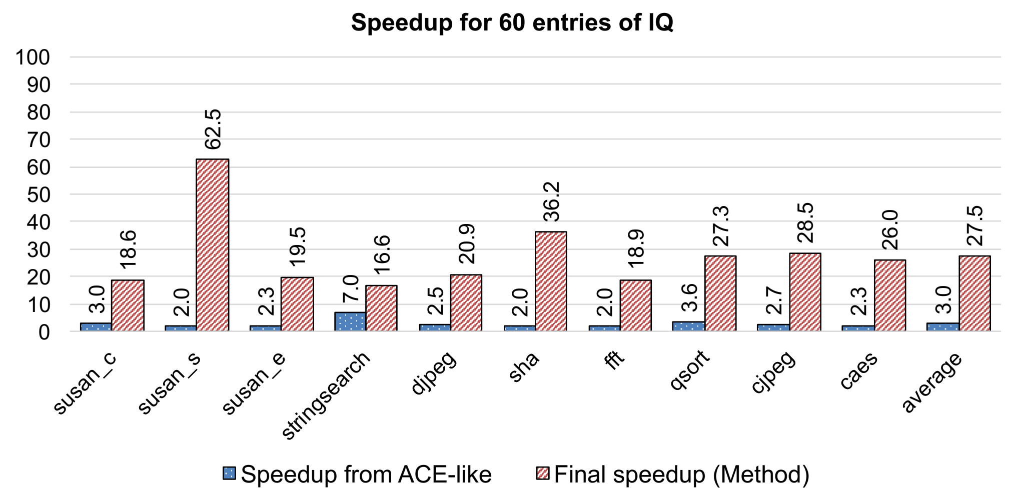


Figure 62: MeRLiN speedup for the Issue Queue with 60 entries running 10 MiBench benchmarks.

Concerning the accuracy, similar to the data-related structures we firstly evaluate the *homogeneity* of the group creation algorithm that was described in equation (10) for the six fine-grained classes of fault effects of Table 4. Figure 63 presents the *homogeneity* of the created groups when MeRLiN targets the Issue Queue with 32 and 60 entries respectively. We can observe that on average the homogeneity is very high for the two configurations of the IQ (0.92 and 0.93 for 32 and 60 entries respectively) even when we use six fault effect categories. This indicates the high accuracy of MeRLiN's group creation algorithm when MeRLiN is applied on the Issue Queue. The average percentage of groups with perfect homogeneity (equal to 1.0) for the Issue Queue when we use six classes of fault effects is 80.5% and 82.5% for 32 and 60 entries respectively; a very good indication of the accuracy of the grouping algorithm.

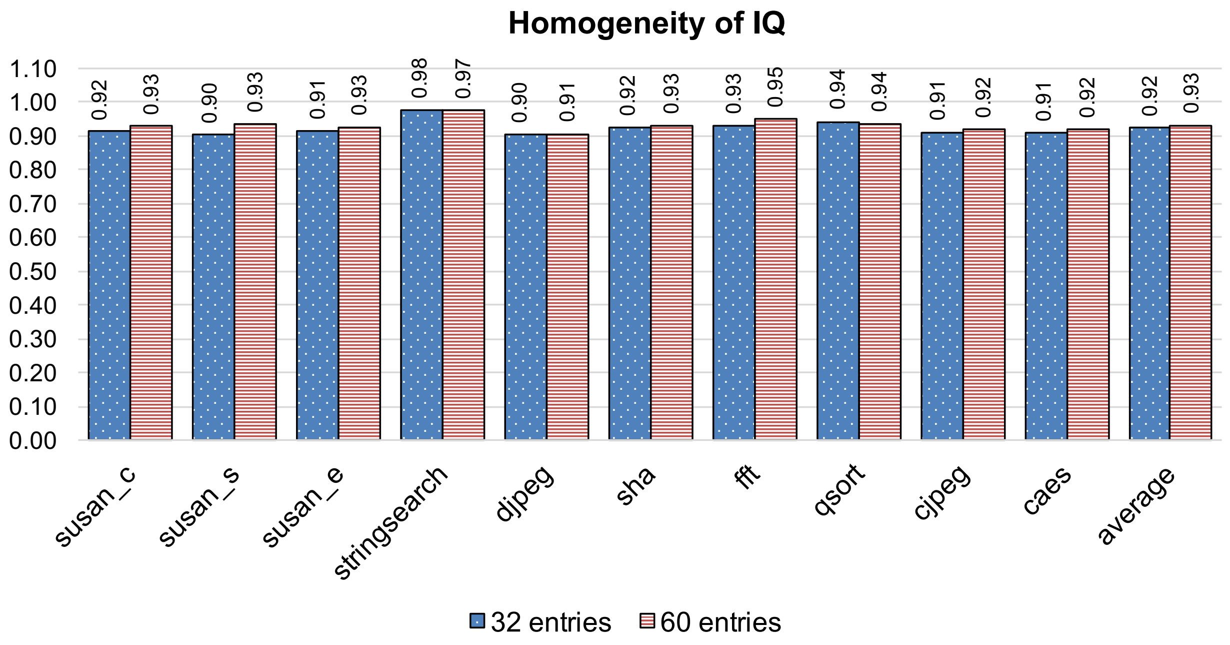


Figure 63: Homogeneity of Issue Queue.

The estimation accuracy of MeRLiN for the Issue Queue against the injection using the remaining fault list *after* the *ACE-like* step is shown in Figure 64. The graph shows the average fault effect classification across the 10 MiBench benchmarks used in our study for the two configurations of the Issue Queue. The first bar (blue) in each class corresponds to the results of the fault injection in the remaining fault list *after* the *ACE-like* analysis, while the second bar (red) illustrates the results on the same fault list after applying MeRLiN's grouping algorithm and injecting only the selected faults. The values on top of each bar represent the measurement per fault effect category. Similar behavior is observed across all benchmarks. For the Issue Queue and the two configurations, MeRLiN reports negligible differences compared to the injection using all the faults that hit only vulnerable intervals.

Figure 65 shows the bigger picture of MeRLiN's accuracy for the Issue Queue, in which the final fault effect classification of the comprehensive baseline fault injection of 60,000 faults (blue bar) is compared to the final classification of MeRLiN (red bar). Each bar represents the average values across the 10 MiBench benchmarks. Similar behavior is observed across all benchmarks. MeRLiN for all cases is extremely accurate and delivers virtually the same reports with the comprehensive injection.

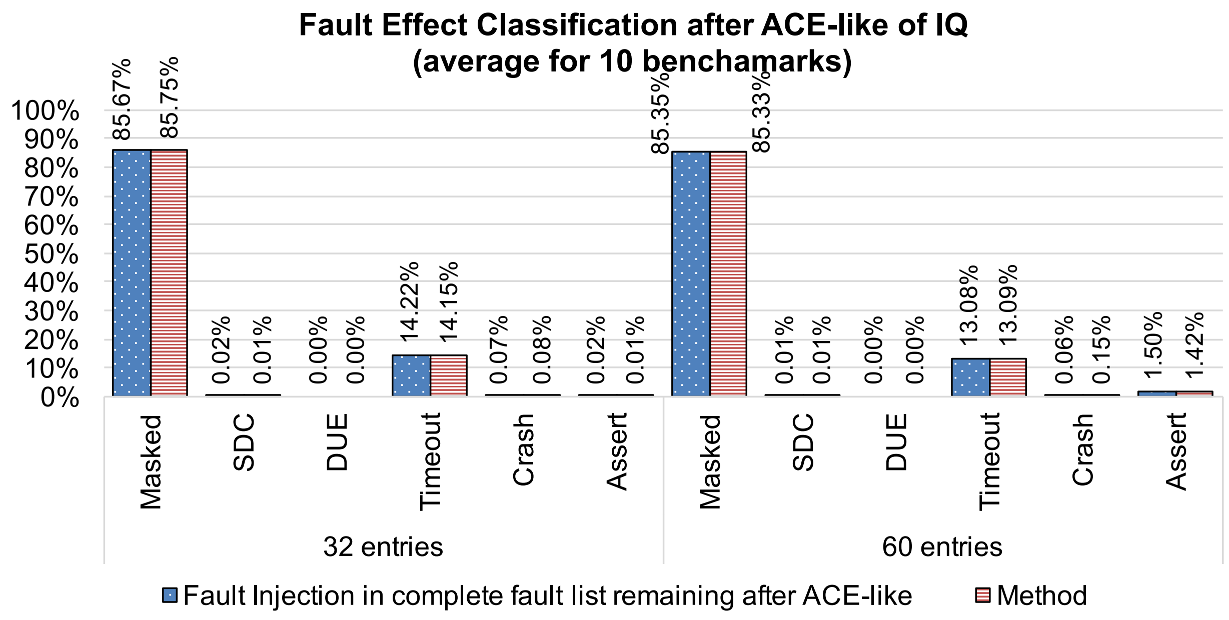


Figure 64: Fault effect classification of MeRLiN against injection with the remaining faults after *ACE-like* step for the Issue Queue; average for 10 MiBench benchmarks.

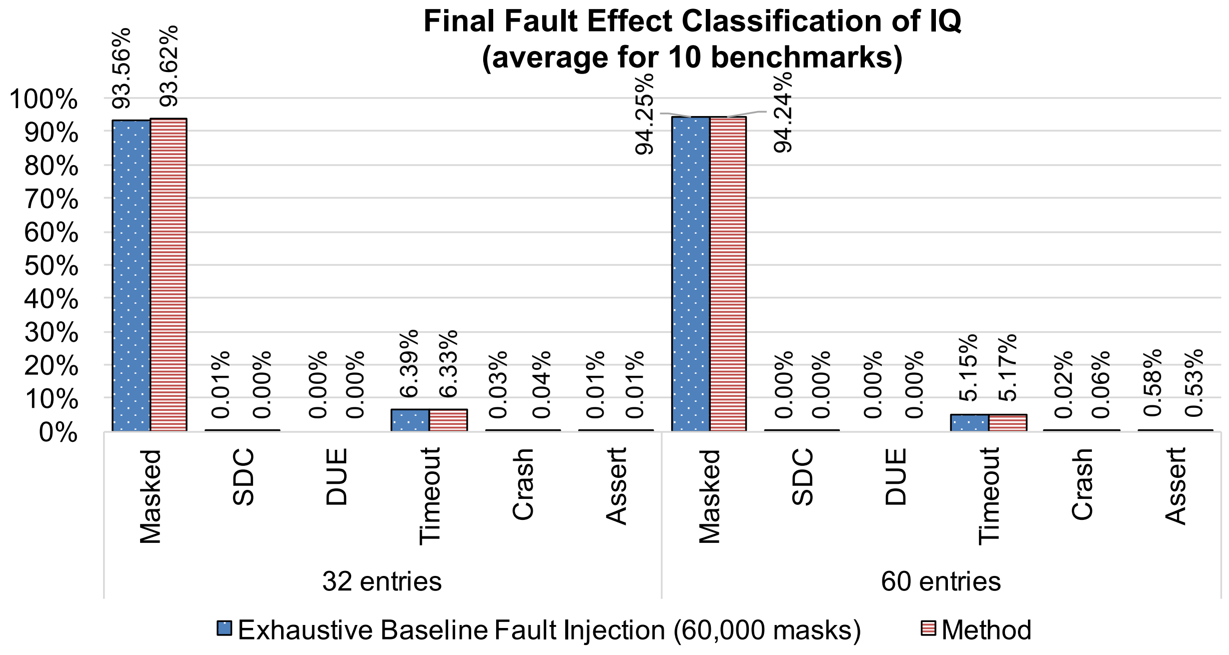


Figure 65: Final fault effect classification of MeRLiN against comprehensive baseline fault injection with 60,000 faults for the Issue Queue; average for 10 MiBench benchmarks.

Finally, Figure 66 demonstrates the final reliability estimation in Failures-in-Time (FIT) rates for the comprehensive baseline campaign (60,000 faults), the MeRLiN method and the *ACE-like* method running the 10 MiBench benchmarks to the end targeting the Issue Queue. The reported FIT rates are the products of AVF, raw FIT rate and number of structure's bits. The AVF of the injection-based methods is the ratio of the non-masked injections over the total injections, while the AVF of the *ACE-like* is measured as in [50]. We used 0.01 FIT per bit, but any raw FIT rate can be used. MeRLiN reports negligible differences compared to the comprehensive baseline injection for the IQ, while the *ACE-like* delivers a pessimistic lower bound of structures' reliability.

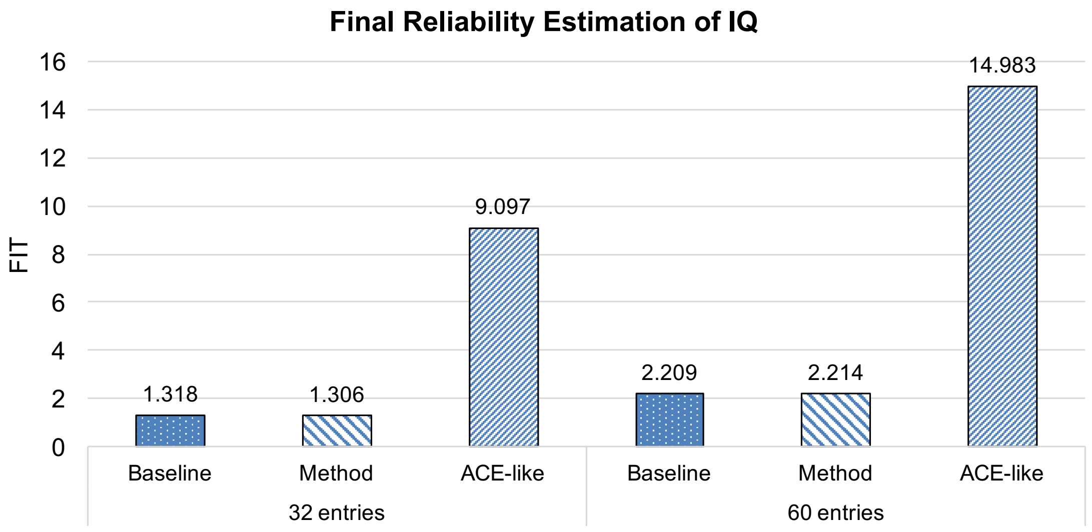


Figure 66: Final reliability assessment (FIT) for the Issue Queue (average for 10 MiBench).

3.3 Comparison of MeRLiN with architecture level fault injection approaches

In this section, we present a quantitative and qualitative comparison between MeRLiN that is an approach that targets the reliability evaluation at the microarchitecture level and Relyzer [87] a state-of-the-art approach to estimate reliability at the software level.

Firstly, the two approaches have fundamental differences concerning the nature of the initial fault list population. An *exhaustive* fault list at the microarchitecture level consists of all flips for every bit of a hardware structure and for every program execution cycle. At the software, the same list consists of bit flips in the operands of the assembly instructions; these faults are not correlated to the execution time of the program and the actual bits of the hardware.

Table 22 presents a high-level quantitative comparison of Relyzer [87] and MeRLiN using as starting point the exhaustive fault list of the corresponding level of abstraction (first column). The second column shows the faults of the exhaustive list that remain for injection after the application of each method, and the third column presents the *gains* (speedup) in terms of fault list reduction achieved by each method over the corresponding exhaustive list. The last two columns show the time needed to inject the *exhaustive* list and the remaining faults in both methods, respectively. Assume that we run one benchmark of 1 billion cycles and we inject faults in the L1D (32KB), the SQ (16 entries) and the RF (64 registers). The throughput of Gem5 for full-system cycle-accurate simulation is 10^5 cycles/sec while for software emulation it is 10^6 cycles/sec [15]. MeRLiN delivers 5 orders of magnitude higher gains than Relyzer having as starting point the exhaustive list, while it reports the reliability of the exhaustive list 10 orders of magnitude faster. Thus, statistical fault sampling is unavoidable due to the huge number of faults in the exhaustive fault list for both hardware and software level methods.

Table 22: MeRLiN vs. Relyzer using exhaustive fault list.

	Exhaustive fault list	Remaining faults	Gain	Evaluation time using exhaustive fault list	Evaluation time using remaining faults
MeRLiN	10^{13}	10^3	10^{10}	~3×10^9 years	4 months
Relyzer	10^{11}	10^6	10^5	~3×10^6 years	32 years

Both MeRLiN and Relyzer prune faults of the initial fault list being injected at different levels of the system stack. Thus, in the next bullets we analyze the applicability of Relyzer heuristics at the microarchitecture level injection:

- **Bounding addresses**: It prunes faults in the address field of store and load instructions if the valid address space is violated. This heuristic requires an unaffordable amount of memory to track the addresses in data related structures (e.g. caches). Also, MeRLiN provides finer grained effect classification for non-masking categories (Table 4) and is not limited to symptom-based techniques.
- **Def-use**: It prunes faults in the destination architectural register of an instruction followed by another instruction that consumes this value, as these faults will have the same effect.
- **Store-equivalence** is similar to the def-use for store and load instructions. These two heuristics cannot be applied at the microarchitecture level of our work. The destination register of an instruction and the source register of a subsequent correspond to the same physical entity [88].
- **Control-equivalence:** Software analysis using basic blocks tracks the control flow paths of all the dynamic instances of all the static instructions to separate Masked from SDC faults [89]. For each path Relyzer randomly chooses only one pilot. To evaluate this heuristic, we ran the 10 MiBench to the end with 128 registers, 16 SQ entries and 32KB L1D. Exhaustive fault injection is infeasible; thus, we used the remaining faults (from 60,000 initial faults) after the pruning by our ACE-like step. We used a control flow path depth of 5, exactly as Relyzer does [87].

  In terms of speedup, MeRLiN slightly prevails on average in the RF (62.1X compared to 60.5X) and the L1D (60.1X compared to 59.1X), while for the SQ, MeRLiN provides 146.9X speedup compared to 150.6X of Relyzer's heuristic. Figure 67 illustrates the results of the comparison in terms of inaccuracy in percentile units compared to the injection using the same fault list.

  A source of Relyzer’s inaccuracy is the static instructions with large population of faults that are represented by only one randomly selected pilot. In [87], 52% on average of all static instructions have only 1 pilot. We measured that Relyzer leaves 9% of the groups correlated to a static instruction with large population of faults (more than 100 faults) with only 1 pilot, while MeRLiN leaves less than 2%. The heuristic of Relyzer if applied to our statistical concept selects only one pilot for code loops with large number of iterations. Assume a for-loop with 1000 iterations that consists of only one static instruction with only two control flow paths with 995 and 5 instances, respectively. Due to statistical sampling, all faults may come only from the first path. In this case, Relyzer chooses only one pilot for this loop. On the contrary, MeRLiN, due to the homogenous distribution of faults, chooses more than one from different bytes and dynamic instances. These large loops exist in most program execution phases, including initialization and output phase that are not examined by [87]. Despite of Relyzer’s indisputable merit in software resilience, this heuristic of Relyzer is not so efficient to be employed in our concept.

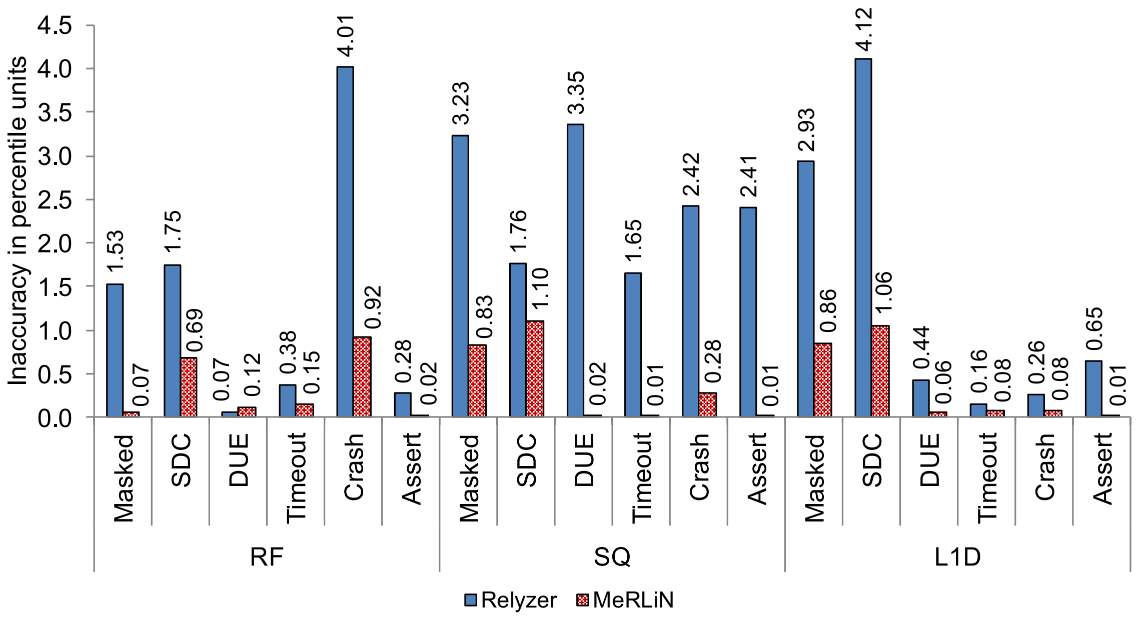


Figure 67: Inaccuracy of MeRLiN and Relyzer vs. injection with the remaining faults after *ACE-like*; average for 10 MiBench.

3.4 Theoretical analysis of MeRLiN

In this section, we analyze the statistical behavior of MeRLiN comparing the mean and the variance of the AVF measurements it reports to the corresponding mean and variance of the comprehensive fault injection campaign. We assume that soft errors affecting the microprocessor bits follow a normal distribution [62]. A fault injection campaign can be described as a binomial experiment of F individual injections, each of which has a probability of *success* (program is affected) or *failure* (program is not affected; fault is masked). Thus, the AVF measurement k ($0 \le k \le 1$) in our case means that $k \cdot F$ faults are Not-Masked.

MeRLiN's first phase prunes a fraction m ($0 \le m \le 1$) of the F faults that are guaranteed masked: $m \cdot F$. The remaining $(1-m) \cdot F$ faults (which now contain all $k \cdot F$ Not-Masked faults of the initial list of F faults) are forwarded to the second phase of MeRLiN (grouping). This second phase produces n groups of faults with sizes s_i (i=1, 2, … , n). The sum of the group sizes is equal to the number of faults passed to the second phase:

$$s_1 + s_2 + \ \dots\ + s_n = (1-m) \cdot F \tag{11}$$

When the comprehensive injection campaign (without MeRLiN) is applied, all F faults are injected and the outcome r of each run is observed (Not-Masked=1 or Masked=0). In this case, the AVF (k) is[3]:

$$k = \frac{\sum_{i=1}^{n} \sum_{j=1}^{s_i} r_i^j}{F} \tag{12}$$

We assume that the probability of Non-Masking within a group i is p_i. Within a group i, all faults have the same probability p_i because of MeRLiN's grouping criterion: faults in a

[3] We could consider as group 0 with size $s_0 = m \cdot F$ the group of faults from MeRLiN's pre-processing step but since all faults of this group are masked, i.e. r=0, this group is not needed in the calculations.

group hit the *same* byte of the entries during a vulnerable interval that ends with the *same* instruction that reads the entry. The results of Figure 47 show the validity of this assumption; they indicate that the vast majority of groups have homogeneity close to 1.0 (considering only the masked and non-masked categories) and that the percentage of groups with perfect homogeneity is very large in all cases. Across groups, probabilities p_i are different since the groups correspond to faults eventually read by different instructions. The mean (expected value; *E*) of the AVF measurement *k* in the comprehensive campaign is[4]:

$$E(k) = E\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} r_i^j}{F}\right) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} E(r_i^j)}{F} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} p_i}{F} = \frac{\sum_{i=1}^{n} s_i \cdot p_i}{F} \tag{13}$$

When MeRLiN is employed it delivers a new AVF measurement k_{MeRLiN}. For each run *r* of the selected fault from a group *i* all faults are assumed to have the same result (1=Not-Masked, 0=Masked). So, the true measurement in this case is $s_i\, r_i$ for each group *i* and the new AVF k_{MeRLiN} is:

$$k_{MeRLiN} = \frac{\sum_{i=1}^{n} s_i \cdot r_i}{F} \tag{14}$$

which has a mean

$$E(k_{MeRLiN}) = E\left(\frac{\sum_{i=1}^{n} s_i \cdot r_i}{F}\right) = \frac{\sum_{i=1}^{n} E(s_i \cdot r_i)}{F} = \frac{\sum_{i=1}^{n} s_i E(r_i)}{F} = \frac{\sum_{i=1}^{n} s_i \cdot p_i}{F} = E(k) \tag{15}$$

therefore, MeRLiN reports AVF with the same mean value as the original comprehensive set of *F* fault injections. The variance of the AVF measurements *k* and k_{MeRLiN} is shown in the following equations[5]:

[4] We use the linearity property of the means of independent variables which holds for binomial distribution. The mean of a binomially distributed variable is $E(X) = n \cdot p$ with *n* experiments and *p* success probability.
[5] We use the relation $\sigma^2(a \cdot X + b \cdot Y) = a^2 \cdot \sigma^2(X) + b^2 \cdot \sigma^2(Y)$ for the variances of independent variables. Group 0 has zero variance.

$$\sigma^2(k)=\sigma^2\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} r_i^j}{F}\right)=\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i}\sigma^2(r_i^j)}{F^2}=\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} p_i\cdot(1-p_i)}{F^2}=\frac{\sum_{i=1}^{n} s_i\cdot p_i\cdot(1-p_i)}{F^2} \qquad (16)$$

$$\sigma^2(k_{MeRLiN})=\sigma^2\left(\frac{\sum_{i=1}^{n} s_i\cdot r_i}{F}\right)=\frac{\sum_{i=1}^{n} s_i^2\cdot\sigma^2(r_i^j)}{F^2}=\frac{\sum_{i=1}^{n} s_i^2\cdot p_i\cdot(1-p_i)}{F^2} \qquad (17)$$

The values of both $\sigma^2(k)$ and $\sigma^2(k_{MeRLiN})$ are very small (several orders of magnitude smaller than the means of k and k_{MeRLiN}, respectively) for two reasons:

- the groups generated by MeRLiN are very homogeneous; thus, either p_i or $(1–p_i)$ is zero or is very small as shown in Section 3.1 and Section 3.2
- the sizes of the groups (s_i values) are very small compared to F

In our experiments, the average size of a MeRLiN group is always less than 100 and typically ranges between 5 and 40. Thus, with simple calculations on the above equations the variance of the initial AVF value when F consists of 60K faults is about 8 to 10 orders of magnitude smaller than the mean. Therefore, the multiplication with the s_i values in the variance of MeRLiN's AVF measurements $\sigma^2(k_{MeRLiN})$ keeps this variance from 6 to 8 orders of magnitude smaller than the mean (assuming s_i values up to 100): still *a very small variance*, only slightly increased compared to the initial one.

Overall our analysis shows that the AVF measurement of MeRLiN has the same mean as the comprehensive experiment of F injections, while both have a very small variance. These two statistical properties make them almost statistically equivalent although MeRLiN reports AVF in 1 to 3 orders of magnitude shorter time.

3.5 Related work

There are several studies that focus on reliability assessments. Next, we present and categorize these studies according to their correlation with the contributions of this thesis:

- **Approaches for reliability estimation of hardware components**: In Section 2.1.4, we described in detail the four more popular techniques that are used to estimate the reliability of hardware structures: (i) RTL injection [45] [46] [47] [95], (ii) microarchitecture level injection [29] [30] [31] [48] [49], (iii) ACE (Architecturally Correct Execution) analysis [50] [51] [52] [53] and (iv) probabilistic models [54] [55] [56] [57].
- **Fault injection tools for reliability assessments**: A microarchitecture-level injection tool built on M5 simulator [96] for Alpha ISA only is briefly described in [48]. The injector was built on a simple in-order microarchitecture and reliability studies of complex out-of-order x86 or ARM microprocessors are not supported. An injection tool based on Gem5 and the Alpha ISA is described in [97]; the tool only injects transient faults in architectural registers. Also, [57] [58] use PTLsim for fault injections on very few hardware structures.

Other approaches combine performance simulators with lower-level simulators to improve reliability assessments accuracy. The approach in [78] presents a combination of GEMS and Simics simulators with Cadence NC-Verilog gate-level simulator. For logic components, it delivers a more accurate estimation at the expense of long simulation times. In [46], a fault injection method at the RTL and gate-level is described for the control blocks of an Alpha microprocessor.

Microarchitectural simulators have been also used for injections only at architectural visible points (the architectural registers) to measure the effectiveness of error protection techniques. In [98], the ASIM functional simulator is used and faults are only injected at the registers.

Other tools target even lower levels and work at the RTL, on FPGA realizations of a microarchitecture or on hardware emulators. The experimental study of [99] injects faults in a DLX processor FPGA realization and an ASIC realization of an Alpha processor. The framework described in [100] uses an FPGA-based system for the reliability characterization of a full system stack. In [101], an FPGA-based reliability analysis framework is described. In [47] and [102] an RTL model of an Alpha processor is developed and used for fault injection experiments. In [103], faults are injected in an RTL model of picoJava-II processor. In [104], a hardware emulation platform is used for injections at the latches of a microarchitecture. An interesting study [45] quantitatively evaluates the impact of flip-flop soft errors using several injection approaches at different levels of abstraction and discussed the sources of inaccuracies when higher levels of abstraction are employed in fault injection setups. This study does not evaluate the accuracy of the reliability estimations at the microarchitecture level that is the goal of this thesis.

- **Lifetime analysis studies**: Lifetime analysis [40] is a common analysis that is used in many reliability-reliability studies. This analysis is based on the separation of a hardware entry lifetime into vulnerable (in which a fault can potentially affect the program execution) and non-vulnerable intervals (in which a fault cannot affect the program execution) according to the access patterns on the same entry. Lifetime analysis has been previously used in several reliability-related studies. The method of [86] uses execution intervals sampling for reliability evaluation of caches. In [42] and [43] the authors separate the Hardware Vulnerability Factor (HVF) from the Program Vulnerability Factor (PVF), while [105] focuses on on-line vulnerability estimation and [60] aims to develop stressmarks to measure the maximum vulnerability of hardware structures to soft errors. The methods in [106], [107], [108] and [109] use lifetime analysis to support decision-making for error protection. However, none of these studies uses the lifetime analysis to accelerate the fault injection campaigns at the microarchitecture level (as MeRLiN does).
- **Cross-layer approaches to evaluate system reliability**: Performing cross-layer system reliability analysis, requires a deep understanding of the layers where faults appear in the system, how faults generate errors, and how errors propagate across layers, eventually impacting the final mission of the system. Performing system reliability analysis means calculating the different vulnerability factors associated with the components of a system, and then understanding how all masking effects work together and how they influence the behavior of the system. The key concept is to analyze the three system layers (technology, hardware and software layers) separately computing different vulnerability factors for the different blocks. Vulnerability factors are then statistically combined in

order to infer reliability measures at the system level. Analyzing the layers in isolation has the main advantage to reduce the complexity of the analysis focusing on the peculiar masking effects each layer can provide. Each layer defines an interface with the upper layer, which in turn sets how faults can be propagated from one layer to the next one. In [90], [91] and [92], MaFIN was used to propose a scheme for cross-layer system reliability analysis, while GeFIN [29] [30] was used in [93].

3.6 Findings Summary

In Chapter 2 and Chapter 3, we presented several techniques to analyze the reliability of modern microprocessors against different types faults in the early pre-silicon design phase. Here, we summarize the most important highlights that were presented:

- We developed a fault injection framework (called MaFIN [27] [29]) to support injection of transient, intermittent, permanent faults and multiple faults in x86-64 modern microprocessors. Moreover, we enhanced the microarchitectural simulator with the data arrays in all cache levels to ensure accurate reliability estimations on these structures. MaFIN tool supports fault injection in the most important array-based structures of the microprocessor (see Table 5) that occupy the majority of chip's area. Finally, MaFIN can parse the results of the fault injection campaigns in six different fault effect categories (Masked, SDC, DUE, Timeout, Crash, Assert) according to Table 4. MaFIN tool can be used to support several reliability estimation studies, such as: (i) reliability-performance tradeoffs assessment studies [28], (ii) differential studies on reliability assessments [29], and (iii) studies that focus to accelerate the reliability assessments [31].
- In [28], we presented a complete fault injection analysis of transient faults which jointly considers the reliability and the performance impact of several important design parameters on a modern out-of-order x86-64 architecture. To achieve this, we also proposed a simple and flexible *fitness function* that measures the aggregate effect of such design changes on the reliability and the performance of the studied workload.
- In [29], we investigated the limits of microarchitecture level fault injection for x86 and ARM ISAs conducting a *differential analysis* on two comprehensive fault injector tools (MaFIN [27] [29] and GeFIN [30]) supporting the same fault models and running the same workloads. This differential analysis brought insights concerning the sensitivity of the vulnerability of hardware structures and workloads to the underlying microarchitecture as well as the ISA of the microprocessor. It also identified common trends and diverging reliability reports in the two tools which can lead to informed design decisions for error protection. We explained the common trends and the sources of difference when diverging reliability reports are provided by the tools using benchmarks runtime statistics.
- In [31], we extended the baseline mode of MaFIN fault injection framework with two extra modes of operation in order to speed up the statistical fault injection campaigns at the microarchitecture level. The common characteristic of the two proposed techniques of [31] is that they are implemented after the actual injection of the fault in the hardware structure during its lifetime. In the first mode, an injection experiment is forced to completion when the fault is overwritten before it is read and thus we classify it early and accurately as masked. In the second mode, an injection experiment is forced to completion when the fault is overwritten before it is read or when an x86 instruction reads the fault from the faulty entry and reaches the commit stage. The second method provides a

tradeoff between speedup and accuracy in order to deliver a fast but not so accurate solution in the early reliability estimation problem. From our experimental results, we observed for the first mode a speedup up to 2.92X with no loss of accuracy in the vulnerability measurements for all structures, while in the second mode an even higher speedup of up to 4.06X has been obtained with small loss in the accuracy of the vulnerability measurements.

- In [32], we presented MeRLiN methodology a state-of-the-art method to accelerate statistical fault injection campaigns at the microarchitecture level by pruning the faults of the initial fault list before their actual injection. This method is different than the techniques proposed in [31] that target to accelerate the fault injection campaigns during faults lifetime. MeRLiN accelerates statistical microarchitecture level fault injection from 1 to 3 orders of magnitude. Our experiments with full runs of 10 MiBench benchmarks [74] show 93X, 225X, 68X and 28X speedup on average for different sizes of the register file, the store queue, the first level data cache and the issue queue, respectively. When applied to 10 SPEC CPU2006 benchmarks, MeRLiN reveals larger average speedups of 1644X, 2018X and 171X for the register file, the store queue and the first level data cache, respectively. Moreover, MeRLiN reports virtually the same reliability estimations as conventional microarchitectural fault injection with extremely high statistical significance. Finally, MeRLiN delivers fine-grained insights of the fault effects (Silent Data Corruptions-SDC, Detected Unrecoverable Errors-DUE, crashes, locks) unlike ACE analysis which only reports a gross AVF estimate.

4. POST-SILICON RELIABILITY ANALYSIS

This chapter aims to describe two techniques that are implemented during *Post-Silicon Reliability Analysis* phase (see Figure 8), when the massive production of chips begins or the chips are already released to the market. Section 4.1 presents a technique for accelerated online detection of permanent faults in many-core architectures using the 48-core Intel's SCC architecture as experimental vehicle, while Section 4.2 presents a statistical analysis methodology that was proposed to boost the energy efficiency of the eight ARMv8-based cores of the X-Gene 2 chip by predicting the safe voltage operation margins of each individual core.

4.1 Online permanent fault detection in many-core architectures

Massive parallelism in many-core architectures holds an important place in modern computing. The microprocessors industry is moving rapidly to the many-core era and such architectures are expected to dominate in the near future in various computing domains, including general purpose microprocessors [110], graphics processing units (GPUs) [111] or other hardware accelerators, as well as specialized Systems-on-chip (SoCs) [112] such as network processing units (NPUs) [113], [114] and [115]. The extreme complexity of many-core processor architectures and the pressure for reduced time-to-market (TTM) renders even the most comprehensive verification and testing campaign before and during mass production incomplete. A significant population of manufacturing faults escape in-field and may jeopardize correct operation of the chip. Furthermore, the environmental impact and wear-out effects that lead to intermittent or permanent faults increase the failure probability of modern designs. Thus, developing techniques capable of detecting errors online is a necessity [11].

Several online error detection schemes have been proposed in the literature to enhance microprocessor's reliability. Online schemes are divided into concurrent testing, which takes place during normal system operation and non-concurrent testing, which is performed during idle periods or while system normal operation is interrupted. Concurrent testing schemes [116] typically built with hardware assertion checkers impose many constraints (e.g. area) on the microprocessor design process. On the other hand, there are non-concurrent online error detection approaches, such as the software-based online testing, that do not intrude the processor design but may impact system throughput as they need themselves some execution time for error detection.

Functional or software-based testing techniques have gained increasing acceptance for microprocessor error detection during the last years and currently forms an integral part of the processor manufacturing test flow [117]. Functional online error detection approaches for many-core architectures are based on the application of the test programs during normal system operation and should adhere to the following requirements:

- *Reduced test program execution time*: reduction of the test execution time allows a better trade-off between *reliability* (executing more tests or the same tests more frequently), *performance overhead* (interrupting user workload for a shorter period) and *availability* (the smaller the test execution time, the shorter the service downtime period).
- *Small memory footprint*: A single copy of the test program (test code and data) is stored in the shared memory (system's main memory) instead of using separate copies in each core; this reduces storage requirements and avoids the scaling of test program memory footprint with the number of cores.

- *Test program replication*: All processor cores have to execute the same test program to detect faults and guarantee high fault coverage levels.

There are two approaches that are described with the example of Figure 68 for the online execution of a test program in many-core architectures:

- *Non-intrusive approach*: Each core runs the test programs individually during its idle periods.
- *Intrusive approach*: Groups of cores or all cores together are periodically isolated and set out-of-service (do not run normal workload) to run simultaneously the test programs.

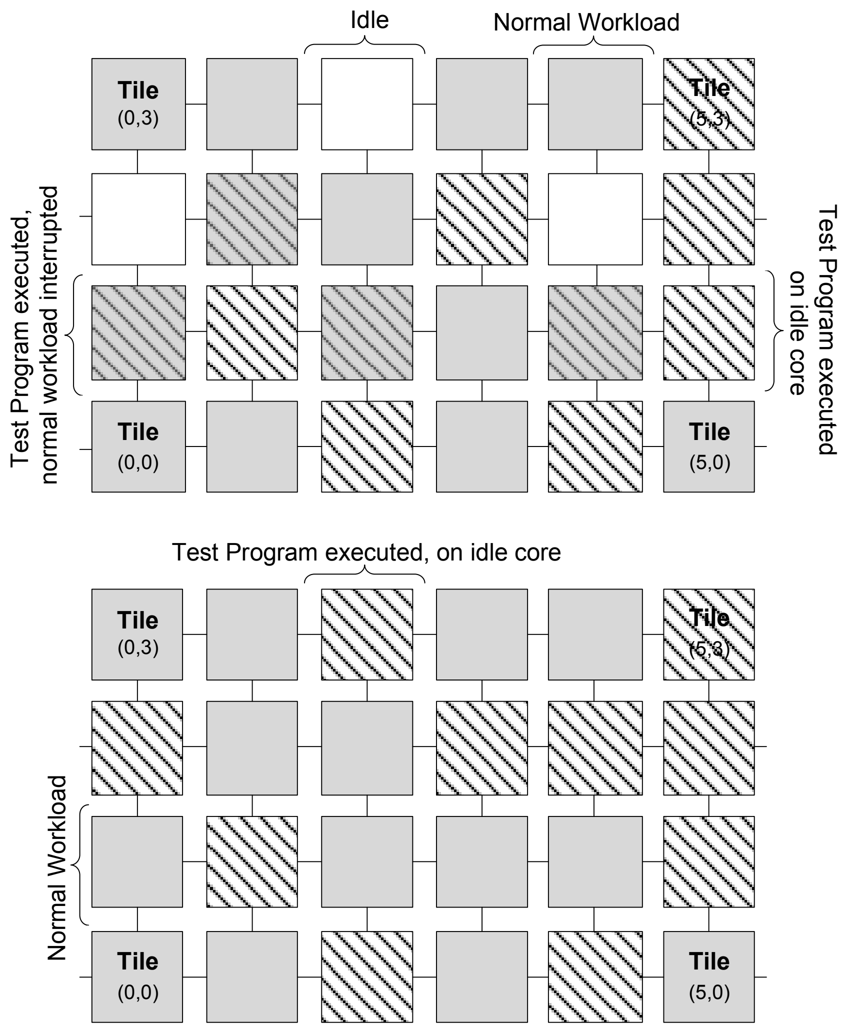

Idle
Normal Workload
Tile (0,3)
Tile (5,3)
Test Program executed, normal workload interrupted
Test Program executed on idle core
Tile (0,0)
Tile (5,0)
Test Program executed, on idle core
Tile (0,3)
Tile (5,3)
Normal Workload
Tile (0,0)
Tile (5,0)

Figure 68: Intrusive (upper) and non-intrusive (lower) online functional detection approach.

An online error detection methodology has to address a major challenge: reduce the duration of the test program execution exploiting the massively execution parallelism and also limiting the contention of cores for the shared resources. Otherwise, the overall test execution time will excessively scale with the number of cores. This phenomenon is exacerbated in many-core designs, where multiple processing elements compete to take control of shared resources. For the case of the non-intrusive approach, the time reduction makes feasible the execution of test programs in shorter idle periods increasing the likelihood of an idle time slot to execute the test program even in high-utilized processor cores. On the contrary, shorter execution time for the intrusive approach means either less throughput overhead or more frequent runs, thus shorter error detection latency.

Note also that online error detection in parallel architectures radically differs from conventional parallel programing problems. In classic programming a serial program is broken in parts that are executed in the different processors, while in online error detection the entire test program must be applied individually in every core.

In [33], we propose an intrusive online error detection methodology to detect permanent faults, which guarantees the periodicity of the test program execution bounding the maximum detection latency and has a minimal impact on system's performance. In Section 4.1.1, we describe all the details of the proposed method to accelerate the detection of permanent faults in many-core architectures. In Section 4.1.2, we present the results of the evaluation of the proposed technique and finally Section 4.1.3 summarizes the related work in the field.

4.1.1 Proposed method to accelerate permanent fault online detection in many-core architectures

In [33], we used as experimental vehicle the Intel's Single-chip Cloud Computer (SCC), a many-core chip created by Intel Labs that contains 48 in-order Pentium cores [34] and is assumed to be one of the predecessors of Intel's today many-core accelerator, Xeon Phi^{TM} [118]. The SCC architecture shown in Figure 69 consists of 24 tiles (two cores per tile) and 4 integrated DDR3 memory controllers supporting up to 64GB DRAM. Each processor SCC core has a 16KB L1 instruction cache, a 16KB L1 data cache and a 256KB L2 cache. The off-chip DRAM can be divided into private and shared memory. Private memory regions are cacheable while shared memory regions can be configured either as cacheable or non-cacheable. Moreover, each tile has a 16KB message passing buffer (MPB). When a core sends a message to another core, the data move from the L1 cache of the sending core to its MPB and then to the L1 cache of the receiving core, while L2 cache is bypassed. Messages employ XY routing (first along the horizontal direction and then along the vertical direction). A C-based library (called RCCE) is used for message passing programs and parallel programming.

There are many challenges in setting up and evaluating an online test method for a real many-core chip such as SCC, as opposed to using an architecture simulator as experimental vehicle. One important issue when building an intrusive online error detection approach in a many-core architecture is that all cores must start to execute the test program simultaneously. This means that all cores must first pause or complete execution of their normal workload and synchronize before they start executing the test program. They must be also synchronized again before they restart their normal workload. In the SCC architecture, core synchronization is achieved using the barrier function; we use this mechanism in our experiments.

For the experiments on the SCC chip, we developed two test programs with different characteristics which represent typical test program formats used in functional online testing:

- *Load-Apply-Accumulate* (LAA) test program. It applies ATPG-generated test patterns (or pre-computed by other means) stored in the off-chip DRAM. An LAA test program first reads two test vectors from the DRAM (assuming two-operand operations are being tested); it applies the target instruction (i.e. an arithmetic or logic instruction) and finally accumulates the results. We experiment with a loop-based LAA test program which applies a certain amount of test patterns (192KB or 384KB). LAA test program is memory-intensive and stresses the memory system of the SCC processor.
- *Linear-Feedback-Shift-Register* (LFSR) test program. This CPU-intensive test program applies pseudorandom patterns generated by a 32-bit LFSR. Similarly

to LAA program, it first generates two pseudorandom test patterns, applies the target instruction and accumulates the results. LFSR test program generates either the same number of test data with LAA or 10 times more (e.g. for 384KB LAA, LFSR generates 3840KB test data).

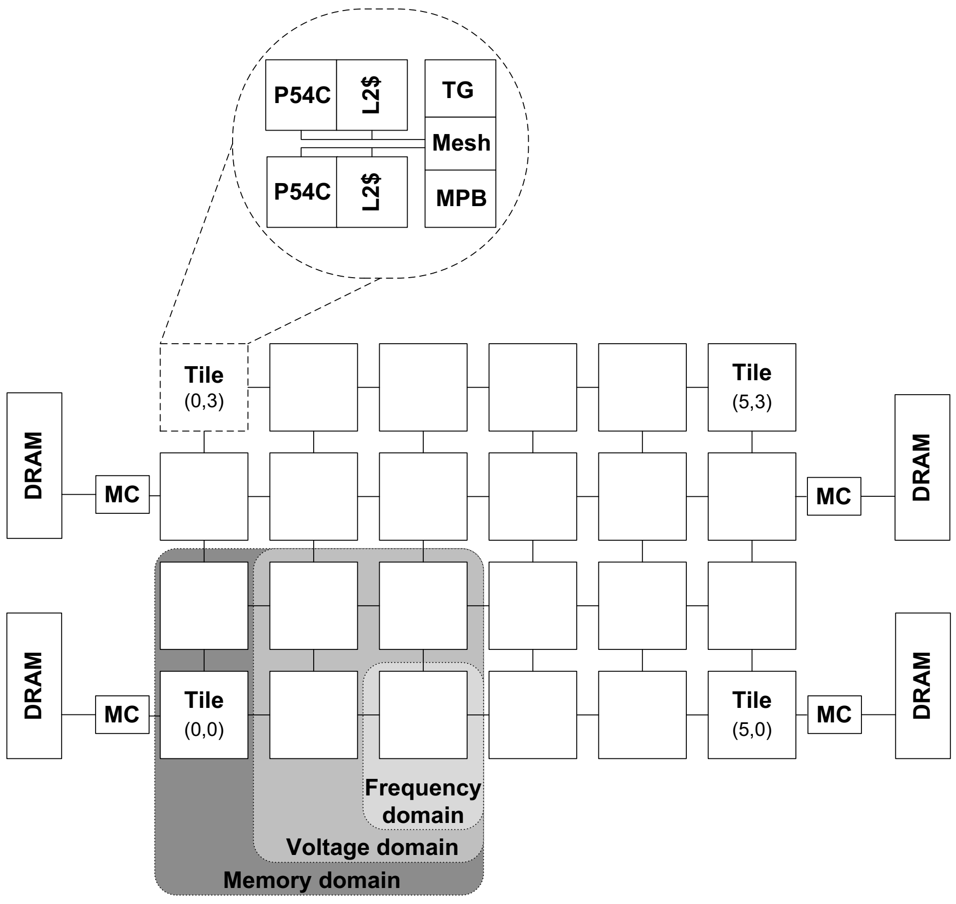


Figure 69: Intel's many-core SCC Architecture.

Firstly, we explain why an online test program is not a conventional program from the parallel computing perspective. Next, we describe a naïve parallelization method of the two typical test programs and explain why it achieves very low speedup against an obvious (but senseless) "serial" method. After that, we explain why a more advanced parallelization method should consider the memory system parameters of a many-core processor architecture that primarily affect the performance of parallel test programs. Finally, we propose an effective parallelization method that exploits the high-speed on-chip message passing buffers infrastructure to accelerate the generation of test patterns. It should be noted that the proposed parallelization method is coupled to the particular network topology and memory organization, but with some modifications and extensions can be applicable in similar many-core architectures.

Among the three basic phases of functional test programs, i.e. test *preparation*, test *application* and test *response compaction*, the last two cannot be parallelized because each test pattern must be applied to every core and the core's response must be compacted separately. Only the test preparation phase (production of all patterns that must be eventually applied to each core) can be performed *only once* for the entire chip. The test preparation task can be divided and parallelized in a many-core processor architecture balancing the test preparation workload between the cores to achieve the maximum speedup.

Naïve Parallelization Method:

In a potential naïve parallelization method, all processor cores execute in parallel the same test program starting execution simultaneously. According to the naïve parallel intrusive approach shown in Figure 70, after the normal workload has been paused in all processor cores, the cores are synchronized to execute the test program in parallel. Although all cores run an identical test program they do not finish at the same time due to the non-uniformity of the memory accesses of the SCC architecture. The worst case execution time determines the online test cycle duration. Upon completion of the online test phase, normal workload is resumed in all cores.

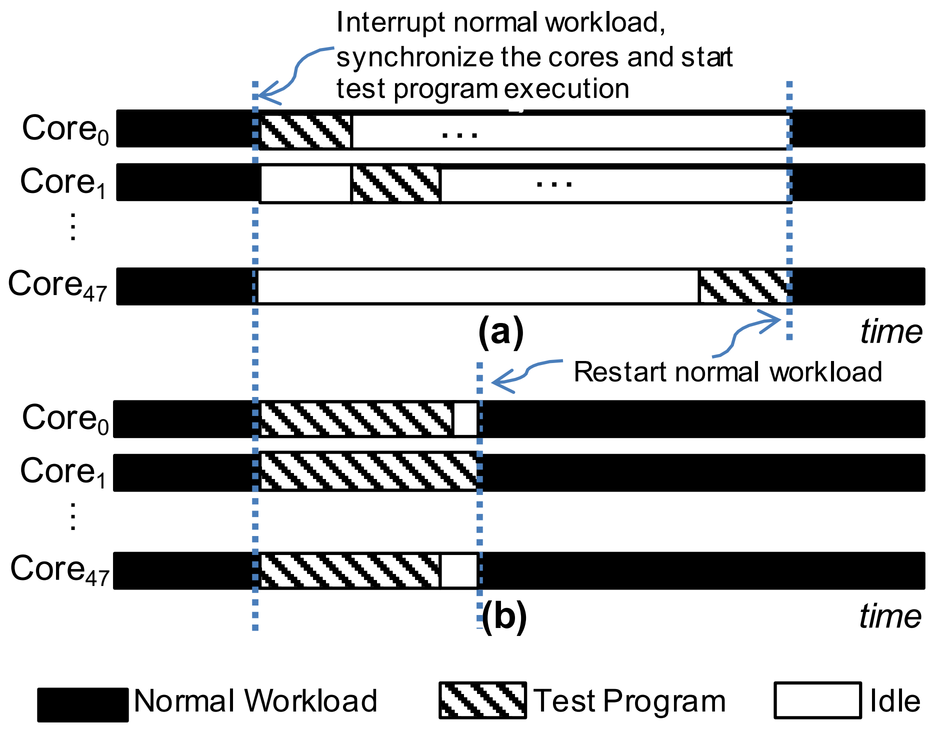


Figure 70: (a) Serial test program, (b) Naïve parallel test program.

Table 23 compares the total test execution time of the naïve parallel method and the “serial” method for the two sample test programs for different test data sizes. In the serial method shown in Figure 70, the processor cores execute the test program one after the other, while the remaining cores remain idle. As mentioned above, the serial method is senseless (when test time reduction is the objective) since the test programs running on the different cores have no dependencies, and thus they can simply run in parallel. We simply mention here the serial method as a baseline method to indicate that the speedup achieved in some cases by the naïve parallel method is far below the theoretical maximum speedup, i.e. 48X in the case of the 48-core SCC chip. In both serial and naïve parallel methods, test data are stored in the shared memory of the SCC in order to be directly accessible by all cores. Intel supports the configuration of shared memory either as cacheable or non-cacheable memory. The default configuration of shared memory is non-cacheable which releases the programmers from handling the data coherency problem. When shared memory is configured as cacheable the programmer is responsible to maintain data coherency flushing the caches when needed. In, [33] we consider both cases for the sake of completeness. The experimental results show that in the case of the memory-intensive LAA test program the speedup is less than 7X and 11X for non-cacheable and cacheable shared memory respectively, while in the case of the CPU-intensive LFSR test program the speedup is close to the theoretical maximum. Thus, in the rest of this study, we focus on the effective parallelization of the LAA test program since the LFSR test program is embarrassingly parallel.

Table 23: Naïve parallel method vs. serial method (Times in 10^6 cycles. Numbers in parentheses denote speedup against serial execution).

Test Program	Serial Non-cacheable	Serial Cacheable	Naïve Parallel Non-cacheable	Naïve Parallel Cacheable
LAA - 192KB	317.8	73.8	47.3 (6.7X)	7.4 (10X)
LAA - 384KB	634.7	145.8	95.1 (6.7X)	14.0 (10.4X)
LFSR – 384KB	105.3		2.5 (42.1X)	
LFSR – 3840KB	1047.8		23.1 (45.4X)	

The main reason of the low speedup of the naïve parallel method for the LAA test program is the excessive traffic it produces in the interconnection network and the DRAM controllers especially during its test preparation phase, which is its most memory-intensive phase. Previous works have measured the performance degradation of the system when multiple cores run in parallel increasing the traffic congestion in the mesh. In study [119], the authors assessed the memory performance of the SCC when multiple cores access the same memory controller in parallel. The performance degradation when all 12 cores of a memory domain simultaneously issue memory requests reaches up to 14.2% compared to the single-core performance. In study [120], the authors showed a performance drop up to 27% when all 12 cores of a memory domain run in parallel the same memory intensive test programs. Note that the experimental results in [119] proved that running the benchmark in all 12 cores of a single memory domain produces the same memory performance results compared to running it in all 48 cores due to the completely different memory domains. However, in our case running the test program in all 48 cores in parallel increases significantly the total execution time compared to 12 cores running in parallel. This is due to the higher DRAM memory latencies in the case of 48 cores. Although test vectors of the LAA test program are equally distributed to the memory regions of the four DRAM controllers, the memory requests of the cores must travel a longer distance to reach the remote controllers, thus increasing the average memory latency of most cores.

<u>Memory Performance Issues:</u>

The development of an efficient parallelization method for online test programs requires first a thorough analysis of the main memory performance issues of the SCC architecture, its bottlenecks and the throughput of the different communication paths. Table 24 summarizes some approximate memory latencies for a core reading a 32-byte cache line [34], [119]. Memory latency of a core is defined as the total delay between sending a memory request and receiving the requested data from memory. The L2 cache hit time is 18 core cycles, the local MPB access time bypassing the L2 cache is 15 core cycles, the delay of a router forwarding request is 4 mesh clock cycles, and the DRAM access time is 46 memory clock cycles. In Table 24, *n* denotes the number of hops in the route between the core and the MPB or its memory controller. A brief analysis of the memory latencies shows that the message passing buffer provides a significantly faster means to transfer data between two cores than using the shared off-chip DRAM. Thus, parallelization should focus on the maximum utilization of these fast on-chip communication paths through MPBs.

Table 24: Memory latencies in SCC.

Memory type	Approximate latency (cycles)
L2	$18\times T_{core}$
Local MPB (bypass L2 cache)	$15\times T_{core}$
Remote MPB	$45\times T_{core} + 4\times n\times 2\times T_{mesh}$
Off-chip DRAM	$40\times T_{core} + 4\times n\times 2\times T_{mesh}+ 46\times T_{ram}$

We also carried out a set of experiments to identify the most efficient way to load test vectors from the off-chip DRAM. We measure the execution time of a simple program loading 8KB data (test patterns) from the shared (non-cacheable) memory or the private memory or transferring 8KB data between cores through the MPB. Figure 71 shows the execution time when 1 core (or 2 cores in the case of MPB), 12 cores in the same memory domain or all 48 cores run in parallel the simple program.

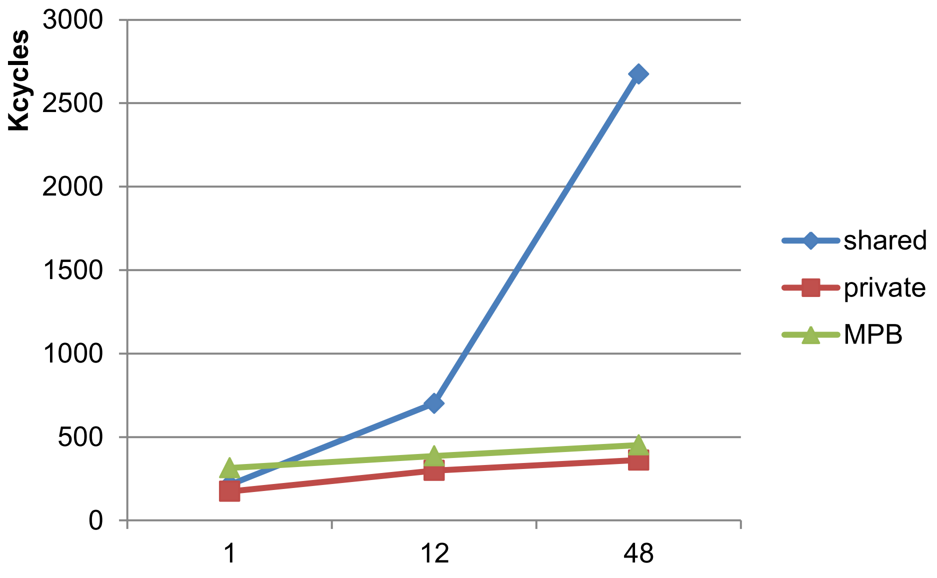


Figure 71: Execution times (in core clock cycles) of loading 8KB test data. Clock frequency settings: tile/mesh/DDR-533MHz/800MHz/800MHz.

Based on the experimental results and the theoretical analysis of the memory performance we concluded that a parallelization method should handle the test preparation phase of the LAA program applying the following guidelines:

- Use the private instead of the non-cacheable shared memory region to store test patterns. Both the serial and the naïve parallel programs store the test patterns in the shared memory and not in the cores' private memories. This allows us to store a single copy of test data in the off-chip DRAM accessible by all cores (a major requirement for online testing); otherwise each core must store its own copy of test data in its private memory. On the other hand, the private memory access is much faster than the non-cacheable shared memory access. Thus, a more efficient parallel test program should partition the test patterns and store them in the private memory regions and assign to each core the task of loading and distributing to the others its test data portion.

- Use message passing instead of off-chip DRAM to share test data between cores: When a core wants to share its test patterns stored in its local caches with other cores located in the same or different memory domain, the most efficient way as shown by the experimental results is to use the message passing buffer. To achieve the highest bandwidth of the internal mesh, the route of test data between the cores must conform to the XY routing rule in order to reduce the number of hops and consequently the total memory latency. Also, the test data exchange must be done in chunk sizes that exploit the available space of the MPBs in order to reduce the communication overhead and avoid misses in the local caches.

Proposed Parallelization Method:

Based on the previous memory performance analysis, we propose an effective method for the parallelization of the LAA test program. The proposed method shown in Figure 72 focuses on the efficient parallel execution of the test preparation phase. The test patterns are divided into 48 segments each one assigned to the private memory region of a core. The LAA test program is divided into two phases. First, all cores load in parallel the test patterns from their private memories, apply the tests and accumulate the responses. Subsequently, each core copies the corresponding test patterns from the local MPBs of the other 47 cores and applies/accumulates the tests. It is essential that in each cycle of the second phase each MPB serves the memory requests of only one core in order to limit the traffic congestion in the mesh and the routers. The rationale of the proposed method is that having every core to read test patterns from its private memory and distribute them to the other cores is the most efficient way to parallelize the test preparation phase of the LAA program.

Regarding the LFSR test program, its test preparation phase cannot be parallelized in a more efficient way since the time each core requires to run the LFSR code to generate a certain number of test patterns is shorter than the time to copy these test patterns from the local MPB of an adjacent core. Thus, a second improvement in the parallelization of the entire online test program could be the parallel execution of memory-intensive test programs (e.g. LAA test) and CPU-intensive test programs (i.e. LFSR test). As shown in Figure 73, the LAA-LFSR test program is divided into steps. First, all cores load in parallel the test patterns from their private memories and run the tests similar to the first phase of the parallel LAA program. Subsequently, one core of the tile runs the test application and response accumulation phases of the LAA program while the other core runs the LFSR program. Finally, the two cores of the tile change roles executing the opposite test programs. In the case that one core of the tile completes earlier the execution of its test during the second phase it can proceed to the third phase. The rationale of the proposed method is to balance the memory-intensive and the CPU-intensive test programs to avoid high traffic congestion in the mesh and the routers.

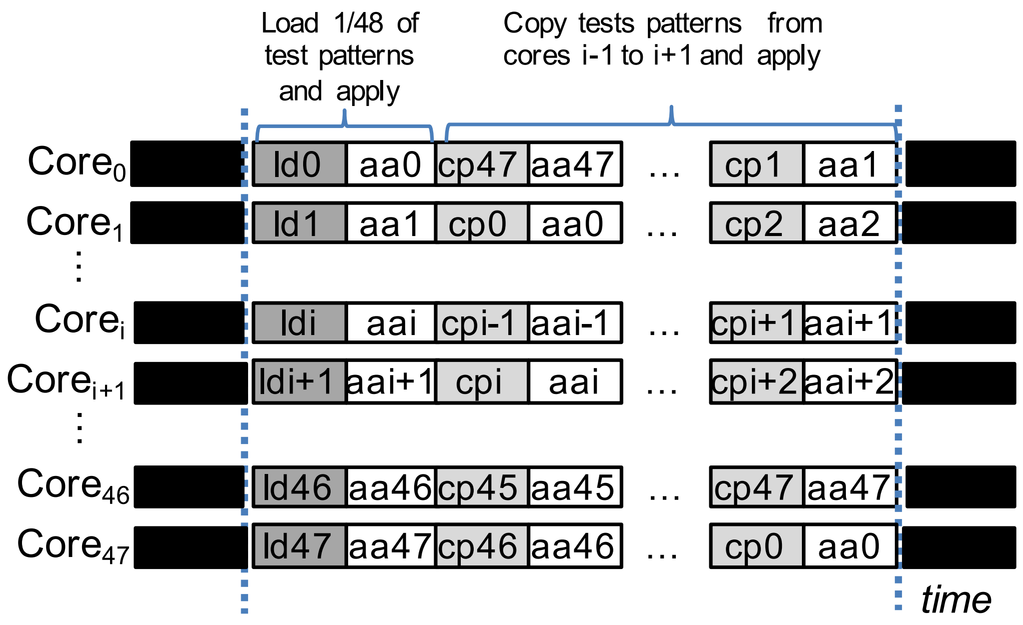

Load 1/48 of test patterns and apply
Copy tests patterns from cores i-1 to i+1 and apply
Core0
Core1
Corei
Corei+1
Core46
Core47
ld0 aa0 cp47 aa47 … cp1 aa1
ld1 aa1 cp0 aa0 … cp2 aa2
ldi aai cpi-1 aai-1 … cpi+1 aai+1
ldi+1 aai+1 cpi aai … cpi+2 aai+2
ld46 aa46 cp45 aa45 … cp47 aa47
ld47 aa47 cp46 aa46 … cp0 aa0
time

Figure 72: The proposed parallelization method of LAA test program (ldi: core i loads test patterns from its private memory, aai: core applies and accumulates the test pattern segment i, cpi: core copies patterns from the MPB of core i).

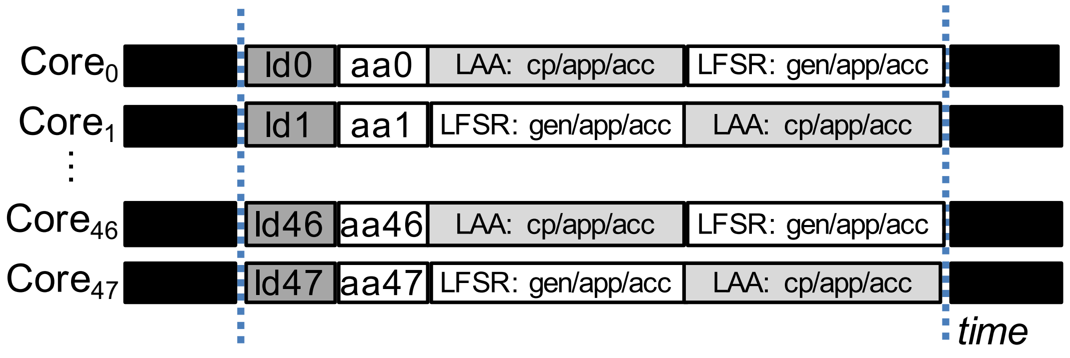

Core0
Core1
Core46
Core47
ld0 aa0 LAA: cp/app/acc LFSR: gen/app/acc
ld1 aa1 LFSR: gen/app/acc LAA: cp/app/acc
ld46 aa46 LAA: cp/app/acc LFSR: gen/app/acc
ld47 aa47 LFSR: gen/app/acc LAA: cp/app/acc
time

Figure 73: The proposed parallelization method of LAA-LFSR test program (ldi: core i loads test patterns from its private memory, cp/app/acc: core copies, applies & accumulates test patterns).

4.1.2 Experimental Results

The experimental results demonstrate the efficiency of the proposed parallelization method. Table 25 presents the execution time of 12 cores (all in the same quadrant) and 48 cores running the LAA test program for 192KB test data (16KB per core) and 384 KB test data (8KB per core), respectively. The test program has been parallelized according to the proposed method. Different chunk sizes are evaluated for the transfer of test data through the MPBs. The speedup of the proposed method against the “fast” cacheable serial method is up to 5.9X and 38.2X for 12 and 48 cores running the LAA test program, while the optimum chunk size in both cases is 2KB. The case of 12 cores running the test program is interesting because it emulates a semi non-intrusive approach, where only one quadrant is set out-of-service and its cores run the test program. The extremely high speed up of the proposed method against the “slow” serial method is due to the use of cacheable memories (i.e. both private DRAM and MPBs are cacheable memories) instead of non-cacheable shared memory.

Table 25: Execution time of the proposed parallel method of LAA.

Cores/ Test data	Chunk size	Proposed method (10^6 cycles)	Speedup (Non-cacheable Serial/Proposed)	Speedup (Cacheable Serial/Proposed)
12 cores/ 192KB	1KB	3.6	22.3X	5.3X
	2KB	3.2	24.6X	5.9X
	4KB	3.4	23.0X	5.5X
	8KB	3.6	21.8X	5.2X
48 cores/ 384KB	1KB	4.3	149.3X	34.3X
	2KB	3.8	166.2X	38.2X
	4KB	4.0	160.6X	36.9X
	8KB	4.0	156.7X	36.0X

Table 26 presents the execution time of 12 cores (all in the same quadrant) and 48 cores running the combined LAA/LFSR test program. LAA test program applies 192KB (12 cores) and 384KB (48 cores) test data, while LFSR program applies 10 times more pseudorandom test data: 1920KB in 12 cores and 3840KB in 48. The transfer of data through the MPBs is done using 8KB chunk. In column 2, cores run first the proposed parallel version of LAA test program and after that the naïve parallel version of the LFSR test program, while in column 3, the two test programs are executed in parallel as shown in Figure 73. The proposed parallel LAA/LFSR method achieves up to 7% improvement over the proposed parallel LAA+naïve LFSR and up to 47.6X speedup over the serial approach.

Table 26: Execution time of the parallel execution of LAA and LFSR test programs.

Cores	Proposed LAA+ Naïve LFSR (10^6 cycles)	Parallel LAA/LFSR (10^6 cycles)	Improve-ment	Speedup (Cacheable Serial/Parallel)
12	14.6	13.9	4%	10.8X
48	26.9	25.1	7%	47.6X

4.1.3 Related work

Software-based testing techniques have gained increasing acceptance for microprocessor error detection during the last. Previous approaches [121] have studied the feasibility of software-based testing for on-line periodic error detection and investigated the best trade-off between fault detection latency and system performance degradation. Recent approaches [122], [123], [124] have studied the acceleration of software-based testing in multiprocessor chips. In [122], the authors consider the application of software-based testing on symmetric shared-memory multiprocessors (SMP) with a small number of processor cores and propose a methodology that reduces bus contentions and data cache invalidations in order to reduce test execution time. The methodology has been applied to various SMP benchmarks with up to 8 cores. The authors in [123] exploit both core-level and thread-level execution parallelism and schedule the test threads into the processor hardware threads to reduce the test execution time and increase the fault coverage. They have demonstrated the methodology in a multithread processor architecture consisting of 8 cores and 32 threads. All the above approaches have studied the scheduling and acceleration of online test programs for various multicore architectures, which however present significant differences compared with the state-of-the-art many-core architectures, especially in the memory subsystem. Finally, the authors in [124] propose test application time reduction through selective software-based testing which monitors system activity of the functional modules of the cores and tests only the strained modules, while under-utilized modules are only sporadically tested. The methodology has been demonstrated in a 16-core CMP architecture using a full-system, simulation framework. The selective testing approach reduces significantly the system performance overhead, however, memory performance issues arisen due to the parallel execution of test programs in multiple cores that have not been considered. None of the previous approaches have addressed the problem of test program parallelization that is the goal of study [33]. In [120], a preliminary discussion of the online test programs scheduling problem in a many-core architecture is presented.

4.2 Statistical analysis to predict the safe voltage margins in multicore CPUs for energy efficiency

During chip fabrication, process variations can affect transistor dimensions (length, width, oxide thickness etc. [125]) which have direct impact on the threshold voltage of a MOS device [126]. As technology scales, the percentage of these variations compared to the overall transistor size increases and raises major concerns for designers, who aim to improve energy efficiency. This variation is classified as static variation and remains constant after fabrication. On top of that, transistor aging and dynamic variation in supply voltage and temperature, caused by different workload interactions, is also of primary importance. Both static and dynamic variations lead microprocessor architects to apply conservative guardbands (operating voltage and frequency settings) to avoid timing failures and guarantee correct operation, even in the worst-case conditions excited by unknown workloads [127] [128].

However, these guardbands impede the low power consumption and the high performance, which can be derived by reducing the supply voltage and increasing the operation frequency, respectively. To bridge the gap between energy efficiency and performance improvements, several hardware and software techniques have been proposed, such as Dynamic Voltage and Frequency Scaling (DVFS) [129]. The premise of DVFS is that the microprocessor’s workloads as well as the cores’ activity vary. Voltage and frequency-scaling during epochs where peak performance is not required enables a DVFS-capable system to achieve average energy-efficiency gains without affecting peak-performance adversely. At a specific frequency of operation, energy-efficiency gains are limited by guardbands that guarantee correct operation in the presence of dynamic margins.

Several techniques have been proposed [130] [131] that eliminate a subset of these guardbands for efficiency gains over and above what is dictated by design guardbands. However, all of these techniques are associated with significant area, design, test and measurement overheads that limit its application in the general case. For instance, in the Razor technique [130], support for timing-error detection and correction has to be explicitly designed into the processor micro-architecture which has significant verification overheads. Similarly, in adaptive-clocking approaches [131], extensive test and measurement effort is required for system sign-off. Ensuring the eventual success of these techniques requires a deep understanding of dynamic margins and their manifestation during normal code execution.

Recently, to avoid the extra area and verification overheads, several system-level approaches have been proposed to predict the safe operation limits of the processors [132], [133], [134], [135]. In particular, the authors in [133] and [134] propose a prediction approach that identifies the critical parts of benchmarks in which large voltage noise droops are likely to occur leading to system malfunctions. In the same concept, [12] and [135] propose a firmware implemented approach to predict the lowest safe voltage operation value based on the observation of the errors manifested on caches of an Intel Itanium processor during the execution of benchmarks in off-nominal voltage conditions.

The main idea of these studies is to gain energy without compromising performance based on the great variations of the V_{min} that are observed when: (a) the same benchmark is executed on different cores of the same chip (*core-to-core* variation), (b) different benchmarks run on the same core of the same chip (*workload-to-workload* variation), (c) the same benchmark is executed on different chips (*chip-to-chip* variation). For instance, in Figure 74 we illustrate the *core-to-core* and the *workload-to-workload* variations of the V_{min} that were observed when we ran 10 benchmarks from the SPEC CPU2006 suite [75] on the most robust and the most sensitive cores (those with the lowest and the highest V_{min} on average) of the X-Gene 2 chip [36]. In general, we can observe that the most robust core has always lower V_{min} than the sensitive core for all the benchmarks, with a difference that ranges between 10mV and 35mV. Moreover, the variation of the V_{min} when we execute different benchmarks on the same core is remarkable for both cores (885mV-915mV for the sensitive and 865mV-885mV for the robust core respectively). Finally, the highest observed V_{min} is 915mV and the lowest is 865mV that is translated into 12.83% and 22.10% power savings compared to the case of using the pessimistic nominal value of 980mV as a safe operation limit. Thus, the identification using early prediction of the safe voltage margin per benchmark and per core without sacrificing area and verification time is a major concern for the designers.

Towards this direction, in [35] and [167] we proposed several prediction models based on linear regression to predict the V_{min} and the *Severity* (a metric that reflects the behavior of the cores below their safe V_{min} and before the occurrence of any catastrophic error) of the ARMv8 cores of the X-Gene 2 chip. Our models use as inputs the microprocessor's performance counters values of benchmarks that were collected in nominal voltage conditions execution and the results of the characterization phase when the chip operates in scaled voltage conditions.

In this thesis, we present the results of [35] and [167] concerning the prediction. The linear models we deliver can predict with low inaccuracy (only 0.51%) the V_{min} and the *Severity* of any workload at scaled voltage levels operation using only the performance counters values at the nominal voltage operation mode. These findings can be used as inputs by a software implemented mitigation mechanism leading to power gains from 11.87% up to 20.28% depending on the aggressiveness of the prediction mechanism.

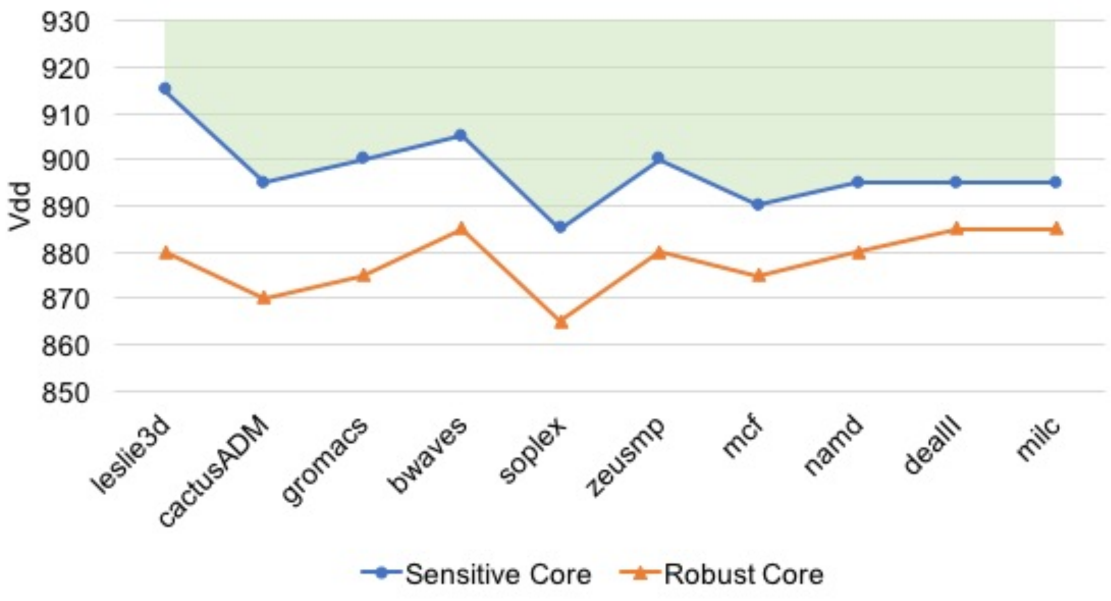


Figure 74: Variation of V_{min} between the most sensitive and most robust cores running 10 SPEC CPU2006 benchmarks on X-Gene 2.

4.2.1 Characterization of the ARM-v8 CPUs

The APM X-Gene 2 micro-server [36] consists of eight 64-bit ARMv8-compliant cores. The X-Gene 2 architecture offers high-end processing performance and capabilities. For example, the X-Gene 2 subsystem features a Power Management processor (PMpro) and a Scalable Lightweight Intelligent Management processor (SLIMpro) to enable breakthrough flexibility in power management, resiliency and end-to-end security for a wide range of applications. The dedicated PMpro processor provides advanced power management capabilities, such as multiple power planes and clock gating, thermal protection circuits, Advanced Configuration Power Interface (ACPI) power management states and external power throttling support. The dedicated SLIMpro processor monitors system sensors, configures system attributes (e.g. regulate supply voltage, change DRAM refresh rate etc.) and accesses all error reporting infrastructure, using an integrated I2C controller as the instrumentation interface between the X-Gene 2 cores and this dedicated processor. SLIMpro can be accessed by the system's running Linux Kernel.

In Figure 75 we present the three independently regulated power domains of X-Gene 2:

- **PMD (Processor Module)**: Each PMD contains two ARMv8 cores. Each of the two cores has separate instruction and data L1 caches, while they share a unified L2 cache. The operating voltage of all four PMDs together can change with a granularity of 5mV beginning from 980mV. While PMDs operate at the same voltage, each PMD can operate in a different frequency. The frequency can range from 300 MHz up to 2.4 GHz at 300 MHz steps.
- **PCP (Processor Complex)/SoC**: It contains the L3 cache, the DRAM controllers, the central switch and the I/O bridge. The PMDs do not belong to the PCP/SoC power domain. The voltage of the PCP/SoC domain can be independently scaled downwards with a granularity of 5mV beginning from 950mV.
- **Standby Power Domain**: This includes the SLIMpro and PMpro microcontrollers and the interfaces for the I2C buses.

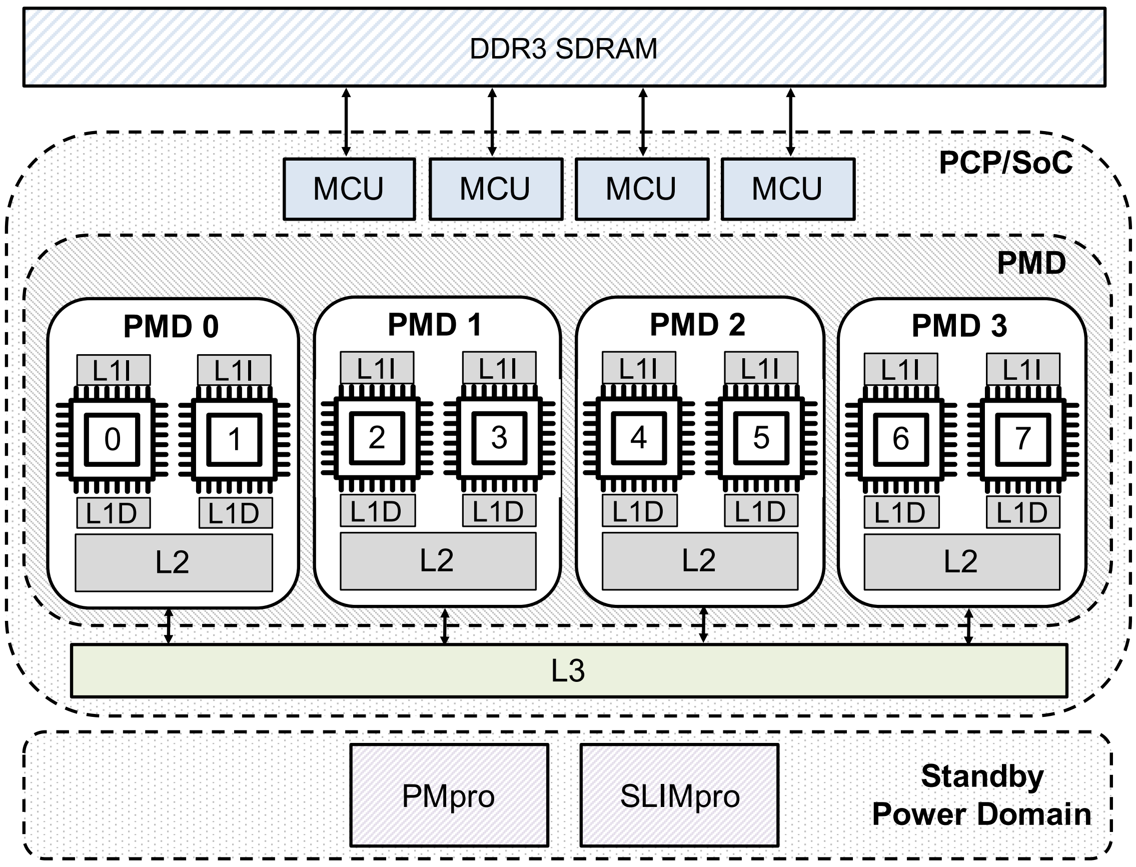


Figure 75: X-Gene 2 block diagram.

Table 27 summarizes the most important architectural and microarchitectural parameters of the APM X-Gene 2 micro-server.

Table 27: Microarchitectural parameters of APM X-Gene 2.

Parameter	Configuration
ISA	ARMv8 (AArch64, AArch32, Thumb)
Pipeline	64-bit OoO (4-issue)
CPU	8 cores
Core clock	2.4 GHz
L1 Instruction cache	32KB per core (Parity Protected)
L1 Data cache	32KB per core (Parity Protected)
L2 cache	256KB per PMD (ECC Protected)
L3 cache	8MB (ECC Protected)
Technology node	28nm
Max TDP	35W

To identify the safe voltage operation margins of all cores of the chip we used a fully automated framework [35] [136] (see Figure 76) to run all the benchmarks from the SPEC CPU2006 suite with different input datasets (40 different programs in total). The primary goals of this framework are: (1) to identify the target system's limits when it operates at scaled voltage and frequency conditions, (2) to record/log the effects of a program's execution under these conditions, and (3) to ensure the integrity of the experimental results in off-nominal operation conditions. The framework provides the following features:

- compares the outcome of the program with the correct output of the program when the system operates in nominal conditions to record Silent Data Corruptions (SDCs),
- monitors the exposed corrected and uncorrected errors from the hardware platform's error reporting mechanisms,
- recognizes when the system is unresponsive and restores it automatically using a Raspberry Pi board,
- monitors system failures (crash reports, kernel hangs, etc.),
- determines the safe, unsafe and non-operating voltage regions for each application for all frequencies, and
- performs massive repeated executions of the same configuration.

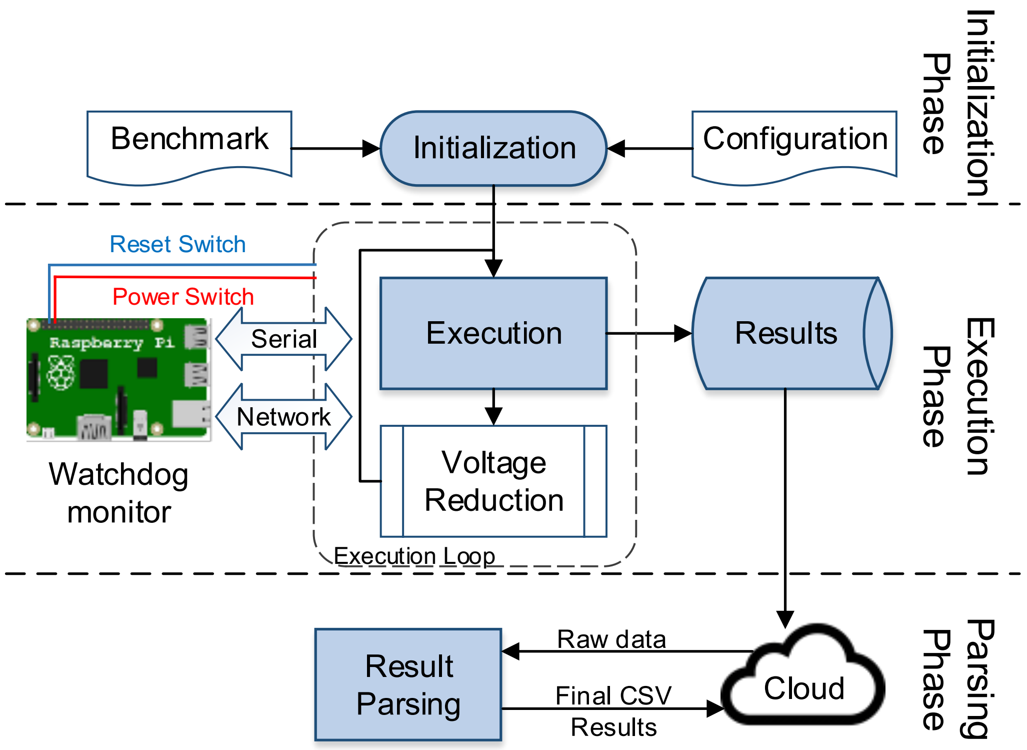


Figure 76: Characterization framework used in [35] and [167].

The framework of Figure 76 consists of three phases: *initialization*, *execution*, and *parsing*. During the *initialization* phase, the user sets the benchmark, the input dataset, the characterization setup (voltage and frequency values) and the cores on which the experiment will be run. The *execution phase* consists of multiple runs of the same benchmark according to the pre-defined characterization setup. Finally, in the *parsing phase*, all log files that are stored during the execution phase are parsed in order to provide a fine-grained classification of the effects observed for each characterization run. The categories that are used for our classification are summarized in Table 28.

Table 28: Effects classification used in [35] and [167].

Effects	Description
NO (Normal Operation)	The benchmark was successfully completed without any indications of failure.
SDC (Silent Data Corruption)	The benchmark was successfully completed, but a mismatch between the program output and the correct output was observed.
CE (Corrected Error)	Errors were detected and corrected by the hardware (provided by Linux EDAC driver [137]).
UE (Uncorrected Error)	Errors were detected, but not corrected by the hardware (provided by Linux EDAC driver [137]).
AC (Application Crash)	The application process was not terminated normally (the exit value of the process was different than zero).
SC (System Crash)	The system was unresponsive; meaning that the X-Gene 2 is not responding or the timeout limit was reached.

For each characterization experiment of our analysis, we ran an entire benchmark on a core with a different voltage value, starting from the nominal value (980mV) and reducing the supply voltage by a step of 5mV. Each characterization experiment was repeated 10 times considering the non-deterministic nature of the results and to maintain the statistical importance of our measurements. The execution time of all these experiments was about 6 months.

The first important general finding from the characterization phase is that three different regions of operation were observed for all the benchmarks of our study. These regions are illustrated in Figure 77 that demonstrates some indicative results for the most robust and sensitive cores of the chip for 10 selected benchmarks from the SPEC CPU2006 suite. The three regions of operation are summarized below:

- Safe region (green): The experiments run to the end having a normal operation without any SDC, error or crash. The last point of this region defines the V_{min}.
- Unsafe region (grey): The experiments generated an abnormal behavior (SDC, corrected/uncorrected errors, application crash) but not a system crash.
- Crash region (red): The experiments lead to system crashes at least in some executions.

In Figure 77, we can observe important divergences of these regions between the two cores highlighting the need of employing accurate prediction schemes to identify these divergences and simultaneously, to increase the total power gains on the chip. In general, the limit of the Crash region for the sensitive core ranges from 890mV to 870mV, while for the robust core ranges between 860mV and 855mV. Also, the width of the Unsafe region is remarkable for both the cores (40mV-85mV for the sensitive and 65mV-105mV for the robust core, respectively).

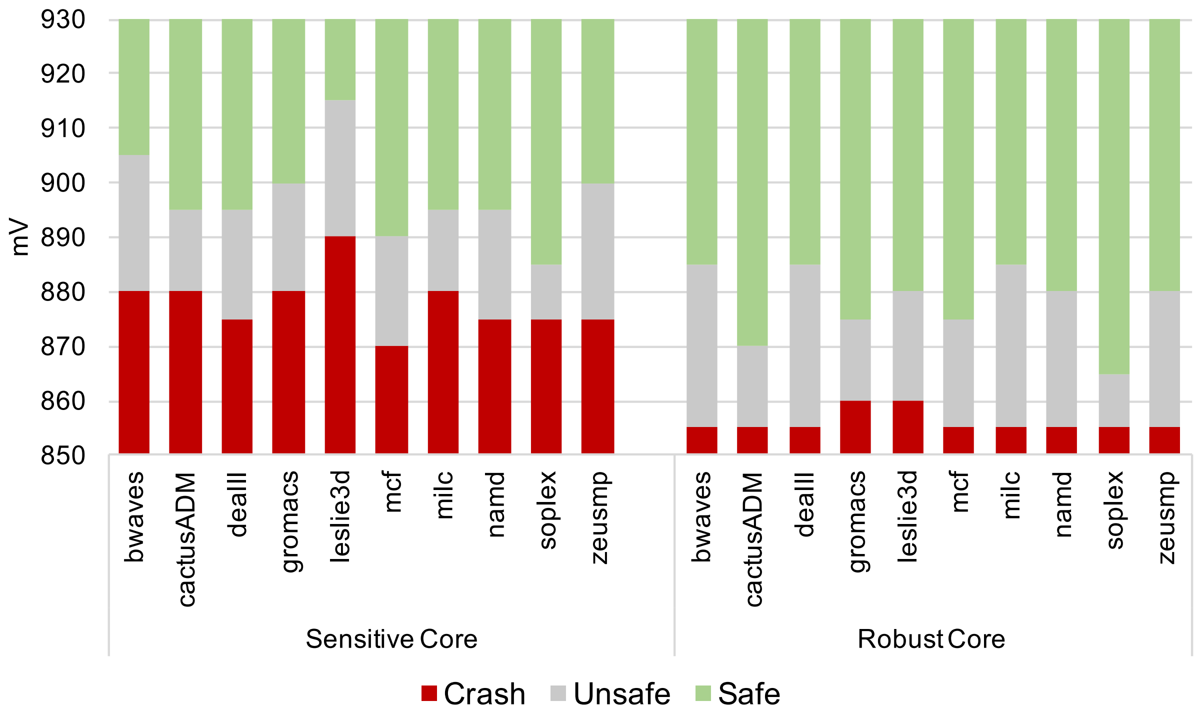


Figure 77: Regions of operation for 10 benchmarks from SPEC CPU2006 suite running on the most robust and most sensitive core of the X-Gene 2.

Another important observation from the characterization phase is that for the case of the X-Gene 2 chip the silent data corruptions can appear at higher voltage levels than corrected errors alone for many benchmarks. This is not the case for previous studies on Intel Itanium CPUs in which it was observed that by reducing the voltage, the number of corrected errors increases gradually for many voltage steps of voltage reduction before the system exposes SDCs, uncorrected errors, or crashes [12] [135]. The reason of this difference between the two chips is that the Itanium chips present more robust behavior to timing failures due to the existence of the continuous clock-path de-skewing mechanism during their normal operation [138], a mechanism that is not implemented in the case of the X-Gene 2 chips. Consequently, due to the occurrence of SDCs first, it is not possible to guide the voltage speculation for prediction based only on the manifested errors.

To overcome this issue concerning the prediction, we define the Severity function (*Sv*) that is presented in the following equation (where *v* is the voltage value):

$$S_v = W_{SDC} \cdot \frac{SDC}{N} + W_{CE} \cdot \frac{CE}{N} + W_{UE} \cdot \frac{UE}{N} + W_{AC} \cdot \frac{AC}{N} + W_{SC} \cdot \frac{SC}{N} \qquad (18)$$

In equation (18), *N* is the total number of experiments for a particular voltage level, while parameters *SDC*, *CE*, *UE*, *AC* and *SC* can take values from 0 to *N* and represent the times that an effect appears to these experiments. Parameters W_{SDC}, W_{CE}, W_{UE}, W_{AC} and W_{SC} correspond to "weights" that can be arbitrarily set to characterize the severity of each effect of Table 28, practically meaning that the higher the weight, the more critical is considered the effect by our function. In [35] and [167], we use the values of Table 29 as weights, but different weight values could be used according to the importance of the effects. In Figure 78, we indicatively illustrate the results for the Severity function when running the *bwaves* benchmark in each core of the chip.

Table 29: Weights of Severity function used in [35] and [167].

Weight	Value
W_{SC}	16
W_{AC}	8
W_{SDC}	4
W_{UE}	2
W_{CE}	1
W_{NO}	0

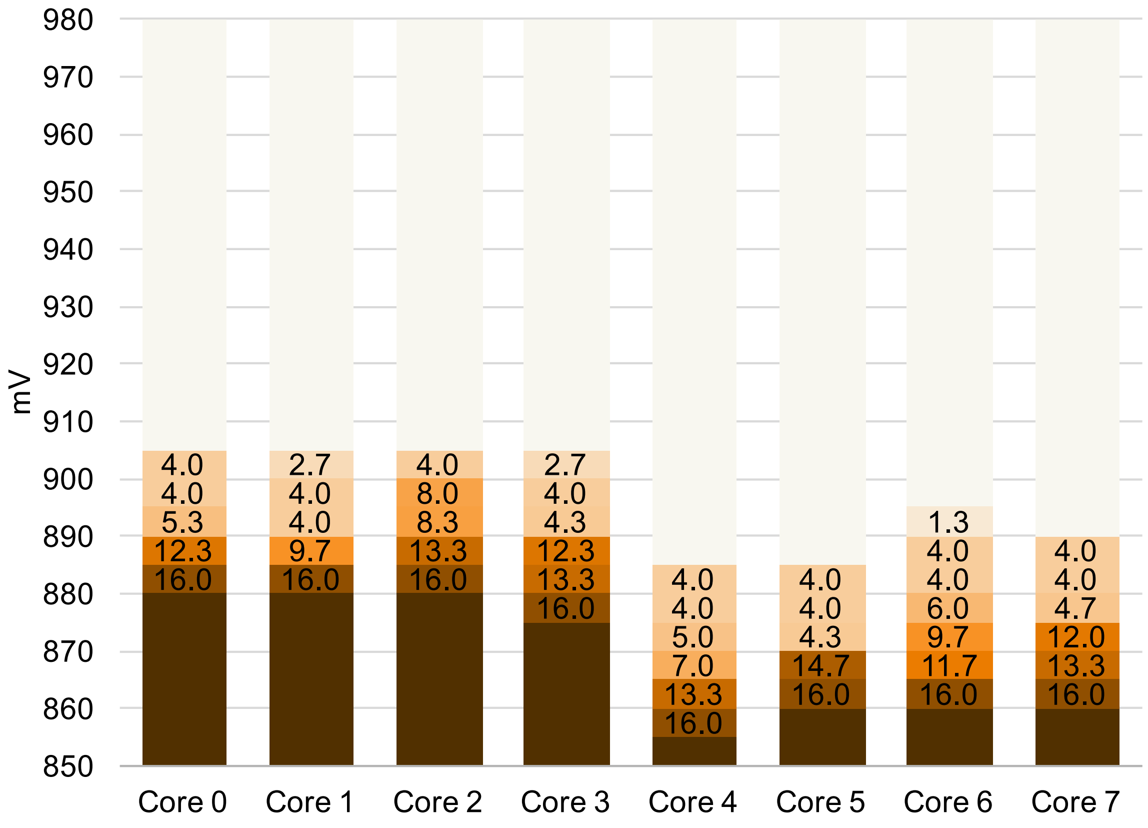


Figure 78: Severity for *bwaves* benchmark on all cores.

To summarize, the *Severity* function is able to:

(i) illustrate the scaling of abnormal behaviors due to voltage reduction,

(ii) aggregate the results that are produced from multiple runs of the same characterization experiment (10 runs in our case),

(iii) quantify the microprocessor's ability to operate beyond nominal conditions and especially beyond the V_{min}, giving the flexibility to the predictor to be more aggressive due to the knowledge of the Unsafe region.

There are many applications and systems that can tolerate SDCs and can benefit from the existence of such an aggressive predictor that targets Severity. These applications can be approximate computing algorithms, video streaming and image processing algorithms, security oriented applications such as jammer attacks detectors, etc. Moreover, if a rollback mechanism to previous checkpoint is implemented in the system, then the detected but uncorrected errors (*UE*) could also be handled, boosting the need of the existence of a more aggressive predictor beyond the safe V_{min}. For instance, assuming a system that can tolerate the *SDCs*, *CEs* and *UEs* (practically meaning that the *Severity* could be equal or less than 7 units according to Table 29), then the power savings that come from the use of a more aggressive prediction scheme that targets Severity or the use of a more conservative prediction scheme that targets V_{min} when we run the *bwaves* benchmark are presented in Figure 79. According to Figure 79, a more aggressive prediction scheme that targets the Severity can lead from 1.10 up to 4.47 percentile units more power savings compared to a prediction that targets the V_{min}.

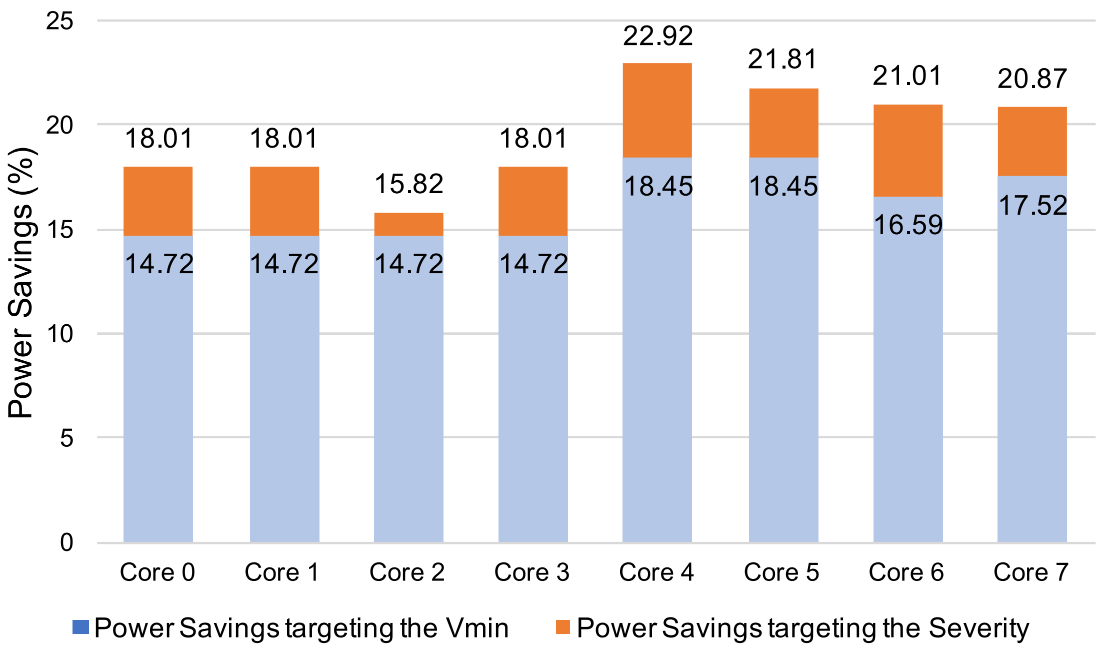


Figure 79: Power savings targeting the V_{min} or using Severity function.

4.2.2 Proposed techniques to predict safe voltage operation margins of ARM-v8 CPUs

Predicting safe voltage regions of the microprocessor using as input the performance counters provided by the system has recently gained the interest of the computer architecture community [12] [133] [135]. In this section, we present the feasibility of predicting the safe V_{min} (for a conservative prediction by taking into account only the V_{min} values for each core and workload), as well as the Severity (a more aggressive prediction below V_{min}) using the microarchitectural events measured for the entire benchmarks execution provided by the X-Gene 2 hardware.

Specifically, we implemented our linear regression models with three different feature selection algorithms aiming to predict both the V_{min} and the Severity in the cores of the X-Gene 2 running all the benchmarks from the SPEC CPU2006 suite with all their inputs (40 programs in total). In this thesis, we present only the most representative cases of our analysis on the most robust (Core 4) and the most sensitive cores (Core 0) of the chip, respectively.

We used linear regression models, which are able to provide high prediction accuracy with a relatively small population of microarchitectural counters. Moreover, linear regression functions of a small number of performance counters can be easily calculated on hardware, while non-linear models are more complex and more time consuming. Therefore, linear models are more suitable for online prediction purposes [139].

In general, regression techniques give the ability to calculate a function to predict a value of the dependent variable from a set of independent variables. Assuming a set of $x_1, x_2, x_3, ..., x_N$ independent variables and y the dependent variable, the classical linear regression model for y that we use in [35] and [167], is based on the Ordinary Least Squares (OLS) model [140]. Specifically, it yields a set of weights β, one for each predictor variable x, and an error term e:

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_k x_{ki} + e_i \tag{19}$$

In this formula, y_i is the i^{th} response value (in our case the V_{min} or the Severity value), x_{ji} is the j^{th} microarchitectural counter (e.g. the L1 Cache Accesses) evaluated at the i^{th} observation, and e_i is the i^{th} statistical error. The goal of the regression analysis is to find the optimal values of the coefficients $\beta_1, \beta_2, \beta_3, ..., \beta_k$ so as to minimize the sum of the squares of the differences between the observed responses (values of the predicted variable) in the given dataset and those predicted by a linear function of a set of explanatory variables.

This analysis also provides a “coefficient of determination” (R^2) that indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. The larger the values of R^2, the better fit the model provides, while the best fit exists when R^2 is equal to 1. The R^2 can be 0 when the model predicts the expected value disregarding the input features or even negative (because the model can be arbitrary worse). R^2 is an important indicator for linear regression analysis in order to quantify the accuracy of each model. However, to evaluate the accuracy of our prediction model for different test cases, we also use the Root Mean Square Error (RMSE) that represents the deviation between the predicted values and the observed values. The smaller the RMSE the more efficient the prediction model is.

Our analysis is based on four steps as presented in Figure 80: (i) offline characterization that reveals the safe voltage operation margins, (ii) collection of all the performance counters provided by the X-Gene 2 system during nominal conditions, (iii) feature selection of the most important counters that mostly affect the prediction according to the test case (targeting either the V_{min} or the severity), and (iv) training and evaluation of the different test cases on which we implemented linear regression analysis. For the feature selection and the linear regression model of our analysis we used the *sklearn* python libraries provided by [140]. As Figure 80 presents, in phase 1 we perform an extensive characterization, which exposes the regions of operation (Safe, Unsafe, Crash), the V_{min} and the severity values. In phase 2, we perform application profiling for all available performance counters. In phase 3, we train the predictor using the outputs from step 1 and 2, and in phase 4, we make the actual prediction evaluating the estimations using the test dataset. We further analyze the three steps (except for characterization) illustrating also the results of our statistical analysis for the different test cases (targeting both the V_{min} and the severity).

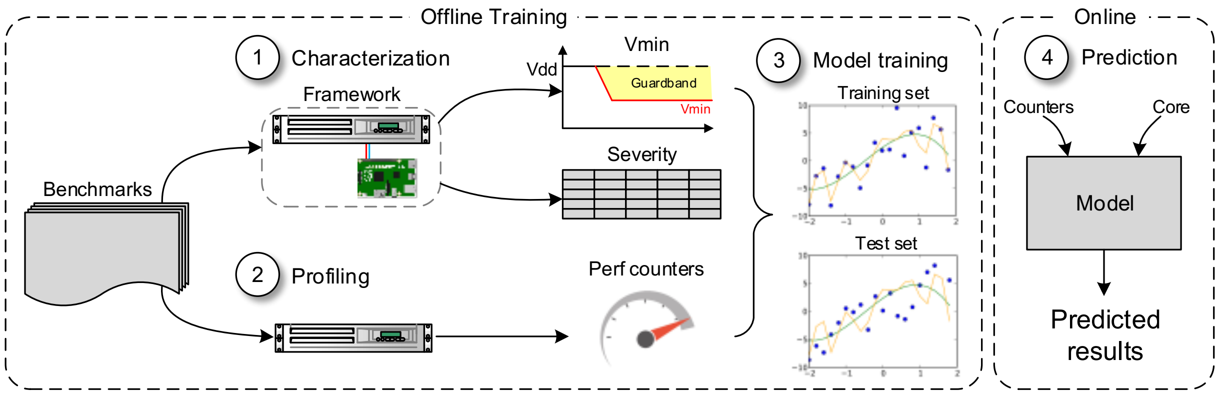


Figure 80: Overview of the prediction flow used in studies [35] and [167].

Collection of all the performance counters:

The X-Gene 2 chip provides 101 performance counters in total (see ANNEX I) which report microarchitectural events of the entire system for individual cores, for the memory hierarchy (accesses and misses of all caches, TLB and page walks levels, unaligned accesses, prefetches, etc.), the pipeline (flushes, mispredictions, etc.), and the system (bus accesses, etc.). The selection of the most appropriate performance counters in order to ensure the accuracy of the final estimations that each model delivers is of major importance for our statistical analysis. We collect the performance counters using the Linux Perf tool [141].

Selecting counters for each test case:

To reduce the population of performance counters used by our prediction models targeting either the V_{min} or the Severity, we implemented three different feature selection techniques:

- **F_regression**: This approach is based on *F-distribution* and computes the correlation of each independent feature *X*[:, *i*] with the dependent variable *y*, according to the following formula:

$$\rho_i = \frac{(X[:,i] - mean(X[:,i])) * (y - mean(y))}{std(X[:,i]) * std(y)} \tag{20}$$

Then, for each feature, the *F-value* is computed according to equation (21), where *n* is the population of *y* value. Finally, using the *F-value*, the associated *p-value* of each feature is determined [142]. Note, that the *p-values* are predefined according to the *F-values* of the *F-distribution*. The lower the *p-value* and the higher the *F-value*, the more efficient the model becomes when we include this feature in the model.

$$F_i = \frac{\rho_i^2}{1 - \rho_i^2} * (n-1) \tag{21}$$

The *F-values* and the *p-values* after being calculated for all the features, they are sorted and finally, the best *k* features with the best scores are selected and used as input from the linear regression model.

- **Recursive Feature Elimination (*RFE*)**: Given an external estimator that assigns initial weights to all features (e.g., a linear regression model), the goal of *RFE* is to select features by recursively considering smaller and smaller sets of features [140]. First, the estimator is trained on the initial set of features, and weights are assigned to each one of them according to their correlation coefficient. Then, features whose absolute weights are the smallest are pruned from the current set of features. This procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.
- **Polynomial Feature Transformation (*Polynomial*)**: In order to evaluate the correlation between two or more features, we implemented polynomial feature transformation that generates a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to a specified degree. For example, if an input sample of features is two dimensional and consists of two features [x_1, x_2], the new polynomial features matrix of degree-2 will be [1, x_1, x_2, x_1^2, x_1x_2, x_2^2] [140]. We implemented the polynomial feature transformation for all the cases targeting either the V_{min} or the Severity with a degree of correlation equal to 2. Our experiments revealed that a degree of correlation greater than 2 cannot provide better efficiency to our prediction models. After the polynomial feature transformation, we use either *f_regression* or *RFE* feature selection to select the best new features for our linear regression models.

Training and evaluating the test cases:

We analyze the results of our analysis for the three cases targeting both the V_{min} and the severity values for the most robust (Core 4) and the most sensitive core (Core 0) of the chip. In Section 4.2.3, we present the four most representative test cases of our study on chip, which are:

- *1st case*: Predict V_{min} of the most sensitive core
- *2nd case*: Predict V_{min} of the most robust core
- *3rd case*: Predict severity of the most sensitive core
- *4th case*: Predict severity of the most robust core

To evaluate the efficiency of our prediction model (apart from the R^2 and the RMSE) we used as baseline model the naïve prediction, which is the average of the target values (V_{min} or Severity) of the samples of the training set.

In our statistical analysis and depending on targeting either the V_{min} or the Severity, we use different *samples* that correspond to the information vectors of the values of the dependent and the independent variables that were used to train and test our models. Specifically, all the independent variables of the *samples* of this study that were used to predict the V_{min} consist of the microarchitectural counters from the entire execution of the benchmarks normalized per kilo committed architectural instructions, while the independent variables of the *samples* that were used to predict Severity consist of the same microarchitectural counters and also of the voltage values of each voltage reduction step that come from the characterization phase. Furthermore, to avoid biasing the prediction results, all the values of the microarchitectural counters of all our *samples* were scaled according to their minimum and maximum observed values, getting a final value with range [0, 1].

For all the experiments, we split the *samples* into 80% training and 20% test datasets (a very typical choice in the evaluation of statistical methods), while to evaluate the accuracy of our models and to avoid over-fitting, we cross-validated our results with different sets of train and test datasets (using 8000K combinations of train and test *samples* for all the datasets of our experiments). Finally, the total population of *samples* that were used to predict the V_{min} and the Severity is 320 and 800 respectively.

4.2.3 Experimental Results

In this section, we present the results for the four test cases presented in the previous section, of implementing our linear regression models with the three different feature selection algorithms targeting both the V_{min} and the Severity in the most robust (Core 4) and the most sensitive (Core 0) cores of the X-Gene 2 when we run all the benchmarks from the SPEC CPU2006 suite with all their inputs (40 programs in total). For all our experiments, we evaluate the accuracy of each prediction model when we use from one up to ten most important features selected for the three different feature selection techniques in order to feed our model. Finally, for the best population of features that provide the highest accuracy (the lowest RMSE), we present the final prediction model according to equation (19).

1st case: Predict Vmin of the most sensitive core

In Figure 81, we illustrate the accuracy results (in terms of RMSE that is measured in *mV*) of all our models when we target the V_{min} of the most sensitive core. The black dotted line presents the accuracy of the baseline (naïve) model which is the average values of the training dataset for prediction, while each set of bars represents the accuracy of the different models by using different population of features for the prediction.

We can observe that by using more than 4 features, all the methods are less accurate than the baseline model. Moreover, the linear regression model using only *RFE* is less accurate compared to the baseline model for all the cases. In general, the linear regression that is accompanied by polynomial transformation with up to 4 selected features using *f_regression* gives better accuracy than the other methods for all the cases. The best accuracy (with RMSE equal to 5.0108mV) was observed when we use the polynomial transformation with *f_regression* selection and only 4 selected polynomial features. This model is illustrated in Table 30 and represents the final prediction model according to equation (19). Moreover, the R^2 that was measured for this prediction model is relatively high (close to 0.75) indicating a good fit of the data to the model.

Our model finally delivers very high accuracy compared to the real values of our experiments (with only 5mV inaccuracy that corresponds to a single voltage step reduction that is supported by the machine). The potential power savings of using this scenario of prediction and adding a very small guardband of only 5mV that is equal to our predicted inaccuracy is 11.87% compared to the case of using a very pessimistic guardband equal to the nominal voltage value.

In general, the most important features that are finally used by all the models of this study (described in in this section) can lead to large voltage droops (BTB mispredictions, decode stalls, exceptions, branches), while some others (integer and FP operations) to timing errors in the pipeline of the X-Gene 2 that does not implement a de-skewing mechanism.

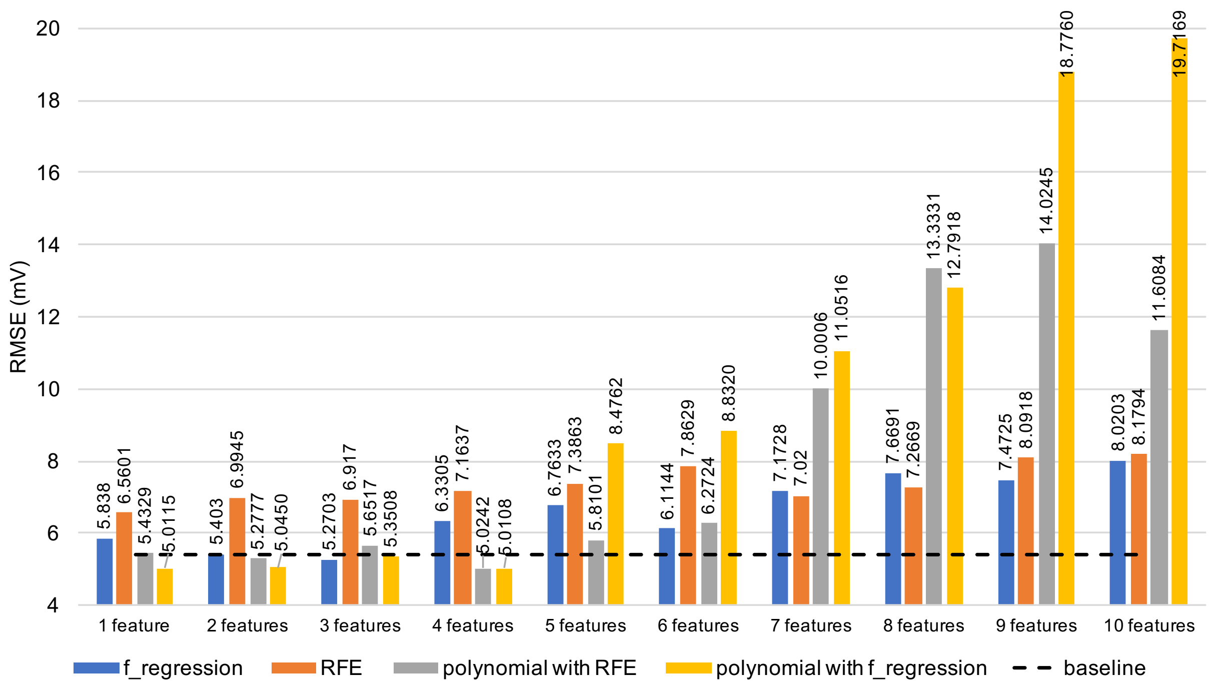


Figure 81: Accuracy of predicting the V_{min} of the most sensitive core.

Table 30: V_{min} prediction model of the most sensitive core of X-Gene 2.

Symbol definition of microarchitectural counters used by the model	Prediction Model (with 4 polynomial features)
L2 data prefetch request (L2_pref)	897.90 + (23.01* *L2_pref*) + (54.20 * *BTB_miss* * *FP*) – (7.15 * *INT* * *L2_pref*) – (58.84 * *FP* * *Indirect_br*)
BTB mispredictions (BTB_miss)	
Floating point operation (FP)	
Integer data processing (*INT*)	
Indirect braches (Indirect_br)	

2nd case: Predict Vmin of the most robust core

The results of the accuracy of the different prediction models that target the V_{min} of the most robust core are illustrated in Figure 82. All the models with more than 4 features or less than 3 features are less accurate than the baseline model. The best accuracy is observed for linear regression after applying polynomial transformation with *f_regression* and using only 3 polynomial features with RMSE equal to 5.0922mV (this model is presented in Table 31). The R^2 is again relatively high (0.70) for this model. Finally, the potential power savings of this model is 17.52% compared to the case of using the nominal voltage value.

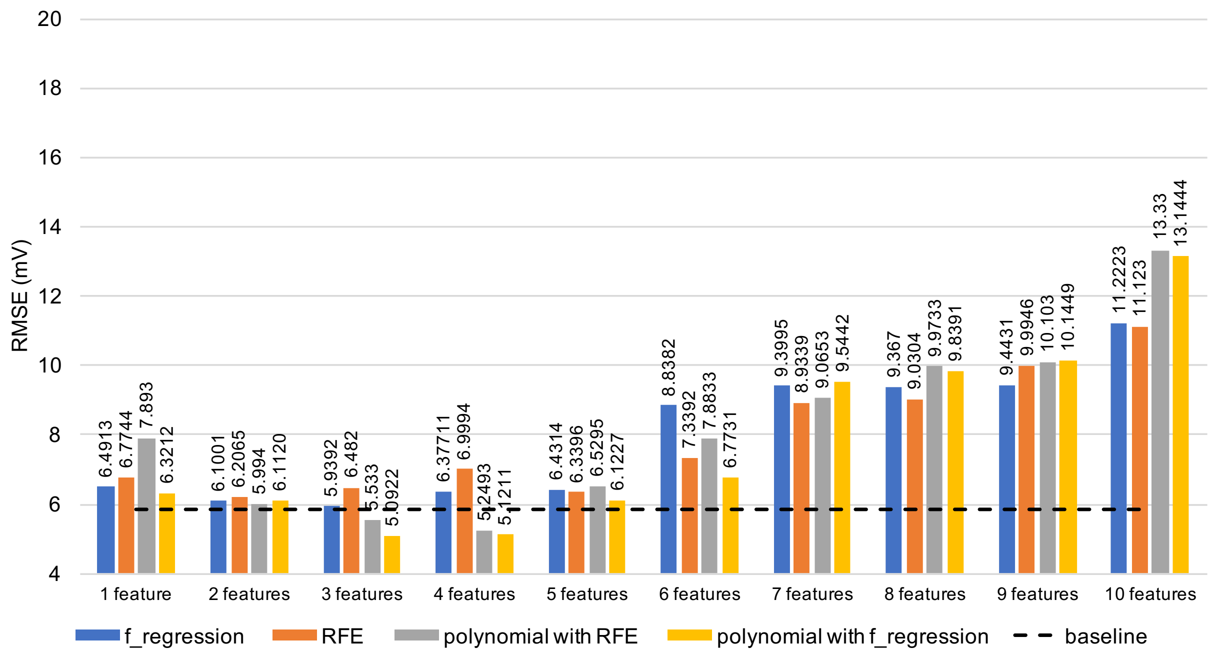


Figure 82: Accuracy of predicting the V_{min} of the most robust core.

Table 31: V_{min} prediction model of the most robust core of X-Gene 2.

Symbol definition of microarchitectural counters used by the model	Prediction Model (with 3 polynomial features)
L2 data prefetch request (L2_pref)	898.24 + (27.36 * *L2_pref*) + (61.24 * *BTB_miss * FP)* – (8.96 * *INT* * *L2_pref*)
BTB mispredictions (BTB_miss)	
Floating point operation (FP)	
Integer data processing (*INT*)	

3rd case: Predict severity of the most sensitive core

Figure 83 presents the results of the accuracy (in terms of RMSE that is measured in *Severity units* as was defined in equation 18) when we target the Severity of the most sensitive core of the chip.

We observe that both the linear regression with a simple *f_regression* feature selection mechanism or the model in which we firstly apply a polynomial transformation and then we select the best features with *f_regression* outperform the baseline model for all the cases of different populations of selected features. On the other hand, the simple *RFE* feature selection and the model with the polynomial transformation and *RFE* feature selection are less accurate than the baseline model for all our experiments.

The best accuracy (with RMSE equal to 2.7223 Severity units) is observed for the model that uses *f_regression* feature selection with 3 features (this model is presented in Table 32). The R^2 for this model is very high (equal to 0.92) that indicates that our linear model with the selected features is able to predict a large proportion of the variance in the dependent variable.

The potential power savings of using this aggressive prediction mechanism that targets the Severity including the measured error are 16.59% compared to the case of using the pessimistic nominal voltage value for protection. These savings correspond to 39.76% more power savings compared to the gains of using the less aggressive prediction that targets the V_{min} value of the same core, indicating the need to move forward to more aggressive prediction schemes in order to save more energy.

Finally, Figure 84 illustrates intuitively the efficiency of our proposed model that targets the Severity of the most sensitive core of X-Gene 2 chip. The blue line in this figure represents the ideal efficiency where the real and the predicted values of the test dataset are equal. The closer the black dots are to the blue line, the more efficient the prediction model is.

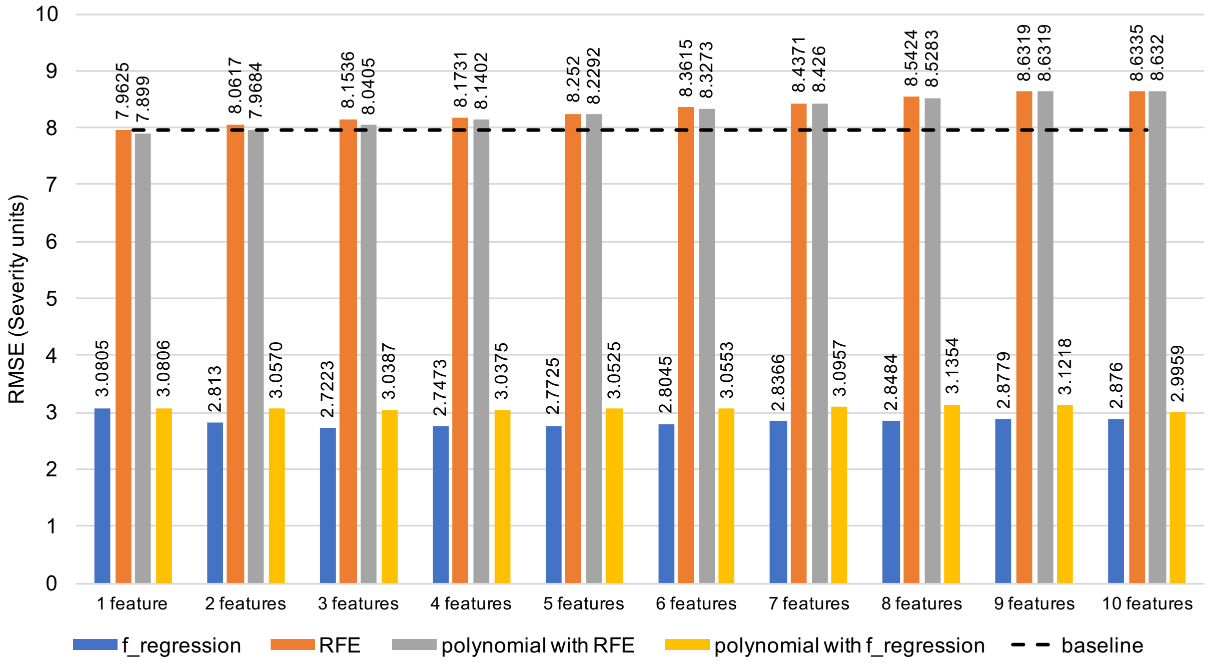


Figure 83: Accuracy of predicting the *Severity* of the most sensitive core.

Table 32: Severity prediction model of the most sensitive core of X-Gene 2.

Symbol definition of microarchitectural counters used by the model	Prediction Model (with 3 features)
Voltage (*Volt*)	20.35 + (3.66 * *Volt*)
L1 data TLB write (*L1D_tlb_write*)	– (2.34 * *L1D_tlb_write*)
Decode stalls (*dec_stalls*)	– (22.53 * *dec_stalls*)

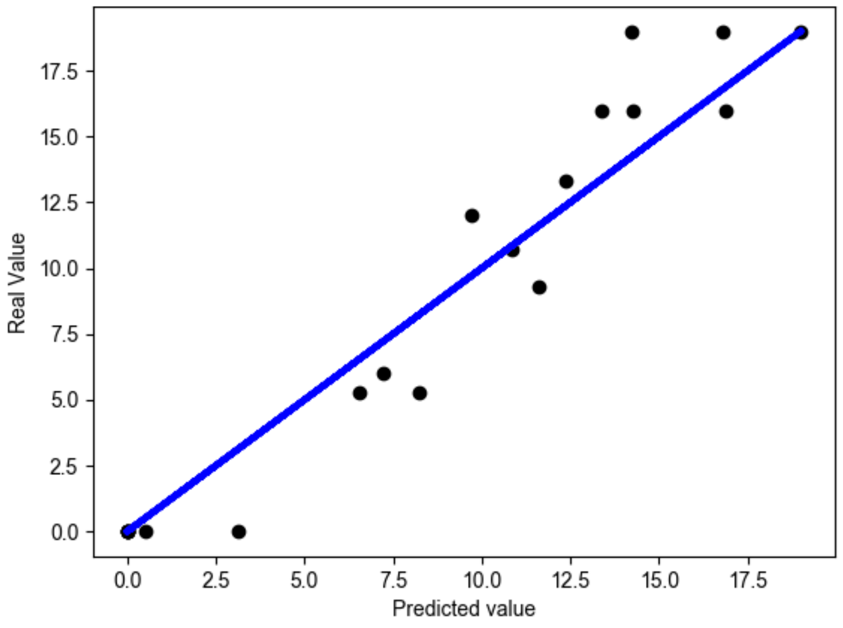


Figure 84: Intuitively representation of the efficiency of the proposed model that targets the Severity of the most sensitive core of X-Gene 2.

4th case: Predict severity of the most robust core

Finally, Figure 85 illustrates the results of the accuracy of the models that predict the Severity values of the most robust core of the chip. All the models apart from the case of polynomial transformation with *RFE* feature selection outperform again the baseline prediction (the difference is about 5.2 Severity units). The best accuracy in terms of RMSE is equal to 2.5 Severity units and it is observed for the linear regression model with *RFE* feature selection using 6 features (this model is presented in Table 33). The R^2 for this model is again very high (equal to 0.91) indicating the high efficiency of the prediction model. Including the error of this model, the potential power savings using the aggressive prediction that targets the Severity of this core are 20.28% instead of using the nominal voltage value, while these gains correspond to 15.75% more power savings compared to the case of using a more conservative scheme that predicts the V_{min}.

Finally, Figure 86 illustrates intuitively the efficiency of our proposed model that targets the Severity of the most robust core of X-Gene 2 chip. The blue line in this figure represents the ideal efficiency where the real and the predicted values of the test dataset are equal. The closer the black dots are to the blue line, the more efficient the prediction model is.

Table 33: Severity prediction model of the most robust core of X-Gene 2.

Symbol definition of microarchitectural counters used by the model	Prediction Model (with 6 features)
Voltage (Volt)	18.94 + (47.80 * *Volt*) – (20.56 * *excpt_taken)* + (7.09 * *L1I_miss*) – (3.92 * *cond_br)* – (29.50 * *L1D_tlb_write)* – (22.11 * *dec_stalls)*
Exceptions taken (excpt_taken)	
L1 instruction cache miss (L1I_miss)	
Conditional branches (*cond_br*)	
L1 data TLB write (L1D_tlb_write)	
Decode stalls (dec_stalls)	

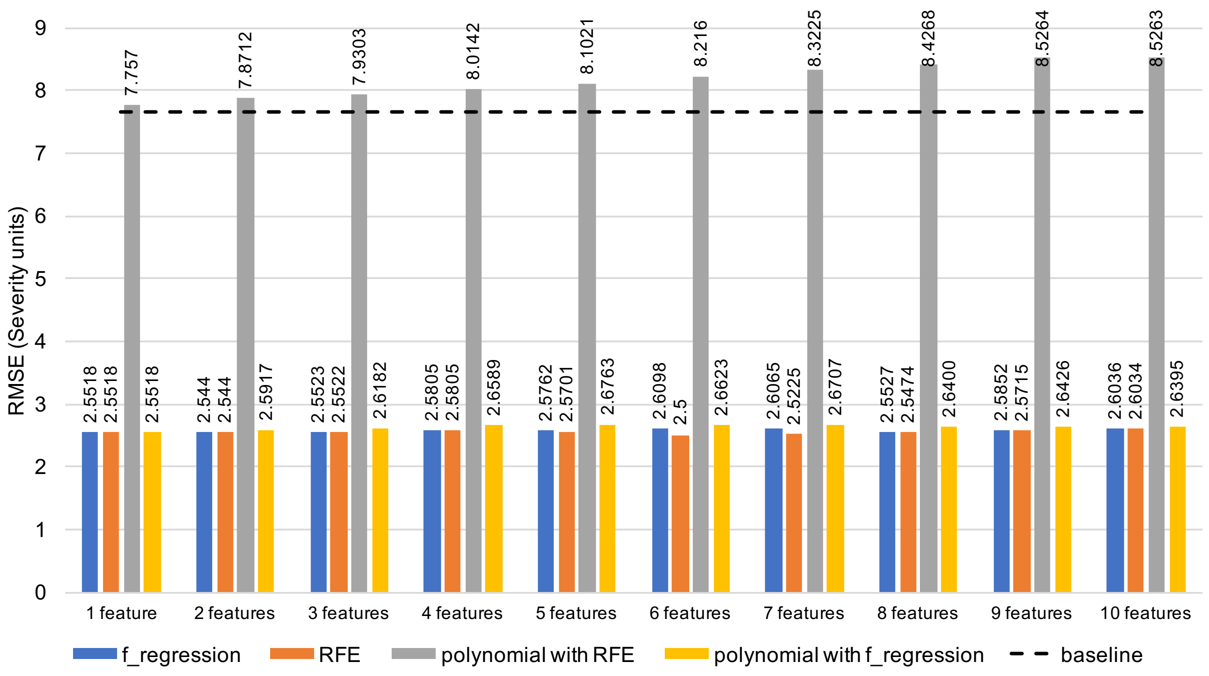


Figure 85: Accuracy of predicting the *Severity* of the most robust core.

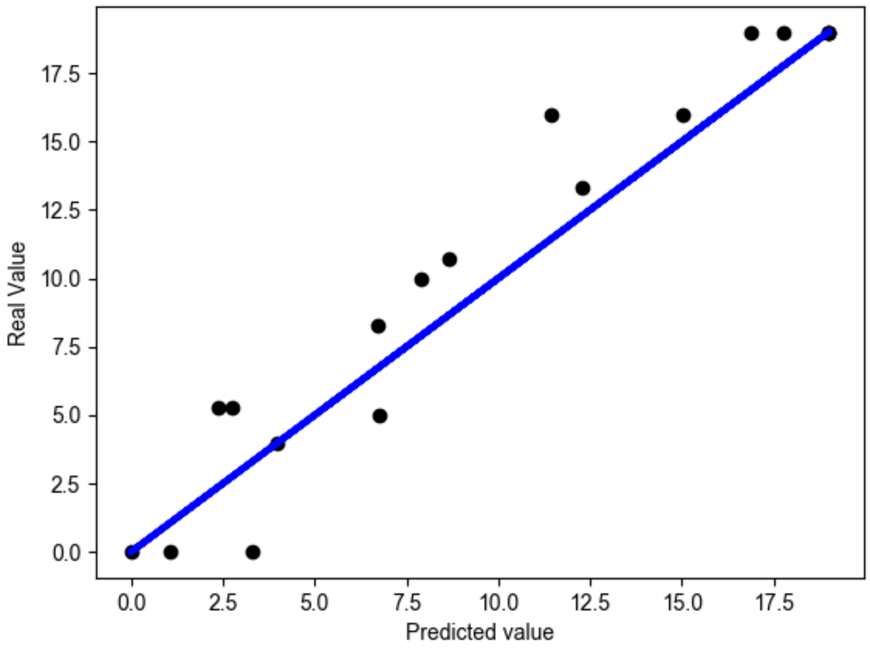


Figure 86: Intuitively representation of the efficiency of the proposed model that targets the Severity of the most robust core of X-Gene 2.

4.2.4 Related work

The goal for improving microprocessors' energy efficiency by reducing their supply voltage is a main concern of many recent scientific studies. For example, [143], [144] and [145] propose methods to maximize voltage droops in single core and multicore chips in order to investigate their worst-case behavior due to the generated voltage noise effects. Studies [133] and [134] focus on the prediction of critical parts of benchmarks, in which large voltage noise glitches are likely to occur, leading to system malfunctions. In the same context, several studies either in the hardware or in the software level were presented to mitigate the effects of voltage noise ([128], [146],

[147], [148] and [149]) or to recover from them after their occurrence [150]. The authors in [151] presented a core that was designed for voltage scalability that can work in high-performance mode at nominal V_{dd} and in a very energy-efficient mode at low V_{dd}.

Apart from these studies that are mainly concentrated on the core and the voltage droops, [12] and [135] focus on the observation of the errors manifested on caches of a commercial Intel Itanium processor during the execution of benchmarks off-nominal voltage values. In [152], we presented an experimental study that aims to identify the voltage margins in two different commercial x86-64 microprocessors; an ultra-low power and a high-end microprocessor. In the same work, we presented the guardbands of these microprocessors, and combined them to the power and temperature savings, when they operate beyond nominal voltage conditions. Moreover, the authors of [20], [153] and [154] propose several microarchitectural approaches to ensure the correct operation of caches in ultra-low voltage conditions. Finally, in [166] we developed some micro-viruses programs that target the caches and the pipeline for fast characterization of the operation voltage margins of the X-Gene 2 chip.

The characterization studies of commercial chips in off-nominal voltage conditions are limited ([12], [135], [155], [156], [157] and [158]) strengthening the need of the existence of our work in [35] and [167] that targets the APM X-Gene 2 micro-server. Similar characterization effort for emerging ARM-based enterprise server systems is sparse. Authors in [159], [160], [161] and [162] from ARM Research developed an electrical simulation framework for power-delivery analysis and used an on-chip voltage monitoring circuit to characterize supply voltage droops in a dual-core ARM Cortex-A57 cluster operating at 1.2 GHz.

Regression analysis has been used in many performance and power studies ([163], [164] and [165]), as well as in reliability estimation concerning soft errors ([105] and [139]), but in [35] and [167] was the first time that was used to predict the safe voltage operation margins of the individual cores of an ARM-v8 multicore CPU.

4.3 Findings Summary

In this chapter, we presented two techniques to boost post-silicon reliability analysis (see Figure 8). Here, we summarize these two techniques that were presented in detail in Section 4.1 and Section 4.2:

- In [33], we have proposed effective parallelization to accelerate online error detection of permanent faults for many-core architectures. We carried out a set of experiments that demonstrate the efficiency of the proposed methodology on the 48-core Intel's SCC architecture. Our methodology exploits the high-speed message passing on-chip network commonly used in such many-core architectures to accelerate the parallel execution of the test preparation phase of memory-intensive test programs. Also, the parallel execution of memory-intensive and CPU-intensive test programs is proposed that further reduces the overall test execution time showing an up to 47.6X speedup compared to a serial test program execution approach.
- In [35] and [167], we presented a comprehensive statistical analysis using linear regression with different feature selection methods to predict the safe voltage operation margins as well as the behavior (in terms of *severity*) of the cores of the enterprise X-Gene 2 micro-server running all the benchmarks from SPEC CPU2006 suite with different input datasets. Our analysis revealed that the potential power savings targeting the V_{min} can be up to 17.52% compared to the case of using the nominal voltage value or up to 20.28% in the case of using a more aggressive regression scheme that targets the Severity instead of the V_{min}.

5. CONCLUSION AND FUTURE WORK

The evolution in semiconductors manufacturing technology gives designers the opportunity to integrate more transistors in the same chip. This leads to increase of performance of the modern microprocessors as more aggressive architectures are implemented. However, this scaling in performance is also accompanied by increase in the vulnerability of microprocessors due to: (a) the strict deadlines that are required to diminish the Time-to-Market (TTM), (b) the modern device integration techniques, and (c) the increased design complexity. Specifically, the microprocessors face serious reliability issues during their entire life-cycle due to: (i) the errors that come from *transient faults* that are caused by cosmic rays, alpha particles and electromagnetic interference, (ii) aging that leads to operational errors that appear at regular time intervals (*intermittent errors*) or exist indefinitely (*permanent errors*), and (iii) manufacturing defects that can either be manifested as permanent errors or lead to timing errors when the chips operate in off-nominal voltage and frequency conditions.

In this thesis, we proposed several techniques to boost reliability of modern microprocessors. The proposed techniques can be implemented either on the early design phases (Pre-Silicon Reliability Analysis phase; see Figure 8) or during manufacturing or even after the chips release to the market (Post-Silicon Reliability Analysis phase; see Figure 8). Here, we summarize the contributions that were described in detail in this thesis:

- **Pre-Silicon Reliability Analysis**: Statistical fault injection on microarchitectural structures modeled in performance simulators is a state-of-the-art method to accurately measure the reliability, but suffers from low simulation throughput. In this thesis, we firstly presented a novel fully-automated versatile architecture-level fault injection framework (called MaFIN) for accurate characterization of a wide range of hardware components of an x86-64 microarchitecture with respect to various fault models (transient, intermittent, permanent faults). Next, using the same tool and focusing on transient faults, we presented several reliability and performance related studies that can help design decision in the early design phases.

  Finally, in this thesis we proposed two methodologies to accelerate the statistical fault injection campaigns. In the first one, we accelerate the fault injection campaigns after the actual injection of the faults in the hardware structures. In the second, we further accelerate the microarchitecture level fault injection campaigns by proposing MeRLiN that is based on the pruning of the initial fault list by grouping the faults in equivalent classes according to the instruction that finally accesses the faulty entry.

- **Post-Silicon Reliability Analysis**: The contributions of this thesis in this phase of the life-cycle cover two important research fields. Firstly, using the 48-core Intel's SCC architecture, we proposed a technique to accelerate online error detection of permanent faults for many-core architectures by exploiting their high-speed message passing on-chip network. Secondly, we proposed a detailed statistical analysis methodology to accurately predict in the system level the safe voltage operation margins of the ARMv8 cores of the X-Gene 2 chip when it operates in scaled voltage conditions.

We hope that the techniques presented in this thesis will advance the research in the field of dependable and energy efficient computing. In Table 34, we indicatively present some future directions of our research in the two phases of processors' reliability life-cycle, on which we mainly focused in this thesis.

Table 34: Future work.

Phase of reliability life-cycle	Future work
Pre-Silicon Reliability Analysis	• Extension of MaFIN tool with protection mechanisms in different hardware structures • Implementation of MeRLiN methodology in GPUs and other multi-core architectures • Use of MeRLiN's methodology outcomes to analyze software reliability
Post-Silicon Reliability Analysis	• Implementation of the proposed methodology of [33] on different many-core architectures with different network topologies (for instance in Xeon Phi) • Implementation of the linear regression analysis of [35] and [167] on different multicore architectures (for instance the X-Gene 3 [36]) and on different ISAs. • Statistical analysis to predict temperature and power of the X-Gene 2.

ACRONYMS

TTM	Time-to-Market
IoT	Internet of Things
ILP	Instruction-Level Parallelism
TLP	Thread-Level Parallelism
DLP	Data-Level Parallelism
ISA	Instruction Set Architecture
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
RTL	Register Transfer Level
SBST	Software-based Self-testing
ECC	Error Correction Code
SEU	Single Event Upset
SCC	Intel’s Single-chip Cloud Computer
SER	Soft Error Rate
SOI	Silicon on Insulator
TTF	Time to Failure
MTTF	Mean Time to Failure
FIT	Failure in Time
AVF	Architectural Vulnerability Factor
HVF	Hardware Vulnerability Factor
PVF	Program Vulnerability Factor
SDC	Silent Data Corruption
DUE	Detected Unrecoverable Error
FP	Floating Point
OoO	Out-of-order
RAS	Return Address Stack
LSQ	Load Store Queue
ALU	Arithmetic Logic Unit
AGU	Address Generation Unit
IPC	Instructions per Cycle
WB	Write Back
WT	Write Through
SIMD	Single Instruction, Multiple Data

BTB	Branch Target Buffer
ROB	Reorder Buffer
uop	micro-operation
RIP	instruction pointer
uPC	micro Program Counter
TDD	Transitively Dynamically Dead
RF	Physical Register File
SQ	Store Queue
IQ	Issue Queue
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
GPU	Graphics Processing Unit
SoC	Systems-on-Chip
NPU	Network Processing Unit
ATPG	Automatic Test Pattern Generation
LFSR	Linear-Feedback Shift Register
DRAM	Dynamic Random Access Memory
MPB	Message Passing Buffer
SMP	Symmetric shared-memory Multiprocessors
CMP	Chip-Multiprocessors
DVFS	Dynamic Voltage and Frequency Scaling
PMpro	Power Management processor
SLIMpro	Scalable Lightweight Intelligent Management processor
ACPI	Advanced Configuration Power Interface
PMD	Processor Module
PCP	Processor Complex
TDP	Thermal Design Power
RMSE	Root Mean Square Error
OLS	Ordinary Least Squares

ANNEX I

Event-id	Description
0x000	Instruction architecturally executed, condition code check pass, software increment
0x001	L1 Instruction cache refill
0x002	L1 instruction TLB refill
0x003	L1 data cache refill
0x004	L1 data cache access
0x005	L1 data TLB refill
0x008	Instruction architecturally executed
0x009	Exception taken
0x00A	Instruction architecturally executed (condition check pass) -Exception return
0x00B	Instruction architecturally executed (condition check pass) - Write to CONTEXTIDR
0x010	Mispredicted or not predicted branch speculatively executed
0x011	Cycle
0x012	Predictable branch speculatively executed
0x013	Data memory access
0x014	L1 instruction cache access
0x016	L2 data cache access
0x017	L2 data cache refill
0x018	L2 data cache write-back
0x019	Bus access
0x01A	Local Memory Error
0x01B	Operation speculatively executed
0x01C	Instruction architecturally executed (condition check pass) - Write to translation table base
0x01E	Counter chain
0x040	L1 data cache access - Read
0x041	L1 data cache access - Write
0x042	L1 data cache refill - Read
0x048	L1 data cache invalidate
0x04C	L1 data TLB refill - Read
0x04D	L1 data TLB refill - Write
0x050	L2 data cache access - Read
0x051	L2 data cache access - Write
0x052	L2 data cache refill - Read
0x053	L2 data cache refill - Write
0x056	L2 data cache write-back - victim
0x057	L2 data cache write-back - Cleaning and coherency
0x058	L2 data cache invalidate
0x060	Bus access - Read
0x061	Bus access - Write
0x062	Bus access - Normal, cacheable, sharable
0x063	Bus access - Not normal, cacheable, sharable
0x064	Bus access - Normal
0x065	Bus access - Peripheral
0x066	Data memory access - Read

0x067	Data memory access - write
0x068	Unaligned access - Read
0x069	Unaligned access - Write
0x06A	Unaligned access
0x06C	Exclusive operation speculatively executed - Load exclusive
0x06D	Exclusive operation speculative executed - Store exclusive pass
0x06E	Exclusive operation speculative executed - Store exclusive fail
0x06F	Exclusive operation speculatively executed - Store exclusive
0x070	Operation speculatively executed - Load
0x071	Operation speculatively executed - Store
0x072	Operation speculatively executed - Load or store
0x073	Operation speculatively executed - Integer data processing
0x074	Operation speculatively executed - Advanced SIMD
0x075	Operation speculatively executed - FP
0x076	Operation speculatively executed - Software change of PC
0x078	Branch speculative executed - Immediate branch
0x079	Branch speculative executed - Procedure return
0x07A	Branch speculative executed - Indirect branch
0x07C	Barrier speculatively executed - ISB
0x07D	Barrier speculatively executed - DSB
0x07E	Barrier speculatively executed - DMB
0x081	Exception taken, other synchronous
0x082	Exception taken, Supervisor Call
0x083	Exception taken, Instruction Abort
0x084	Exception taken, Data Abort or SError
0x086	Exception taken, IRQ
0x087	Exception taken, FIQ
0x08A	Exception taken, Hypervisor Call
0x08B	Exception taken, Instruction Abort not taken locally
0x08C	Exception taken, Data Abort or SError not taken locally
0x08D	Exception taken, other traps not taken locally
0x08E	Exception taken, IRQ not taken locally
0x08F	Exception taken, FIQ not taken locally
0x090	Release consistency instruction speculatively executed - Load Acquire
0x091	Release consistency instruction speculatively executed - Store Release
0x100	Operation speculatively executed - NOP
0x101	FSU clocking gated off cycle
0x102	BTB misprediction
0x103	ITB miss
0x104	DTB miss
0x105	L1 data cache late miss
0x106	L1 data cache prefetch request
0x107	L2 data prefetch request
0x108	Decode starved for instruction cycle
0x109	Op dispatch stalled cycle
0x10A	IXA Op non-issue
0x10B	IXB Op non-issue
0x10C	BX Op non-issue
0x10D	LX Op non-issue
0x10E	SX Op non-issue

0x10F	FX Op non-issue
0x110	Wait state cycle
0x111	L1 stage-2 TLB refill
0x112	Page Walk Cache level-0 stage-1 hit
0x113	Page Walk Cache level-1 stage-1 hit
0x114	Page Walk Cache level-2 stage-1 hit
0x115	Page Walk Cache level-1 stage-2 hit
0x116	Page Walk Cache level-2 stage-2 hit

REFERENCES

[1] G.Moore, “Cramming more components onto integrated circuits”, In Electronics, April, 1965.

[2] 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.

[3] Datasheet Intel 4004; http://datasheets.chipdb.org/Intel/MCS-4/datashts/intel-4004.pdf [Accessed 07/11/2017].

[4] AMD Enterprise CPU Roadmap 2015-2019 Leaked – Features 14nm Naples With 32 Cores and 7nm Starship with 48 Cores; https://wccftech.com/amd-cpu-roadmap-leak-7-nm-starship-14nm-naples-snowy-owl-zen-core/ [Accessed 07/11/2017].

[5] Intel presents technology and manufacturing day; https://newsroom.intel.com/news/intel-presents-technology-manufacturing-day-live-video-updates/ [Accessed 07/11/2017].

[6] J.Hennessy, D.Patterson, Computer Architecture: A Quantitative Approach (5 Edition), Elsevier, 2012.

[7] E.Normand, “Single Event Upset at Ground Level,” IEEE Trans. on Nuclear Science, Vol. 43, No. 6, Dec 1996.

[8] R.C.Baumann, “Sun Microsystems found cosmic ray strikes on L2 cache with defective error protection caused Sun’s flagship servers to suddenly and mysteriously crash”, IRPS Tutorial on SER, 2000.

[9] J.F.Zielger, H.Puchner, “SER-History, Trends, and Challenges”, Cypress, 2004.

[10] D.Lorenz, G.Georgakos, U.Schlichtmann, “Aging Analysis of Circuit Timing Considering NBTI and HCI”, IEEE International On-Line Testing Symposium (IOLTS), 2009.

[11] D.Gizopoulos, et al., “Architectures for online error detection and recovery in multicore processors”, ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011.

[12] A.Bacha, R.Teodorescu, “Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II Processors”, ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013.

[13] M.T.Yourst, “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator”, IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2007.

[14] A.Patel, F.Afram, S.Chen, K.Ghose, “MARSSx86: A Full System Simulator for x86 CPUs”, ACM/IEEE Design Automation Conference (DAC), 2011.

[15] N.Binkert, et al., “The Gem5 simulator”, ACM SIGARCH Computer Architecture News, vol. 39, no. 2, May 2011.

[16] A.Avizienis, J.-C.Laprie, B.Randell, C.Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, January-March 2004.

[17] D.Gizopoulos, S.Mukherjee, “Guest Editors’ Introduction: Special Section on Dependable Computer Architecture”, IEEE Transactions on Computers (TC), vol. 60, no. 1, January 2011.

[18] Y.Luo, et al., “Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory”, IEEE/IFIP International Conference on Dependable systems and Networks (DSN), 2014.

[19] L.T.Wang, C.Stroud, N.Touba, “System-on-Chip test architectures: Nanometer design for testability”, Elsevier, 2007.

[20] Z.Chishti, A.R.Alameldeen, C.Wilkerson, W.Wu, S.-L.Lu, “Improving cache lifetime reliability at ultra-low voltages”, ACM/IEEE International Symposium on Microarchitecture (MICRO), 2009.

[21] S.Raasch, A.Biswas, J.Stephan, P.Racunas, J.Emer, “A fast and accurate analytical technique to compute the AVF of sequential bits in a processor”, ACM/IEEE International Symposium on Microarchitecture (MICRO), 2015.

[22] B.Bentley, “Validating the Intel® Pentium® 4 Microprocessor”, ACM/IEEE Design Automation Conference (DAC), 2011.

[23] K.Constantinides, O.Mutlu, T.Austin, “Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation”, ACM/IEEE International Symposium on Microarchitecture (MICRO), 2008.

[24] Y.C.Hsu, F.Tsai, W.Jong, “Visibility enhancement for silicon debug”, ACM/IEEE Design Automation Conference (DAC), 2006.

[25] J.M.Rabaey, “Digital integrated circuits: a design perspective”, Prentice-Hall, Inc., 1996.

[26] D.Gizopoulos, “Advanced electronic testing: challenges and methodologies”, Springer, 2006.

[27] N.Foutris, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Versatile architecture-level fault injection framework for reliability evaluation: a first report”, IEEE International On-Line Testing Symposium (IOLTS), 2014.

[28] S.Tselonis, M.Kaliorakis, N.Foutris, G.Papadimitriou, D.Gizopoulos, “Microprocessor reliability-performance tradeoffs assessment at the microarchitecture level”, IEEE VLSI Test Symposium (VTS), 2016.
[29] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, “Differential fault injection on microarchitectural simulators”, IEEE International Symposium on Workload Characterization (IISWC), 2015.
[30] A.Chatzidimitriou, D.Gizopoulos, “Anatomy of microarchitecture-level reliability assessment: throughput and accuracy”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016.
[31] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, D.Gizopoulos, “Accelerated microarchitectural fault injection-based reliability assessment”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), 2015.
[32] M.Kaliorakis, D.Gizopoulos, R.Canal, A.Gonzalez, “MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment”, ACM/IEEE International Symposium on Computer Architecture (ISCA), 2017.
[33] M.Kaliorakis, M.Psarakis, N.Foutris, D.Gizopoulos, “Accelerated online error detection in many-core microprocessor architectures”, IEEE VLSI Test Symposium (VTS), 2014.
[34] SCC Programmer's Guide, rev.1.0, Jan., 2012. [Accessed 13/11/2017].
[35] G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, D.Gizopoulos, P.Lawthers, S.Das, “Harnessing voltage margins for energy efficiency in multicore CPUs”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017.
[36] X-Gene^{TM} Word’s First ARMv8 64-bit Server on Chip® Solution; https://www.apm.com/products/data-center/x-gene-family/x-gene/ [Accessed 13/11/2017].
[37] R.C.Baumann, “Soft errors in advanced computer systems”, IEEE Design & Test of Comp., vol. 22, no. 3, pp. 258-266, May/June 2005.
[38] C.Constantinescu, “Trends and challenges in VLSI circuit reliability”, IEEE Micro, vol. 23, pp. 14-19, July 2003.
[39] L.Huang, Q.Xu, “AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs”, ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010.
[40] S.Mukherjee, “Architecture design for soft errors”, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2008.
[41] M.Riera, R.Canal, J.Abella, A.Gonzalez, “A detailed methodology to compute Soft Error Rates in advanced technologies”, ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016.
[42] V.Sridharan, D.R.Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2009.
[43] V.Sridharan, D.R.Kaeli, “Using hardware vulnerability factors to enhance AVF analysis”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2010.
[44] J.Goodenough, R.Aitken, “Post-silicon is too late avoiding the $50 million paperweight starts with validated designs”, ACM/IEEE Design Automation Conference (DAC), 2010.
[45] H.Cho, S.Mirkhani, C.-Y.Cher, J.A.Abraham, S.Mitra, “Quantitative evaluation of soft error injection techniques for robust system design”, ACM/IEEE Design Automation Conference (DAC), 2013.
[46] M.Maniatakos, N.Karimi, C.Tirumurti, A.Jas, Y.Makris, “Instruction-level impact analysis of low-level faults in a modern microprocessor controller”, IEEE Transactions on Computers, vol. 60, no. 9, pp.1260-1273, September 2011.
[47] N.J.Wang, A.Mahersi, S.J.Patel, “Examining ACE analysis reliability estimates using fault-injection”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2007.
[48] G.Yalcin, O.S.Unsal, A.Cristal, M.Valero, “FIMSIM: A fault injection infrastructure for microarchitectural simulators”, IEEE International Conference on Computer Design (ICCD), 2011.
[49] N.Foutris, D.Gizopoulos, J.Kalamatianos, V.Sridharan, “Assessing the impact of hard faults in performance components of modern microprocessors” IEEE International Conference on Computer Design (ICCD), 2013.
[50] S.S.Mukherjee, C.Weaver, J.Emer, S.K.Reinhardt, T.Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessors”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2004.
[51] A.Nair, S.Eyerman, L.Eeckhout, L.K.John, “A first-order mechanistic model for architectural vulnerability factor”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2012.
[52] A.Biswas, P.Racunas, R.Cheveresan, J.Emer, S.S.Mukherjee, R.Rangan, “Computing architectural vulnerability factors for address-based structures”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2005.

[53] H.Asadi, V.Sridharan, M.Tahoori, D.Kaeli, “Balancing performance and reliability in the memory hierarchy”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2005.

[54] X.Li, S.V.Adve, P.Bose, J.A.Rivers, “SoftArch: An architecture-level tool for modeling and analyzing soft errors”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2005.

[55] J.Suh, M.Annavaram, M.Dubois, “MACAU: A markov model for reliability evaluations of caches under single-bit and multi-bit upsets”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2012.

[56] J.Suh, M.Manoochehri, M.Annavaram, M.Dubois, “Soft error benchmarking of L2 caches with PARMA”, ACM SIGMETRICS, 2011.

[57] S.Feng, S.Gupta, A.Ansari, S.Mahlke, “Shoestring: probabilistic soft error reliability on the cheap”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2010.

[58] N.J.George, C.R.Elks, B.W.Johnson, J.Lach, “Transient fault models and AVF estimation revisited”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2010.

[59] X.Li, S.V.Adve, P.Bose, J.A.Rivers, “Architecture-level soft error analysis: Examining the limits of common assumptions”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2007.

[60] A.A.Nair, L.K.John, L.Eeckhout, “AVF Stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010.

[61] A.Biswas, P.Racunas, J.Emer, S.S.Mukherjee, “Computing accurate AVFs using ACE analysis on performance models: a rebuttal”, IEEE Computer Architecture Letters, vol.7, no. 1, January-June 2008.

[62] R.Leveugle, A.Calvez, P.Maistri, P.Vanhauwaert, “Statistical fault injection: Quantified error and confidence”, ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2009.

[63] M-T.Chang, P.Rosenfeld, S-L.Lu, B.Jacob, “Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2013.

[64] A.Mayberry, M.Laquidara, C.Weeds, “Characterizing the microarchitectural side effects of operating system calls”, IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2013.

[65] J.Stevens, P.Tschirhart, M-T.Chang, I.Bhati, P.Enns, J.Greensky, Z.Cristi, S-L.Lu, B.Jacob, “An integrated simulation infrastructure for the entire memory hierarchy: cache, dram, nonvolatile memory, and disk”, Intel Technology Journal, vol.17, no 1, 2013.

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

[67] N.Foutris, D.Gizopoulos, X.Vera, A.Gonzalez, “Deconfigurable microprocessor architectures for silicon debug acceleration”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2013.

[68] F.Bellard, “QEMU, a Fast and Portable Dynamic Translator”, USENIX Annual Technical Conference, 2005.

[69] A.Gutierrez et al., “Sources of error in full-system simulation”, IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2014.

[70] T.F.Wenisch et al., “SimFlex: Statistical sampling of computer system simulation”, IEEE Micro, vol. 26, no. 4, pp. 18-31, 2006.

[71] M.K.Martin et al., “Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset”, ACM SIGARCH Computer Arch. News, vol. 33, no. 4, Nov. 2005.

[72] Imperas. OVPsim, http://ovpworld.org . [Accessed 13/11/2017].

[73] P.S.Magnusson et al., “Simics: a full system simulation platform”, IEEE Computer, vol. 35, no. 2, pp. 50-58, Feb. 2002.

[74] M.R.Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite”, International Workshop on Workload Characterization (IWWC), 2001.

[75] Standard Performance Evaluation Corporation, https://www.spec.org [Accessed 13/11/2017].

[76] Z.Zhao, D.Lee, A.Gerstlauer, L.K.John, “Host-compiled reliability modeling for fast estimation of architectural vulnerabilities”, Workshop on Silicon Errors in Logic – System Effects (SELSE), 2015.

[77] D.S.Khudia, S.Mahlke, “Harnessing soft computations for low budget fault tolerance”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.

[78] M.-L.Li, P.Ramachandran, U.R.Karpuzcu, S.K.S.Hari, S.V.Adve, “Accurate microarchitecture-level fault modeling for studying hardware faults”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2009.

[79] X.Li, S.V.Adve, P.Bose, J.A.Rivers, “Architecture-level soft error analysis: Examining the limits of common assumptions”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2007.

[80] A.Phansalkar, A.Joshi, L.K.John, “Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2007.

[81] A.Sodani, G.S.Sohi, “Dynamic instruction reuse”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 1997.

[82] A.Sodani, G.S.Sohi, “An empirical analysis of instruction repetition”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1998.

[83] S.Balakrishnan, G.S.Sohi, “Exploiting value locality in physical register files”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2003.

[84] M.-L.Li, P.Ramachandran, S.K.Sahoo, S.V.Adve, V.S.Adve, Yuanyuan Zhou, “Understanding the propagation of hard errors to software and implications for resilient system design”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008.

[85] T.Sherwood, E.Perelman, G.Hamerly, B.Calder, “Automatically characterizing large scale program behavior”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.

[86] J.Suh, M.Annavaram, M.Dubois, “PHYS: Profiled-HYbrid Sampling for soft error reliability benchmarking”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2013.

[87] S.K.S.Hari, S.V.Adve, H.Naemi, P.Ramachandran, “Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[88] H.Schirmeier, C.Borchert, O.Spinczyk, “Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2015.

[89] G.Li, Q.Lu, K.Pattabiraman, “Fine-grained characterization of faults causing long latency crashes in programs”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2015.

[90] A.Vallero, S.Tselonis, N.Foutris, M.Kaliorakis, M.Kooli, A.Savino, G.Politano, A.Bosio, G.Di Natale, D.Gizopoulos, S.Di Carlo, “Cross-layer reliability evaluation, moving from the hardware architecture to the system level: a CLERECO EU Project overview”, Journal of Microprocessors and Microsystems, June 2015.

[91] A.Vallero, A.Savino, S.Tselonis, N.Fourtis, M.Kaliorakis, G.Politano, D.Gizopoulos, S.Di Carlo, “A bayesian model for system level reliability estimation”, IEEE European Test Symposium (ETS), 2015.

[92] A.Vallero, A.Savino, S.Tselonis, N.Fourtis, M.Kaliorakis, G.Politano, D.Gizopoulos, S.Di Carlo, “Bayesian network early reliability evaluation analysis for both permanent and transient faults”, IEEE International On-Line Testing Symposium (IOLTS), 2015.

[93] A.Vallero, A.Savino, G.Politano, S.Di Carlo, A.Chatzidimitriou, S.Tselonis, M.Kaliorakis, D.Gizopoulos, M.R.Villanueva, R.Canal, A.Gonzalez, M.Kooli, A.Bosio, G.Di Natale, “Cross-Layer system reliability assessment framework for hardware faults”, IEEE International Test Conference (ITC), 2016.

[94] A.Chatzidimitriou, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Performance-aware reliability assessment of heterogeneous chips”, IEEE VLSI Test Symposium (VTS), 2017.

[95] A.Chatzidimitriou, M.Kaliorakis, D.Gizopoulos, M.Pipponzi, R.Mariani, S.Di Carlo, “RT Level vs. microarchitecture level reliability assessment: case study on ARM Cortex-A9 CPU”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2017. (industrial track paper)

[96] N.L.Binkert et al., “The M5 simulator: modeling networked systems, IEEE Micro, vol. 26, no. 4, pp. 52-60, July/August 2006.

[97] K.Parasyris, G.Tziantzoulis, C.Antonopoulos, N.Bellas, “GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014.

[98] P.Racunas, K.Constantinides, S.Manne, S.S.Mukherjee, “Perturbation-based fault screening”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2007.

[99] G.Saggese, N.J.Wang, Z.Kalbarczyk, S.J.Patel, R.Iyer, "An experimental study of soft errors in microprocessors" IEEE Micro, vol. 25, no. 6, pp. 30-39, Nov-Dec 2005.

[100] R.Balasubramanian, K.Sankaralingam, “Understanding the impact of gate-level physical reliability effects on whole program execution”, IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014.

[101] A.Pellegrini, K.Constantinides, D.Zhang, S.Sudhakar, V.Bertacco, T.Austin, “CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework”, IEEE International Conference on Computer Design (ICCD), 2008.
[102] N.J.Wang, J.Quek, T.M.Rafacz, S.J.Patel, “Characterizing the effects of transient faults on a high-performance processor pipeline”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2004.
[103] S.Kim, K.Somani, “Soft error sensitivity characterization for microprocessor dependability enhancement strategy”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2002.
[104] P.Ramachandrant, P.Kudvatt, J.Kellingtont, J.Schumannt, P.Sanda, “Statistical fault injection”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2008.
[105] A.Biswas, N.Soundararajan, S.S.Mukherjee, S.Gurumurthi, “Quantized AVF: a means of capturing vulnerability variations over small windows of time”, International Workshop on Silicon Errors in Logic-System Effects (SELSE), 2009.
[106] P.Montesinos, W.Liu, J.Torrellas, “Using register lifetime predictions to protect register files against soft errors”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2007.
[107] X.Xu, M.-L.Li, “Understanding soft error propagation using efficient vulnerability-driven fault injection”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2012.
[108] V.Reddy, E.Rotenberg, “Inherent Time Redundancy (ITR): Using program repetition for low-overhead fault tolerance”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2007.
[109] M.A.Gomaa, T.N.Vijaykumar, “Opportunistic transient-fault detection”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2005.
[110] J.Howard, et al., "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS", IEEE International Solid-State Circuits Conference (ISSCC), 2010.
[111] J.Nickolls, W.J.Dally, "The GPU computing era", IEEE Micro, Volume 30, Issue 2, pp. 56-69 ,March-April 2010.
[112] T.Chen, R.Raghavan, J.Dale, E.Iwata, "Cell broadband engine architecture and its first implementation- A performance view," IBM Journal of Research and Development , vol.51, no.5, pp.559,572, Sept. 2007.
[113] “The Cisco QuantumFlow Processor: Cisco's next generation network processor”, Cisco Systems Inc., 2008.
[114] “OCTEON Plus CN58XX 4 to 16-Core MIPS64-Based SoCs”, Cavium Networks, Mountain View, CA, 2008.
[115] M.Adiletta, M.Rosenbluth, D.Bernstein, "The next generation of intel ixp network processors", Intel Technology Journal, 06(03), Aug. 2002.
[116] S.Mitra, E.J.McCluskey, "Which concurrent error detection scheme to choose?", IEEE International Test Conference (ITC), 2000.
[117] M.Psarakis, D.Gizopoulos, E.Sanchez, M.S.Reorda, "Microprocessor software-based self-testing", IEEE Design & Test of Computers, vol.27, no.3, pp.4,19, May-June 2010.
[118] https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html [Accessed 01/12/2017].
[119] P.Gschwandtner, T.Fahringer, R.Prodan, "Performance analysis and benchmarking of the Intel SCC", IEEE International Conference on Cluster Computing (CLUSTER), 2011.
[120] M.Kaliorakis, N.Foutris, D.Gizopoulos, M.Psarakis, "Online error detection in multiprocessor chips: A test scheduling study", IEEE International On-Line Testing Symposium (IOLTS), 2013.
[121] A.Paschalis, D.Gizopoulos, “Effective software-based self-test strategies for on-line periodic testing of embedded processors”, IEEE Transactions on Computer-Aided Design, vol. 24, no. 1, pp. 88–99, 2005.
[122] A.Apostolakis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors", IEEE Transactions on Computers, vol. 58, no. 12, pp. 1682-1694, 2009.
[123] N.Foutris, M.Psarakis, D.Gizopoulos, A.Apostolakis, X.Vera, A.Gonzalez, "Mt-sbst: self-test optimization in multithreaded multicore architectures", IEEE International Test Conference (ITC), 2010.
[124] M.A.Skitsas, C.A.Nicopoulos, M.K.Michael, "DaemonGuard: O/S- assisted selective software-based self-testing for multi-core systems," IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), 2013.
[125] F.Salehuddin, I.Ahmad, F.A.Hamid, A.Zaharim, A.Maheran, A.Hamid, P.S.Menon, H.A.Elgomati, B.Y.Majlis, “Optimization of process parameter variation in 45nm p-channel MOSFET using L18 Orthogonal Array”, IEEE International Conference on Semiconductor Electronic (ICSE), 2012.

[126] W.Schemmert, G. Zimmer, “Threshold-voltage sensitivity of ion-implanted MOS transistors due to process variations”, Electronics Letters, vol. 10, no. 9, pp. 151–152, May 1974.
[127] N.James, P.Restle, J.Friedrich, B.Huott, B.McCredie, “Comparison of split-versus connected-core supplies in the POWER6 microprocessor”, IEEE International Solid-State Circuits Conference (ISSCC), 2007.
[128] V.J.Reddi, S.Kanev, W.Kim, S.Campanoni, M.D.Smith, G.-Y.Wei, D.Brooks, “Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010.
[129] E.L.Sueur, G.Heiser, “Dynamic voltage and frequency scaling: the laws of diminishing returns”, international conference on Power aware computing and systems (HotPower), 2010.
[130] D.Ernst, N.S.Kim, S.Das, S.Pant, R.Rao, T.Pham, C.Ziesler, D.Blaauw, T.Austin, K.Flautner, T.Mudge, “Razor: A low-power pipeline based on circuit-level timing speculation”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2003.
[131] Y.Zu, C.R.Lefurgy, J.Leng, M.Halpern, M.S.Floyd, V.J.Reddi, “Adaptive guardband scheduling to improve system-level efficiency of the POWER7+”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.
[132] G.Karakonstantis, et al., “An energy-efficient and error-resilient server ecosystem exceeding conservative scaling limits”, IEEE/ACM Design, Automation & Test in Europe Conference (DATE), 2018.
[133] V.J.Reddi, M.S.Gupta, G.H.Holloway, G.-Y.Wei, M.D.Smith, D.M.Brooks, “Voltage emergency prediction: Using signatures to reduce operating margins”, IEEE International Conference on High-Performance Computer Architecture (HPCA), 2009.
[134] M.S.Gupta, V.J.Reddi, G.Holloway, G.-Y.Wai, D.M.Brooks, “An event-guided approach to reducing voltage noise in processors,” IEEE/ACM Design, Automation & Test in Europe Conference (DATE), 2009.
[135] A.Bacha, R.Teodorescu, “Using ECC feedback to guide voltage speculation in low-voltage processors,” IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.
[136] G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, D.Gizopoulos, G.Favor, K.Sankaran, S.Das, “A system-level voltage/frequency scaling characterization framework for multicore CPUs”, IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE), 2017.
[137] The Linux Kernel Documentation (Parent Directory), https://www.kernel.org/doc/Documentation [Accessed 04/12/2017].
[138] R. J. Riedlinger, et all., “A 32nm 3.1 billion transistor 12-wide-issue Itanium® Processor for mission-critical servers,” IEEE International Solid-State Circuits Conference (ISSCC), 2011.
[139] K.R.Walcott, G.Humphreys, S.Gurumurthi, “Dynamic prediction of architectural vulnerability from microarchitectural state”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2007.
[140] F.Pedregosa, et al., “Scikit-learn: Machine learning in Python”, Machine Learning Research, vol. 12, pp. 2825-2830, October 2011.
[141] “Perf: Linux profiling with performance counters”, https://perf.wiki.kernel.org/index.php/Main_Page [Accessed 05/12/2017]
[142] J.R.Lackritz, “Exact p Values for F and t Tests”, The American Statistician, Vol. 38, No. 4, Nov. 1984, pp. 312-314.
[143] M.Ketkar, E.Chiprout, “A microarchitecture-based framework for pre- and post-silicon power delivery analysis”, IEEE/ACM International Symposium on Microarchitecture (MICRO). 2009.
[144] Y.Kim, L.K.John, “Automated di/dt stressmark generation for microprocessor power delivery networks”, IEEE/ACM International Symposium on Low-Power Electronics and Design (ISLPED). 2011.
[145] Y.Kim, L.K.John, S.Pant, S.Manne, M.Schulte, W.L.Bircher, M.S.S.Govindan, “AUDIT: Stress Testing the Automatic Way”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2012.
[146] M.S.Gupta, K.K.Rangan, M.D.Smith, G.-Y.Wei, D.Brooks, “Towards a software approach to mitigate voltage emergencies”, ACM/IEEE International Symposium on Low Power Electronics and Design (ISPLED), 2007.
[147] R.Joseph, D.Brooks, M.Martonosi, “Control techniques to eliminate voltage emergencies in high performance processors”, IEEE International Conference on High-Performance Computer Architecture (HPCA), 2003.
[148] T.N.Miller, R.Thomas, X.Pan, R.Teodorescu, “VRSync: characterizing and eliminating synchronization-induced voltage emergencies in many-core processors”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2012.
[149] M.D.Powell, T.N.Vijaykumar, “Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise”, ACM/IEEE International Symposium on Low Power Electronics and Design (ISPLED), 2003.

[150] M.S.Gupta, K.K.Rangan, M.D.Smith, G.-Y.Wei, D.Brooks, “DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors”, IEEE International Conference on High-Performance Computer Architecture (HPCA), 2008.

[151] B.Gopireddy, C.Song, J.Torrellas, N.S.Kim, A.Agrawal, A.Mishra, “ScalCore: Designing a core for voltage scalability”, IEEE International Conference on High-Performance Computer Architecture (HPCA), 2016.

[152] G.Papadimitriou, M.Kaliorakis, A.Chatzidimitriou, C.Magdalinos, D.Gizopoulos, “Voltage margins identification on commercial x86-64 multicore microprocessors”, IEEE International On-Line Testing Symposium (IOLTS), 2017.

[153] 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”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2008.

[154] H.Duwe, X.Jian, D.Petrisko, R.Kumar, “Rescuing uncorrectable fault patterns in on-chip memories through error pattern transformation”, IEEE/ACM International Symposium on Computer Architecture (ISCA), 2016.

[155] J.Leng, A.Buyuktosunoglu, R.Bertran, P.Bose, V.J.Reddi, “Safe limits on voltage reduction efficiency in GPUs: a direct measurement approach”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[156] C.R.Lefurgy, A.J.Drake, M.S.Floyd, M.S.Allenware, B.Brock, J.A.Tierno, J.B.Carter, “Active management of timing guardband to save energy in POWER7”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.

[157] A.Bacha, R.Teodorescu, “Authenticache: harnessing cache ECC for system authentication”, IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

[158] S.Sundaram, et al., “Adaptive voltage frequency scaling using critical path accumulator implemented in 28nm CPU”, IEEE International Conference on VLSI Design and International Conference on Embedded Systems (VLSID), 2016.

[159] P.N.Whatmough, S.Das, Z.Hadjilambrou, D.M.Bull, “An all-digital power-delivery monitor for analysis of a 28nm dual-core ARM Cortex-A57 cluster”, IEEE International Solid-State Circuits Conference (ISSCC), 2015.

[160] P.N.Whatmough, S.Das, D.M.Bull, “Analysis of adaptive clocking technique for resonant supply voltage noise mitigation”, IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2015.

[161] S.Das, P.Whatmough, D.M.Bull, “Modelling and characterization of the system-level power-delivery network for a dual-core ARM A57 cluster in 28nm CMOS”, IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2015.

[162] P.Whatmough, S.Das, D.M.Bull, “Power Integrity Analysis of a 28 nm Dual-Core ARM Cortex-A57 Cluster Using an All-Digital Power Delivery Monitor”, Journal of Solid-State Circuits (JSSC). vol. 52, no. 6, pp. 1643 – 1654, March 2017.

[163] W.Jia, K.A.Shaw, M.Martonosi, “Stargazer: Automated regression-based GPU design space exploration”, IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2012.

[164] P.J.Joseph, K.Vaswani, M.J.Thazhuthaveetil, “Construction and use of linear regression models for processor performance analysis”, IEEE International Conference on High-Performance Computer Architecture (HPCA), 2006.

[165] B.C.Lee, D.M.Brooks, “Accurate and efficient regression modeling for microarchitectural performance and power prediction”, IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[166] G.Papadimitriou, A.Chatzidimitriou, M.Kaliorakis, Y.Vastakis, D.Gizopoulos, “Micro-Viruses for fast system-level voltage margins characterization in multicore CPUs”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2018.

[167] M.Kaliorakis, A.Chatzidimitriou, G.Papadimitriou, D.Gizopoulos, “Statistical analysis of multicore CPUs operation in scaled voltage conditions”, IEEE Computer Architecture Letters (CAL), Jan. 2018.