


National Technical University of Athens
Department of Electrical and Computer Engineering
Computer Science Division

Technical University of Denmark
Department of Space Research and Technology

Microarchitectural Approaches to Fault Tolerance in Spaceborne Processors

Diploma Thesis

Konstantinos-Nikolaos Papadopoulos, 03118220

Advisors: Dionisios N. Pnevmatikatos, Professor,
School of Electrical and Computer Engineering, NTUA

José M.G. Merayo, Professor,
Department of Space Research and Technology, DTU

April, 2024

National Technical University of Athens
Department of Electrical and Computer Engineering
Computer Science Division

Technical University of Denmark
Department of Space Research and Technology

Microarchitectural Approaches to Fault Tolerance in Spaceborne Processors

Diploma Thesis

Konstantinos-Nikolaos Papadopoulos, 03118220

Advisors: Dionisios N. Pnevmatikatos, Professor,
School of Electrical and Computer Engineering, NTUA

José M.G. Merayo, Professor,
Department of Space Research and Technology, DTU

Approved by the examination committees on April 2nd, 2024 and October 30th, 2023, respectively:

From NTUA: Dionisios N. Pnevmatikatos, Professor,
School of Electrical and Computer Engineering, NTUA

Nektarios Koziris, Professor,
School of Electrical and Computer Engineering, NTUA

Georgios Goumas, Associate Professor,
School of Electrical and Computer Engineering, NTUA

From DTU: José M.G. Merayo, Professor,
Department of Space Research and Technology, DTU

Juan Jose Vegas Olmos, Principal Engineer - Research Program Coordinator,
NVIDIA

April, 2024

Microarchitectural Approaches to Fault Tolerance in Spaceborne Processors

Diploma Thesis
April, 2024

Konstantinos-Nikolaos Papadopoulos, Electrical and Computer Engineering Graduate

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Space Research and Technology, Elektrovej, Building 327, 2800 Kgs. Lyngby Denmark
www.space.dtu.dk
ISSN: [0000-0000] (electronic version)
ISBN: [000-00-0000-000-0] (electronic version)

ISSN: [0000-0000] (printed version)
ISBN: [000-00-0000-000-0] (printed version)

Approval

This thesis has been jointly supervised by Technical University of Denmark -where it was conducted during an Erasmus exchange program- and National Technical University of Athens, author's home institution. The work has been undertaken in partial fulfillment of the requirements for the Electrical and Computer Engineering Integrated MSc and BSc diploma awarded by National Technical Univesity of Athens and in accordance with the Erasmus agreement between the two institutions.

Konstantinos-Nikolaos Papadopoulos

.....

Signature

.....

Date

Abstract

On-board computer systems in space applications -such as satellites or spacecraft- are prone to errors caused by the effects of radiation and thus require increased robustness against faults in order to successfully complete their critical and costly missions. The majority of existing and conventional approaches to fault tolerance use space or time redundancy, resulting in robust designs with high fault coverage. Nevertheless, in these methods, robustness comes at the cost of sacrificing performance, area, and power efficiency, since excessive hardware or execution are duplicated. This thesis examines approaches to fault tolerance that leverage microarchitectural insights by comparing 3 methods in various axes. By implementing and evaluating all techniques on a cycle accurate computer system simulator we demonstrate that novel ideas from computer architecture can be utilized to produce more efficient solutions for fault tolerance which also meet the unique requirements of spaceborne systems.

Keywords: fault-tolerance, microarchitecture, space

Acknowledgements

I would like to sincerely thank my supervisor from DTU, José M.G. Merayo, first of all for giving me the opportunity to work in the DTU Space department, as well as for his invaluable guidance in a new for me subject, his support in a new for me country, and his trust in myself from the very beginning.

I also want to express my great appreciation to my supervisor from NTUA, Dionisios N. Pnevmatikatos, who inspired me through his teaching and his distinct way of thinking to work and pursue a PhD in the field of computer architecture. I am grateful for his patience and for all the time he devoted providing valuable ideas, corrections, and experience.

I owe many thanks to Christina Giannoula and Nikolaos Papadopoulos for their advice, constant encouragement, and genuine interest in the project. I want to also thank Georgios Papadimitriou for his technical advice and insightful research suggestions.

I would also like to thank my friends, new and old ones. My stay in Denmark was made more enjoyable and fascinating by people like Mario, Sara, Alex, Dénes, Luanna, and Tjorven who made me feel like I had known them for years. At the same time, I am happy I spent the last years with long-lasting friends like Giorgos, Elli, Simos, Apostolis, Evangelia, Thanassis, and many more. I am glad we have traveled this path together and I am looking forward to the next ones.

Lastly, I would like to thank my parents for the values they passed on to me, and most importantly, for being always by my side.

Contents

0 Εκτεταμένη Ελληνική Περίληψη

0.1 Εισαγωγή

Το διάστημα αποτελεί ένα εχθρικό περιβάλλον για τα ηλεκτρονικά. Λόγω της έλλειψης ατμόσφαιρας και της παρουσίας πληθώρας πηγών ακτινοβολίας, όπως οι κοσμικές ακτίνες ή ο ήλιος, τα υπολογιστικά συστήματα σε στις διαστημικές αποστολές είναι ευάλωτα σε σφάλματα που προκαλούνται από τη αυξημένη ακτινοβολία που ενυπάρχει στο διάστημα. Ιδιότητες όπως η αξιοπιστία και η ανοχή σε σφάλματα, είναι πρωταρχικής σημασίας, καθώς υπολογιστικά συστήματα εκτελούν κρίσιμες εργασίες και οποιαδήποτε δυσλειτουργία μπορεί να αποδειχθεί επικίνδυνη και καταστροφική. Από τα πρώτα διαστημικά προγράμματα μέχρι σήμερα, έχουν χρησιμοποιηθεί διάφορες μέθοδοι ανοχής σφαλμάτων που μειώνουν με επιτυχία τον κίνδυνο και αποτρέπουν τις αποτυχίες. Για τα υπολογιστικά συστήματα, οι τεχνικές αυτές ανιχνεύουν σφάλματα, απομονώνουν τα ελαττωματικά εξαρτήματα και αποκαθιστούν τη σωστή λειτουργία του συστήματος. Στην παρούσα εργασία, εστιάζουμε σε μεθόδους που εξασφαλίζουν ανοχή σε σφάλματα στην κεντρική μονάδα επεξεργασίας (CPU) οποιουδήποτε υπολογιστικού συστήματος σε τροχία. Ακόμη και αν η πλειονότητα των συμβατικών τεχνικών ανοχής σφαλμάτων για CPUs παρουσιάζουν υψηλές ικανότητες στην ανίχνευση σφαλμάτων που προκαλούνται από ακτινοβολία, αυξάνουν σημαντικά την καταναλισκώμενη ενέργεια και τη συνολική επιφάνεια του τσιπ. Αυτό οφείλεται στο γεγονός ότι οι συμβατικές μέθοδοι συνήθως πολλαπλασιάζουν χωρικά μεγάλα τμήματα του υλικού ή της εκτέλεσης για να δημιουργήσουν πλεονάζοντα αντίγραφα. Η κατανάλωση ενέργειας αποτελεί σημαντικό περιορισμό για τα διαστημικά τεχνολογικά συστήματα, δεδομένου ότι τα διαστημικά οχήματα έχουν περιορισμένη χωρητικότητα μπαταριών. Η επιφάνεια του τσιπ είναι επίσης σημαντική, δεδομένου ότι υλικό μεγαλύτερης επιφάνειας καταναλώνει περισσότερη ενέργεια και είναι επίσης πιο ευάλωτο, καθώς έχει μεγαλύτερη πιθανότητα να χτυπηθεί από σωματίδια ακτινοβολίας. Για τους λόγους αυτούς, είναι επιθυμητές οι προσεγγίσεις ανοχής σφαλμάτων σε CPUs που συνδυάζουν αποδοτικότητα ανίχνευσης σφαλμάτων με χαμηλό κόστος σε προστιθέμενη ισχύ και επιφάνεια.

Η μικροαρχιτεκτονική είναι ένας υποτομέας της αρχιτεκτονικής υπολογιστών με αντικείμενο μελέτης την εσωτερική οργάνωση και υλοποίηση ενός επεξεργαστή, που παραδοσιακά προσφέρει σχεδιαστικές λύσεις που εξισορροπούν διαφορετικούς περιορισμούς, όπως η απόδοση και ο ρυθμός μετάδοσης, η κατανάλωση ενέργειας, το κόστος κ.λπ. Ως εκ τούτου, νέες ιδέες από αυτόν τον τομέα μπορούν να χρησιμοποιηθούν για τη βελτιστοποίηση προσεγγίσεων ανοχής σφαλμάτων και τη δημιουργία εύρωστων επεξεργαστών που ικανοποιούν επίσης τις αυστηρές απαιτήσεις κατανάλωσης ενέργειας και έκτασης. Προηγούμενες εργασίες στην αρχιτεκτονική υπολογιστών έχουν πράγματι ήδη παρουσιάσει τέτοιες προσεγγίσεις. Ωστόσο, αυτές οι υπάρχουσες προσεγγίσεις δεν έχουν μελετηθεί αρκετά ώστε να αποδειχθεί ότι είναι κατάλληλες για πραγματικές εφαρμογές στο διάστημα, καθώς σε πολλές περιπτώσεις δεν έχει αξιολογηθεί η ανίχνευση σφαλμάτων, ενώ σε άλλες περιπτώσεις γίνονται μη τεκμηριωμένες υποθέσεις για το διαστημικό περιβάλλον, ή δεν έχουν γίνει κατανοητές οι ιδιαίτερες απαιτήσεις των συστημάτων εν τροχιά.

Στην παρούσα εργασία επιδιώκουμε να γεφυρώσουμε το χάσμα μεταξύ διαστημικών συστημάτων και των προσεγγίσεων ανοχής σφαλμάτων που εκμεταλλεύονται τις τάσεις και ιδέες της μικροαρχιτεκτονικής, δείχνοντας ότι οι τελευταίες μπορούν να αυξήσουν την αξιοπιστία των υπολογιστικών συστημάτων και να μειώσουν την κατανάλωση ενέργειας

και την επιφάνεια υλικού, σε σύγκριση με τις συμβατικές μεθόδους που χρησιμοποιούνται σήμερα στο διάστημα.

Στην παρούσα μελέτη, συγκρίνουμε 3 μεθόδους:

- Dual Modular Redundancy, η οποία χρησιμοποιείται ευρέως στα διαστημικά συστήματα, αλλά επιφέρει υψηλές αυξήσεις σε υλικό και ενέργεια.
- Redundant Execution with Simultaneous Multithreading [1], η οποία αξιοποιεί υποσχόμενες τεχνικές από την αρχιτεκτονική υπολογιστών, αλλά δεν έχει αξιολογηθεί επαρκώς όσον αφορά την ανίχνευση σφαλμάτων.
- Parallel Detection with Heterogenous Cores [2], η οποία αντιπροσωπεύει την τρέχουσα βέλτιστη τεχνική όσον αφορά την επιβάρυνση στην απόδοση και στην επιφάνεια.

Υλοποιούμε και αξιολογούμε όλες τις τεχνικές σε έναν προσομοιούμενο επεξεργαστή, επεκτείνοντας έναν προσομοιωτή ανοικτού κώδικα που χρησιμοποιείται ευρέως στην έρευνα αρχιτεκτονικής υπολογιστών. Στη συνέχεια, διεξάγουμε μια συγκριτική αξιολόγηση αυτών των τεχνικών, με κριτήρια την αποτελεσματικότητα του εντοπισμού σφαλμάτων, την καθυστέρηση στην ανίχνευση σφαλμάτων, την επιβάρυνση που εισάγεται στην απόδοση του επεξεργαστή και την αύξηση της επιφάνειας στο τσιπ, δηλαδή κριτήρια ευθυγραμμισμένα με τις μοναδικές απαιτήσεις των συστημάτων διαστημικών σκαφών. Για το σκοπό αυτό, σχεδιάζουμε και υλοποιούμε προσομοιούμενη έγχυση σφαλμάτων, ήτοι εισάγουμε τεχνητά σφάλματα στον επεξεργαστή.

Θεωρούμε ότι η έρευνά μας είναι επίκαιρη, καθώς βρίσκεται στο σημείο τομής κρίσιμων τεχνολογικών αλλαγών (αυξημένα ποσοστά σφαλμάτων λόγω συρρίκνωσης των τρανζίστορ, διείσδυση του υπολογιστικού νέφους στο διάστημα) και μπορεί ενδεχομένως να προσφέρει λύσεις στις τρέχουσες και μελλοντικές προκλήσεις στον τομέα των διαστημικών υπολογιστών.

0.2 Υπολογιστικά Συστήματα σε Διαστημικές Εφαρμογές

Το 1961, η διαστημική κάψουλα Mercury του πρώτου επανδρωμένου διαστημικού προγράμματος των Ηνωμένων Πολιτειών, λειτουργούσε χωρίς κανέναν υπολογιστή [3]. Οι κεντρικοί υπολογιστές στη Γη εκτελούσαν όλους τους απαραίτητους υπολογισμούς για τον έλεγχο της τροχιάς, οι οποίοι στη συνέχεια μεταδίδονταν μέσω ραδιοφώνου στο διαστημόπλοιο. Το ίδιο ίσχυε και για τα πρώτα 15 χρόνια της μη επανδρωμένης εξερεύνησης του διαστήματος, καθώς και για τη Σοβιετική Ένωση. Ωστόσο, 8 χρόνια αργότερα, το 1969, η προσεδάφιση στη Σελήνη θα ήταν ακατόρθωτη χωρίς τη χρήση ενσωματωμένων υπολογιστικών συστημάτων. Για την ακρίβεια, η έρευνα και ανάπτυξη για τα υπολογιστικά συστήματα των αποστολών Apollo από το 1962 έως το 1968 κατανάλωσε τα δύο τρίτα της παγκόσμιας προμήθειας ολοκληρωμένων κυκλωμάτων [4]. Σήμερα, οι υπολογιστές αποτελούν αναπόσπαστα στοιχεία όλων των διαστημικών σκαφών και των συστημάτων σε τροχιά, μέρος κάθε υποσυστήματος, υποστηρίζοντας όλες τις λειτουργίες.

Επιπλέον, αξίζει να σημειωθεί ότι πρόσφατες τάσεις, όπως το υπολογιστικό νέφος (cloud computing) ή η υπολογιστική αιχμής (edge computing), διεισδύουν στα δορυφορικά συστήματα [5], ιδίως δεδομένης της αυξανόμενης πρόσφατης ανάπτυξης δορυφορικών αστερισμών (satellite constellations). Αυτά τα υπολογιστικά παραδείγματα, επιτρέπουν την επεξεργασία δεδομένων πιο κοντά στις πηγές παραγωγής τους και, ως εκ τούτου, μπορούν να μειώσουν σημαντικά τις απαιτήσεις του εύρους ζώνης στη κατερχόμενη ζεύξη (downlink) και να αυξήσουν τη διαθεσιμότητα. Ωστόσο, μετασχηματίζουν το σκηνικό των απαιτήσεων των υπολογιστικών συστημάτων που βρίσκονται σε τροχιά, αφού απαιτούν

συστήματα με υψηλότερες δυνατότητες επεξεργασίας. Ως εκ τούτου, πιστεύουμε ότι η παρούσα εργασία, η οποία εξετάζει την ανοχή σε σφάλματα λαμβάνοντας υπόψη και άλλες απαιτήσεις, όπως η απόδοση, είναι επίκαιρη, δεδομένου ότι τα διαστημικά συστήματα υπολογιστών γίνονται όλο και πιο πολύπλοκα και ισχυρά.

0.2.1 Απαιτήσεις για διαστημικά υπολογιστικά συστήματα

Τα διαστημικά υπολογιστικά συστήματα έχουν ένα σύνολο ιδιαίτερων απαιτήσεων λόγω τόσο της φύσης του διαστημικού περιβάλλοντος όσο και της κρισιμότητας και του κόστους της διαστημικής εξερεύνησης. Τα εξωγήινα περιβάλλοντα είναι αφιλόξενα όχι μόνο για τα ηλεκτρονικά και υπολογιστικά συστήματα αλλά και για τα περισσότερα εξαρτήματα ενός διαστημικού σκάφους. Η διαστημική τεχνολογία πρέπει να αντέχει σε ακραίες δυνάμεις και δονήσεις, σε ένα ευρύ φάσμα θερμοκρασιών και σε ακτινοβολία. Επιπλέον, τα συστήματα πρέπει να λειτουργούν καταναλώνοντας ελάχιστη ενέργεια, δεδομένου ότι οι πηγές ενέργειας στο διάστημα είναι περιορισμένες. Τούτοι οι περιορισμοί είναι παρόντες τόσο στο σχεδιασμό όσο και στη λειτουργία των διαστημικών υπολογιστικών συστημάτων, με τη διαθεσιμότητα και την αξιοπιστία να είναι οι δύο θεμελιώδεις ιδιότητες που πρέπει να ικανοποιούν τα συστήματα υπολογιστών στο διάστημα. Έτσι, από τους κεντρικούς άξονες του σχεδιασμού των υπολογιστικών συστημάτων που θα λειτουργούν στο διάστημα είναι η ελαχιστοποίηση του κινδύνου. Μαζί με αυτό, τα συστήματα πρέπει να είναι σε θέση να ανιχνεύουν γρήγορα πιθανά σφάλματα. Επιπλέον, τα ηλεκτρονικά συστήματα πρέπει να καταναλώνουν ελάχιστη ενέργεια.

Η ακτινοβολία είναι ωστόσο μια από τις κύριες πηγές σφαλμάτων. Τα συστήματα στο διάστημα δεν προστατεύονται από την ατμόσφαιρα και όντας εκτεθειμένα σε υψηλά επίπεδα ακτινοβολίας είναι ευάλωτα σε σφάλματα που προκαλούνται από αυτήν. Στην παρούσα εργασία, εστιάζουμε στα σφάλματα που προκαλούνται στους επεξεργαστές υπολογιστών αποκλειστικά από ακτινοβολία.

0.3 Σφάλματα Προκαλούμενα από Ακτινοβολία

Μία από τις πιο αξιοσημείωτες διαφορές μεταξύ της Γης και του Διαστήματος είναι τα υψηλότερα επίπεδα ακτινοβολίας που υπάρχουν στο Διάστημα. Αν και η Γη προστατεύεται από την ατμόσφαιρα και το μαγνητικό πεδίο και έτσι διατηρεί τις χαμηλές συνθήκες ακτινοβολίας που είναι απαραίτητες για την ανθρώπινη ζωή, δεν ισχύει το ίδιο για τα εξωγήινα περιβάλλοντα, όπου η ακτινοβολία αποτελεί σημαντικό κίνδυνο τόσο για τους αστροναύτες που ταξιδεύουν σε επανδρωμένες αποστολές, όσο και για τον εξοπλισμό και τα όργανα των διαστημοπλοίων.

Η ακτινοβολία στο διάστημα προέρχεται από 3 πηγές: τις κοσμικές ακτίνες, την περιοχή ακτινοβολίας Van Allen και τα ηλιακά σωματίδια. Σωματίδια από όλες αυτές τις πηγές ακτινοβολίας, όταν αλληλεπιδρούν με τα ηλεκτρονικά, προκαλούν αλληλεπιδράσεις που μπορούν να κατηγοριοποιηθούν ως εξής:

Συνολική ιοντίζουσα δόση (Total Ionizing Dose): αναφέρεται στις μακροχρόνιες επιπτώσεις ακτινοβολίας που προκαλούνται από την ενέργεια που μεταφέρεται μόνο μέσω ιονισμού, από τα προσπίπτοντα σωματίδια στην ηλεκτρονική συσκευή. Είναι επομένως ένα αθροιστικό αποτέλεσμα που σε ατομικό επίπεδο μετακινεί τα ηλεκτρόνια σε υψηλότερες ενεργειακές καταστάσεις.

Δόση μετατόπισης (Displacement Dose): επίσης ένα αθροιστικό αποτέλεσμα της ακτινοβολίας κατά το οποίο τα υψηλής ενέργειας προσπίπτοντα σωματίδια δεν διεγείρουν τα ιόντα του πυρητίου του υλικού αλλά αυτή τη φορά μετατοπίζουν ολόκληρα άτομα του υλικού, δημιουργώντας κενά στο κρυσταλλικό πλέγμα του ημιαγωγού.

Αποτελέσματα μεμονωμένου γεγονότος (ΑΜΓ) (Single Event Effects): Κάτω από αυτόν τον όρο (SEE), συνδυάζουμε όλες τις επιδράσεις που προκαλούνται από τη ροή σωματιδίων ακτινοβολίας μέσω μιας ηλεκτρονικής συσκευής. Μεταξύ των διαφόρων ΑΜΓ, τα Single Event Upsets (SEU) είναι μεταβατικά σφάλματα που εμφανίζονται ως παροδικοί παλμοί σε συνδυαστικά κυκλώματα ή ως αλλαγές στην κατάσταση ενός bit (bitflips) σε στοιχεία μνήμης.

0.3.1 Μηχανισμοί αλληλεπίδρασης με το υλικό

Δυναμική μνήμη τυχαίας προσπέλασης (DRAM): Η διατήρηση της πληροφορίας στα κελιά μνήμης DRAM είναι παθητική και αυτό έχει δύο επιπτώσεις στη συμπεριφορά των διατάξεων DRAM υπό SEEs. Πρώτον, αυτό σημαίνει ότι δεν υπάρχει εγγενής μηχανισμός στη λειτουργία του κελιού μνήμης που θα μπορούσε να οδηγήσει σε αυτόματη διόρθωση κάποιου SEE (όπως μπορεί να συμβεί σε ορισμένες περιπτώσεις στην SRAM). Δεύτερον, επειδή και πάλι το σήμα αποθηκεύεται στο φορτίο του πυκνωτή του κελιού, μια SEU μπορεί να εμφανιστεί όχι μόνο λόγω της μετάβασης από μια σταθερή κατάσταση σε μια άλλη αλλά και λόγω της υποβάθμισης του αποθηκευμένου σήματος εκτός κάποιων περιθωρίων θορύβου [6].

Στατική μνήμη τυχαίας προσπέλασης (SRAM): Η ευαισθησία των διατάξεων SRAM σε SEEs εξαρτάται σε μεγάλο βαθμό από τη θέση πρόσπτωσης του σωματιδίου μέσα στο κύκλωμα του κελιού, δηλαδή το συγκεκριμένο τρανζίστορ του κελιού [7]. Τα σωματίδια της ακτινοβολίας θα συσσωρεύσουν φορτίο σε έναν από τους ακροδέκτες του κτυπημένου τρανζίστορ προκαλώντας, με τη σειρά τους, ροή ρεύματος μεταξύ των 2 τρανζίστορ της ημιγέφυρας, οδηγώντας σε μεταβατική τάση. Αυτό το μεταβατικό ρεύμα τάσης, επειδή συνήθως συμβαίνει στον ακροδέκτη απαγωγού (drain) του τρανζίστορ, μπορεί να λειτουργήσει ως παλμός εγγραφής, αλλάζοντας την εγγεγραμμένη τιμή της επηρεαζόμενου κελιού αποθήκευσης.

Λογικά κυκλώματα: Για τα λογικά κυκλώματα, τα (μεταβατικά) σφάλματα που προκαλούνται από SEEs εκδηλώνονται όταν ένα μεταβατικό ρεύμα τάσης από μια γραμμή σήματος συλλαμβάνεται σε κάποιο μανδαλωτή [7]. Για να συμβεί αυτό, ένα σωματίδιο ακτινοβολίας επαρκούς ενέργειας πρέπει να χτυπήσει μια ενεργή γραμμή του κυκλώματος, ενώ πρέπει να υπάρχει και ενεργή διαδρομή από την παγιδευμένη γραμμή σε κάποιον μανδαλωτή μέσω άλλων γραμμών ή/και στοιχείων του κυκλώματος (όπως πύλες κλπ.). Με αυτόν τον τρόπο η λογική τιμή του μανδαλωτή θα μεταβληθεί και το μεταβατικό σφάλμα θα εκδηλωθεί στο κύκλωμα.

Για όλους τους παραπάνω λόγους, σε αυτή τη μελέτη θα μοντελοποιήσουμε όλα τα σφάλματα στη λογική του επεξεργαστή ως bitflips σε στοιχεία μανδαλώσεων.

0.4 Έννοιες Αρχιτεκτονικής Υπολογιστών

Ένα βασικό θέμα σε αυτή τη μελέτη είναι η μικροαρχιτεκτονική. Η μικροαρχιτεκτονική μελετά τον σχεδιασμό και την υλοποίηση ενός επεξεργαστή περιγράφοντας μεταξύ άλλων την διοχέτευση και τα διάφορα στάδια αυτής ή την ιεραρχία της κρυφής μνήμης. Ένας από τους σημαντικότερους σχεδιαστικούς στόχους κατά την ανάπτυξη νέων μικροαρχιτεκτονικών είναι η απόδοση του επεξεργαστή, δεδομένου ότι η έρευνα και ο σχεδιασμός νέων επεξεργαστών αποσκοπεί στη βελτίωση της απόδοσης και της ρυθμαπόδοσης (throughput). Για το λόγο αυτό, τις τελευταίες δεκαετίες έχει αναπτυχθεί ένα ευρύ φάσμα τεχνικών με στόχο τη βελτιστοποίηση της απόδοσης των σωληνώσεων επεξεργαστών. Ορισμένες από αυτές αξιοποιούνται από τις τεχνικές ανοχής σφαλμάτων της παρούσας μελέτης και θα τις παρουσιάσουμε εδώ.

Υπερβαθμωτοί επεξεργαστές: Μια σημαντική ιδιότητα που επηρεάζει σε μεγάλο βαθμό την απόδοση είναι ο μέσος ρυθμός εκτέλεσης εντολών του επεξεργαστή, ο οποίος μπορεί να ποσοτικοποιηθεί με τη μετρική των εντολών ανά κύκλο (IPC). Οι σωληνώσεις που μπορούν να εκκινούν και να εκτελούν μόνο 1 εντολή ανά κύκλο ρολογιού, μπορούν να επιτύχουν την καλύτερη δυνατή απόδοση 1 εντολής ανά κύκλο (IPC<=1).Για να αυξηθεί περαιτέρω η απόδοση, έχουν σχεδιαστεί υπερβαθμωτοί επεξεργαστές, οι οποίοι έχουν IPC>1. Οι υπερβαθμωτοί επεξεργαστές είναι σε θέση να εκτελούν ταυτόχρονα περισσότερες από 1 εντολή σε όλα τα στάδια της σωλήνωσης, προχωρώντας ταυτόχρονα πολλαπλές εντολές σε κάθε στάδιο.

Δυναμική δρομολόγηση: Ακόμη και με υπερβαθμωτούς επεξεργαστές, αν η εκτέλεση γίνεται σύμφωνα με την αρχική σειρά του προγράμματος (στατική δρομολόγηση) και μια εντολή προκαλέσει καθυστέρηση στη διοχέτευση, όλες οι επόμενες εντολές δεν μπορούν να προχωρήσουν, οδηγώντας σε μείωση της απόδοσης της διοχέτευσης. Οι επεξεργαστές εκτός σειράς μπορούν να εκτελούν εντολές με διαφορετική σειρά από αυτή που εμφανίζονται στο πρόγραμμα (δυναμική δρομολόγηση), οι οποίες εκδίδονται αμέσως μόλις τα τελούμενά τους γίνουν διαθέσιμα και μια κατάλληλη λειτουργική μονάδα είναι ελεύθερη. Με αυτόν τον τρόπο, οι επόμενες εντολές μπορούν να παρακάμψουν μια εντολή που καθυστερεί- ο δυναμικός προγραμματισμός δημιουργεί παραλληλισμό που δεν είναι διαθέσιμος κατά τη μεταγλώττιση (παραλληλισμός σε επίπεδο εντολών (ILP)). Ωστόσο, στις σωληνώσεις εκτός σειράς, η εκκίνηση και η ολοκλήρωση των εντολών γίνονται σε σειρά.

Ταυτόχρονη πολυνημάτωση: Ακόμη και με όλες αυτές τις βελτιστοποιήσεις, υπάρχουν περιπτώσεις όπου δεν αξιοποιείται όλος ο διαθέσιμος παραλληλισμός σε επίπεδο εντολών. Ορισμένες λειτουργικές μονάδες αδρανοποιούνται επειδή το μίγμα εντολών του προγράμματος περιέχει πολλές εντολές του ίδιου τύπου ή επειδή οι εξαρτήσεις εντολών απαγορεύουν την εκτέλεση εκτός σειράς. Μια λύση σε αυτό είναι η Ταυτόχρονη Πολυνημάτωση (Simultaneous Multithreading (SMT)) [8], μια τεχνική όπου ο επεξεργαστής εκτελεί ταυτόχρονα περισσότερα από ένα νήματα σε έναν πυρήνα, αξιοποιώντας με αυτόν τον τρόπο όλες τις διαθέσιμες λειτουργικές μονάδες, επιτρέποντας την επίτευξη υψηλότερης απόδοσης του επεξεργαστή.

0.5 Περιγραφή των υπό Μελέτη Τεχνικών Ανίχνευσης Σφαλμάτων

Μεταξύ των 3 μεθόδων ανίχνευσης σφαλμάτων που μελετήθηκαν στην παρούσα εργασία, η πρώτη μέθοδος (Dual Modular Redundancy) έχει χρησιμοποιηθεί ευρέως σε πληθώρα πραγματικών διαστημικών αποστολών και ενώ παρουσιάζει την καλύτερη ικανότητα ανίχνευσης σφαλμάτων, απαιτεί εκτεταμένα μεγάλη επιφάνεια τσιπ και κατανάλωση ισχύος. Οι άλλες 2 μέθοδοι (Redundant Multithreading και Parallel Error Detection) έχουν προταθεί από ερευνητές υπολογιστικών συστημάτων και παρόλο που δεν έχουν δοκιμαστεί σε διαστημικές επιχειρήσεις, θεωρητικά βελτιώνουν το κόστος ισχύος και επιφάνειας, ενώ υποβαθμίζουν μέτρια την απόδοση του συστήματος, μέσω της χρήσης μικροαρχιτεκτονικών τεχνικών από την έρευνα αρχιτεκτονικής υπολογιστών. Η ικανότητα ανίχνευσης σφαλμάτων των δύο τελευταίων μεθόδων είναι ένα από τα αναμενόμενα αποτελέσματα αυτής της μελέτης.

Και οι 3 μέθοδοι επικεντρώνονται μόνο στην ανίχνευση σφαλμάτων με τεχνικές βασισμένες στο υλικό και ασφαλίζουν μόνο τη λογική της διοχέτευση του επεξεργαστή, υποθέτοντας ότι η μνήμη προστατεύεται με άλλες μεθόδους, όπως οι κώδικες διόρθωσης σφαλμάτων.

0.5.1 Dual Modular Redundancy

Το Dual Modular Redundancy (DMR) είναι μια τεχνική χωρικού πλεονασμού (redundancy) όπου κάθε στοιχείο υλικού αντιγράφεται δύο φορές και ο υπολογισμός επαναλαμβάνεται και στα δύο αντίγραφα του συστήματος, για να αυξηθεί η αξιοπιστία. Είναι επίσης η πρώτη μέθοδος που θα μελετήσουμε στην παρούσα εργασία. Για ένα υπολογιστικό σύστημα, το DMR πραγματοποιείται με το σχηματισμό ενός συστήματος με 2 επεξεργαστές αντί για 1 και την εκτέλεση του προγράμματος στον καθένα. Με αυτό το σύστημα, τα σφάλματα εντοπίζονται συγκρίνοντας τα αποτελέσματα της εκτέλεσης από τους 2 επεξεργαστές. Η μνήμη πρέπει είτε να είναι επίσης διπλή είτε να προστατεύεται, π.χ. με κώδικες διόρθωσης σφαλμάτων.

Το DMR γενικά είναι μια τεχνική που μεταδίδει υψηλή ευρωστία στο σύστημα, καθώς οποιοδήποτε σφάλμα που προκαλείται από ακτινοβολία θα πλήξει μόνο τον έναν από τους δύο επεξεργαστές και συνεπώς θα αλλάξει τα αποτελέσματα μόνο του ενός, κάτι που θα ανιχνευθεί στη συνέχεια. Κάνουμε την (λογική) υπόθεση ότι η πιθανότητα να χτυπήσουν δύο σωματίδια και να προκαλέσουν σφάλμα και στους δύο επεξεργαστές (επιτρέποντας σε ένα σφάλμα να παραμείνει απαρατήρητο και να προκαλέσει αποτυχία του συστήματος) είναι αμελητέα. Ωστόσο, τα συστήματα DMR έχουν 2 σημαντικά μειονεκτήματα. Πρώτον, το DMR απαιτεί διπλασιασμό όλων των μονάδων υλικού, εισάγοντας διπλάσια επιβάρυνση σε επιφάνεια. Η αυξημένη επιφάνεια είναι ιδιαίτερα ανεπιθύμητη σε όλα τα συστήματα υλικού, επειδή συνοδεύεται από αυξημένη κατανάλωση ενέργειας καθώς και από αυξημένο κόστος κατασκευής/συσκευασίας (packaging). Ειδικά στην περίπτωση των συστημάτων με ανοχή στην ακτινοβολία, η αύξηση της επιφάνειας του υλικού συνεπάγεται επίσης αύξηση της πιθανότητας εμφάνισης σφαλμάτων. Αυτό συμβαίνει επειδή τα συστήματα που καταλαμβάνουν μεγαλύτερη επιφάνεια έχουν μεγαλύτερη πιθανότητα να προσβληθούν από σωματίδια ακτινοβολίας, σε σύγκριση με συστήματα μικρότερης επιφάνειας. Δεύτερον, τα συστήματα DMR χρειάζονται μια μονάδα ψηφοφορίας (voter) που ανιχνεύει τα σφάλματα συγκρίνοντας τα αποτελέσματα από τις αντιγραμμένες μονάδες (π.χ. CPU). Εξ ορισμού, αυτό το στοιχείο πρέπει να είναι μοναδικό και συνεπώς είναι ευάλωτο και απροστάτευτο. Κατά συνέπεια, η εκδήλωση ενός σφάλματος στον ψηφοφόρο μπορεί να οδηγήσει σε αποτυχία του συστήματος και για να μετριαστεί αυτό, πρέπει να χρησιμοποιηθούν εξαρτήματα ανθεκτικά στην ακτινοβολία στη συγκεκριμένη μονάδα, αυξάνοντας περαιτέρω το κόστος κατασκευής. Το τελευταίο σημείο έχει μεγάλη σημασία, δεδομένου ότι, οι κατασκευαστές εξαρτημάτων ανθεκτικών στην ακτινοβολία μειώνονται.

0.5.2 Πλεονάζων Πολυνηματισμός (Redundant Multithreading)

Ο Πλεονάζων Πολυνηματισμός (Redundant Multithreading) [9]–[11] είναι μια κλάση τεχνικών ανοχής σε σφάλματα με πλεονασμό χρόνου (time redundancy), όπου η πλεονάζουσα εκτέλεση συμβαίνει σε αρχιτεκτονικό επίπεδο και υλοποιείται με διαφορετικά νήματα που εκτελούνται στον επεξεργαστή. Με την πλεονάζοντα πολυνηματισμό, κάθε εντολή επαναλαμβάνεται όχι σε ξεχωριστούς πυρήνες της CPU (όπως στην περίπτωση του χωρικού DMR) αλλά σε διαφορετικά νήματα του προγράμματος στον ίδιο πυρήνα της CPU, καθένα από τα οποία εκτελεί τις ίδιες εντολές πλεοναστικά και στη συνέχεια επικυρώνει ότι όλες οι εκτελέσεις παρήγαγαν τα ίδια αποτελέσματα, άρα δεν είχαν συμβεί σφάλματα.

Η παραλλαγή του Πλεονάζοντος Πολυνηματισμού στην οποία θα επικεντρωθούμε ονομάζεται Πλεονάζων Πολυνηματισμός με Ταυτόχρονη Πολυνημάτωση (Redundant Multithreading with Simultaneous Multithreading (R-SMT)) [1]. Η προσέγγιση αυτή χρησιμοποιεί τη μικροαρχιτεκτονική τεχνική της ταυτόχρονης πολυνημάτωσης προκειμένου να εκτελείται αποδοτικά το πλεονάζων νήμα, δημιουργώντας 2 νήματα SMT και αναθέτοντας στο καθένα από αυτά την εκτέλεση ενός αντιγράφου του ίδιου προγράμματος. Σε αυτό το σχήμα, το πρώτο νήμα (πρωτεύον νήμα) εκτελεί τις εντολές του προγράμματος, ενώ το δεύτερο

(πλεονάζον νήμα) τις εκτελεί εκ νέου και επικυρώνει τα αποτελέσματα. Είναι χρήσιμο να διατηρείται το ένα νήμα ελαφρώς προπορευόμενο στην εκτέλεση από το άλλο. Αυτό, επιτρέπει μια σειρά από βελτιστοποιήσεις επιδόσεων, οι οποίες επικεντρώνονται στην ιδέα ότι, εφόσον και τα δύο νήματα εκτελούν τις ίδιες εντολές και αναμένουν να παράξουν τα ίδια αποτελέσματα, το προπορευόμενο (πρωτεύον) νήμα μπορεί να βοηθήσει στην εκτέλεση του ουραγού (πλεονάζοντος) νήματος, επιτρέποντας στο πλεονάζον να εκτελεστεί πιο αποτελεσματικά, χωρίς να επαναλάβει την εκτέλεση εντολών που έχουν ήδη ολοκληρωθεί από το πρωτεύον.

Πιο συγκεκριμένα, τέτοιες βελτιστοποιήσεις που καθίστανται εφικτές από την ύπαρξη του προπορευόμενου και ουραγού νήματος αφορούν προανάκληση, πρόβλεψη διακλαδώσεων και πρόβλεψη τελεστών των εντολών.

0.5.3 Παράλληλη Ανίχνευση Σφαλμάτων με χρήση Ετερογενών Πυρήνων

Η τρίτη προσέγγιση ανοχής σφαλμάτων που συμπεριλαμβάνουμε σε αυτή τη συγκριτική ανάλυση είναι η Παράλληλη Ανίχνευση Σφαλμάτων με χρήση Ετερογενών Πυρήνων (Parallel Error Detection using Heterogeneous Cores) [2]. Στόχος αυτής της μεθόδου είναι η ανίχνευση τόσο μεταβατικών όσο και μόνιμων σφαλμάτων, διατηρώντας παράλληλα χαμηλό κόστος σε προστιθέμενη επιφάνεια, ισχύ και επίδοση, καθώς και ελάχιστη επεμβατικότητα στην αρχική μικροαρχιτεκτονική. Βασική ιδέα για να επιτευχθούν αυτά σύμφωνα με τούτη την προσέγγιση είναι η παραλληλοποίηση της ανίχνευσης σφαλμάτων. Δίπλα στον κύριο επεξεργαστή εκτός σειράς, στον οποίο εκτελούνται οι εντολές, βρίσκονται βοηθητικοί πυρήνες επεξεργαστών που επαναλαμβάνουν τις ίδιες εντολές, επιτυγχάνοντας τον χωρικό πλεονασμό που απαιτείται για την ανίχνευση σφαλμάτων. Οι βοηθητικοί πυρήνες είναι χαμηλής ισχύος, εκτελούν σε σειρά και καταλαμβάνουν μικρότερη επιφάνεια, επιτρέποντας την επίτευξη των επιθυμητών χαμηλών επιβαρύνσεων ισχύος και επιφάνειας. Η εκτέλεση στον κύριο πυρήνα χωρίζεται σε τμήματα που αποτελούνται από ορισμένο αριθμό διαδοχικών εντολών και κάθε τμήμα μετά την εκτέλεσή του στον κύριο πυρήνα εκφορτώνεται σε έναν από τους βοηθητικούς πυρήνες για επανεκτέλεση και επαλήθευση. Ωστόσο, επειδή οι βοηθητικοί πυρήνες είναι μικρότεροι είναι επίσης πιο αργοί και αυτό σημαίνει ότι η επανεκτέλεση κάθε τμήματος διαρκεί περισσότερο από την αρχική εκτέλεση στον κύριο πυρήνα. Εξαιτίας αυτού, καθώς ο κύριος πυρήνας προχωρά στην εκτέλεση του προγράμματος, πολλοί βοηθητικοί πυρήνες ενδέχεται να εξακολουθούν να επαληθεύουν προηγούμενα τμήματα.

Για να επανεκτελέσουν οι βοηθητικοί πυρήνες κάθε τμήμα διαδοχικών εντολών απαιτείται η αντιγραφή της αρχιτεκτονικής κατάστασης (architectural state) από τον κύριο πυρήνα σε κάθε βοηθητικό, κάτι που καθυστερεί τον κύριο επεξεργαστή για μερικούς κύκλους. Επίσης, καθώς οι βοηθητικοί πυρήνες υστερούν στην εκτέλεση σε σχέση με τον κύριο πυρήνα, τα δεδομένα στη μνήμη ενδέχεται να πανωγραφούν από τον κύριο πυρήνα πριν διαβαστούν από τους βοηθητικούς πυρήνες, με αποτέλεσμα οι κύριοι και οι πυρήνες ελέγχου να έχουν διαβάσει διαφορετικές τιμές. Για το λόγο αυτό, οι πυρήνες ελέγχου περιορίζονται από την πρόσβαση στην κύρια μνήμη και κάθε πρόσβαση του κύριου πυρήνα στην κύρια μνήμη αναπαράγεται και αποθηκεύεται σε έναν απομονωτή υλικού (hardware buffer) που ονομάζεται αρχείο καταγραφής φόρτωσης-αποθήκευσης (load-store log). Εάν αυτό το αρχείο καταγραφής φόρτωσης-αποθήκευσης είναι γεμάτο, όλοι οι βοηθητικοί πυρήνες είναι ενεργοί επανεκτελώντας κάποιο τμήμα του προγράμματος και σε αυτή την περίπτωση, ο κύριος πυρήνας πρέπει να σταματήσει την περαιτέρω εκτέλεση και να καθυστερήσει μέχρι να απελευθερωθεί ένας βοηθητικός πυρήνας.

0.6 Μεθοδολογία

Όπως έχει γίνει σαφές στο υπόλοιπο κείμενο, ο κύριος σκοπός αυτής της μελέτης είναι να συγκρίνει τις 3 μεθόδους ανοχής σφαλμάτων σε τέσσερις άξονες: την ικανότητα ανίχνευσης σφαλμάτων, την καθυστέρηση ανίχνευσης, την επιβάρυνση στην απόδοση του επεξεργαστή και στο προστιθέμενο κόστος σε υλικό. Για να το επιτύχουμε αυτό, διεξάγουμε ένα πείραμα έγχυσης σφαλμάτων, προσομοιώνοντας την παρουσία σφαλμάτων σε 3 συστήματα -το καθένα από τα οποία χρησιμοποιεί μία από τις υπο μελέτη τεχνικές ανίχνευσης σφαλμάτων- ενώ εκτελούμε μια ποικιλία από μετροπρογράμματα από τη σουίτα MiBench [12].

Τα πειράματα που σχεδιάζουμε έχουν την εξής δομή: σε κάθε εκτέλεση προγράμματος, εισάγουμε ένα σφάλμα σε έναν τυχαίο καταχωρητή σε μια τυχαία επιλεγμένη χρονική στιγμή. Για να μπορέσουμε να συγκρίνουμε πιο δίκαια τα αποτελέσματα μεταξύ των 3 μεθόδων, διατηρούμε τους καταχωρητές που εισάγονται και τον χρόνο έγχυσης του σφάλματος ίδια μεταξύ κάθε μεθόδου. Εκτελούμε επανειλημμένα μεγάλο αριθμό εκτελέσεων, εγχέοντας σε διαφορετικούς καταχωρητές και σε διαφορετικούς χρόνους, και καταγράφουμε τις μετρικές που μας ενδιαφέρουν.

Για την ανιχνευσιμότητα, καταγράφουμε την έκβαση της εκτέλεσης, δηλαδή αν το εγχυόμενο σφάλμα οδηγεί σε ένα από τα παρακάτω:

- *συντριβή (crash)*, για παράδειγμα σε σφάλμα τμηματοποίησης (segmentation fault), λόγω παράνομης προσπέλασης μνήμης
- ακινητοποίηση του επεξεργαστή σε *μη ανακτήσιμη κατάσταση (hang)*, η οποία ανιχνεύεται μετά την παρέλευση ορισμένου χρονικού διαστήματος.
- *κανονική ολοκλήρωση της εκτέλεσης (masked)* χωρίς την εκδήλωση οποιουδήποτε σφάλματος στο σύστημα
- *ολοκλήρωση της εκτέλεσης (silent data corruption (SDC))* χωρίς την εκδήλωση οποιουδήποτε σφάλματος στο σύστημα αλλά έχοντας παράξει λανθασμένα αποτελέσματα
- επιτυχής *ανίχνευση* από τη μέθοδο ανίχνευσης σφάλματος

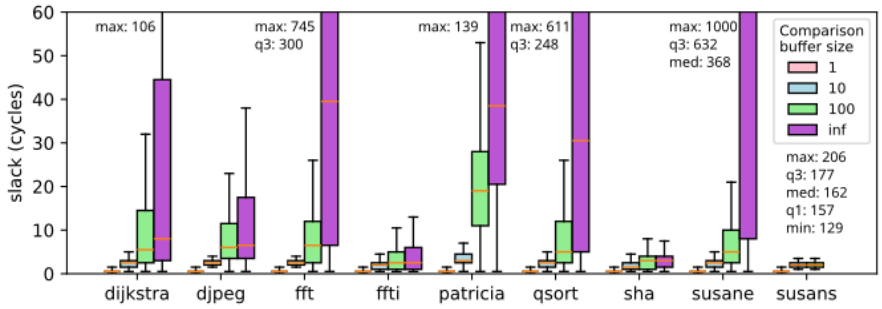
Για την καθυστέρηση ανίχνευσης σφαλμάτων μετράμε το χρόνο που μεσολαβεί μεταξύ της εισαγωγής σφάλματος και της επιτυχούς ανίχνευσης ενός σφάλματος και για την επιβάρυνση της απόδοσης μετράμε τη μετρική των εντολών ανά κύκλο (Instructions per Cycle - IPC).

0.7 Αποτελέσματα και Ανάλυση

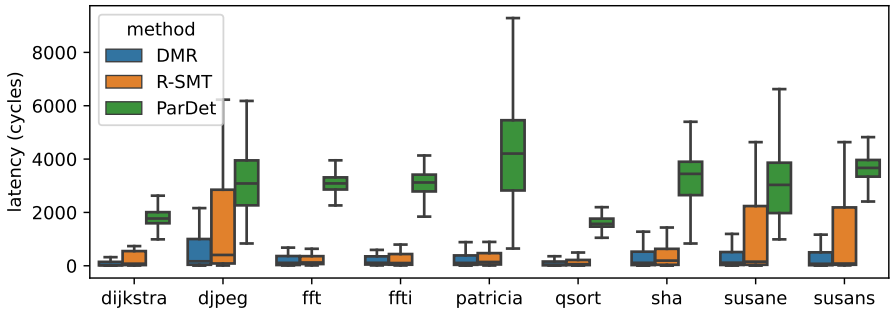
0.7.1 Καθυστέρηση ανίχνευσης σφαλμάτων

Καθυστέρηση νημάτων στο R-SMT

Στο R-SMT, η καθυστέρηση στην ολοκλήρωση μιας εντολής από το πρωτεύον νήμα και το πλεονάζον νήμα, ονομάζεται slack. Δεδομένου ότι στο R-SMT η ανίχνευση σφαλμάτων πραγματοποιείται κατά τη σύγκριση των αποτελεσμάτων της κάθε εντολής κατά την ολοκλήρωσή της, το slack είναι ένας σημαντικός παράγοντας που συμβάλλει στην καθυστέρηση ανίχνευσης. Στο Σχήμα 1, διερευνούμε πως το slack μεταβάλλεται ανάλογα με το μέγεθος του απομονωτή σύγκρισης (buffer) στον οποίο το πρωτεύον νήμα αποθηκεύει τα αποτελέσματα των εντολών προκειμένου να τα αξιοποιήσει το πλεονάζον για τη σύγκριση. Διαπιστώνουμε ότι, μικρότερα μεγέθη τούτου του buffer οδηγούν σε μικρότερο slack. Αυτό είναι αναμενόμενο, καθώς όταν το πρωτεύον νήμα έχει τοποθετήσει περισσότερες εντολές στο buffer, από όσες έχει καταναλώσει το πλεονάζον, το buffer γεμίζει



Σχήμα 1: Αναπαράσταση με θηκόγραμμα της κανονικής κατανομής της καθυστέρησης νημάτων στο R-SMT όταν μεταβάλλεται το μέγεθος του απομονωτή σύγκρισης. Κάθε ορθογώνιο περιορίζεται προς τα κάτω από το πρώτο τεταρτημόριο (quartile) και προς τα πάνω από το τρίτο τεταρτημόριο. Η διάμεσος εμπίπτει εντός του ορθογωνίου. Το ενδοτεταρτημοριακό εύρος (inter-quartile range) είναι η απόσταση μεταξύ του πρώτου και του τρίτου τεταρτημορίου (δηλ. το ύψος του ορθογωνίου). Οι ουρίτσες εκτείνονται στις ελάχιστες και μέγιστες τιμές των σημείων δεδομένων εκατέρωθεν κάθε ορθογωνίου.



Σχήμα 2: Αναπαράσταση κανονικής κατανομής της καθυστέρησης ανίχνευσης για τις 3 μεθόδους.

πλήρως και το πρωτεύον νήμα πρέπει να σταματήσει την εκτέλεση, καθώς δεν υπάρχουν διαθέσιμες καταχωρήσεις για την αποθήκευση των αποτελεσμάτων των επόμενων εντολών. Ως αποτέλεσμα, το πλεονάζον νήμα εκτελεί αντί αυτού, προχωρώντας περαιτέρω στην εκτέλεση και συγκλίνοντας με το πρωτεύον, άρα μειώνοντας το slack. Δεδομένου ότι με μικρότερα μεγέθη buffer, το πρωτεύον νήμα καθυστερεί συχνότερα, μικρότερα μεγέθη buffer οδηγούν τελικά σε μικρότερο slack μεταξύ των νημάτων.

Μέτρηση της καθυστέρησης ανίχνευσης

Όσον αφορά την καθυστέρηση ανίχνευσης, που απεικονίζεται στο Σχήμα 2 για όλες τις μεθόδους, εξάγουμε τα ακόλουθα συμπεράσματα:

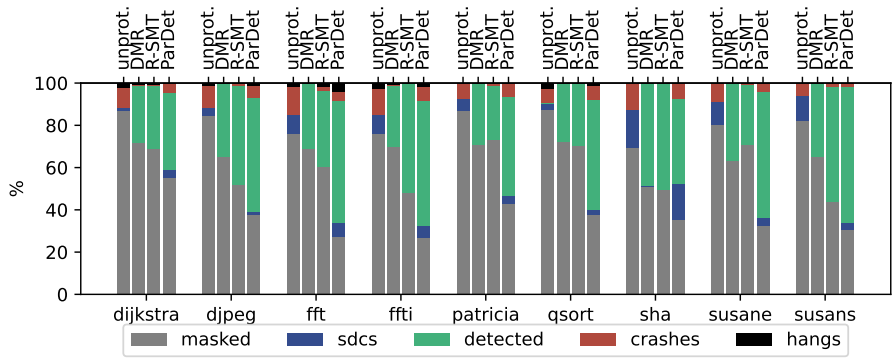
- Το DMR παρουσιάζει τις χαμηλότερες ελάχιστες (σε όλα τα μετροπρογράμματα) και μέσες (σε 8 από τα 9 μετροπρογράμματα) τιμές σε σύγκριση με τις άλλες 2 μεθόδους. Αυτό είναι αναμενόμενο αφού οι 2 επεξεργαστές εκτελούν ταυτόχρονα τις ίδιες εντολές. Ως εκ τούτου, η καθυστέρηση για το DMR είναι μόνο ο χρόνος μεταξύ των σταδίων αποκωδικοποίησης (decode) και ολοκλήρωσης (commit).
- Για το R-SMT, οι ελαφρώς υψηλότερες τιμές αποδίδονται στο slack μεταξύ των 2 νημάτων.
- Τέλος, το ParDet εμφανίζει σταθερά υψηλότερη ελάχιστη και μέση καθυστέρηση ανί-

χνευσης, λόγω του γεγονότος ότι οι βοηθητικοί πυρήνες είναι πιο αργοί (αφού είναι μικρότεροι και σε σειρά) και ο κύριος πυρήνας προοδεύει περισσότερο στην εκτέλεση.

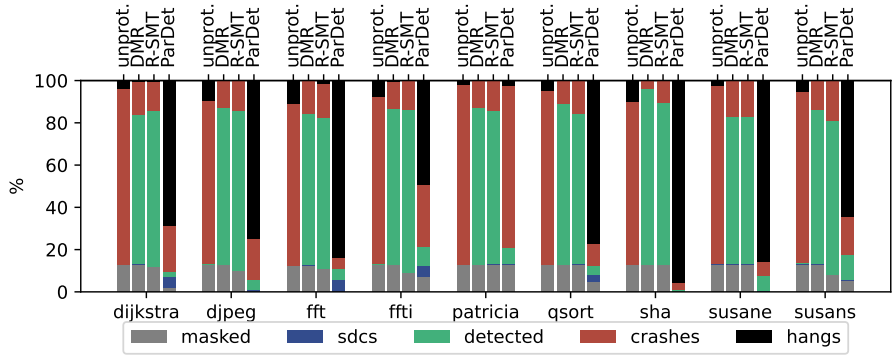
0.7.2 Ανιχνευσιμότητα

Για να αξιολογήσουμε την ανιχνευσιμότητα κάθε μεθόδου, εκτελούμε δύο σειρές πειραμάτων όπως περιγράφεται παραπάνω, μία με μεταβατικά και μία με μόνιμα σφάλματα. Πραγματοποιούμε επίσης τις ίδιες εγχύσεις σφαλμάτων σε ένα σύστημα χωρίς δυνατότητα ανίχνευσης σφαλμάτων, για να το χρησιμοποιήσουμε ως βάση. Από το Σχήμα 3 σχετικά με το πείραμα των μεταβατικών σφαλμάτων αναγνωρίζουμε τα εξής:

- Στο απροστάτευτο σύστημα, παρατηρούμε ότι όλες οι εγχύσεις σφαλμάτων οδηγούν είτε σε crashes, είτε σε masked, είτε σε hangs. Οι masked εκτελέσεις οφείλονται στο γεγονός ότι η έγχυση στον συγκεκριμένο καταχωρητή δεν οδηγεί σε καμία αλλαγή στα αποτελέσματα της εντολής. Για παράδειγμα, αυτό μπορεί να συμβεί εάν η τιμή του επηρεαζόμενου καταχωρητή πανωγράφεται από μια μεταγενέστερη εντολή (όπως μια πράξη AND με 0).
- Το ParDet ανιχνεύει τον μεγαλύτερο αριθμό σφαλμάτων. Αυτό οφείλεται στο γεγονός ότι η υλοποίηση του ParDet για την σύγκριση αρχιτεκτονικών καταστάσεων για την ανίχνευση σφαλμάτων (σε αντίθεση με τη σύγκριση αποτελεσμάτων) χαρακτηρίζει ως ανιχνευμένα σφάλματα τα οποία για τις DMR και R-SMT θα ταξινομούνταν σωστά ως masked.
- Το R-SMT ανιχνεύει περισσότερα σφάλματα σε σύγκριση με το DMR, γεγονός που αποδίδεται στη δομή του πειράματος έγχυσης σφαλμάτων. Για να εξασφαλιστεί μια δίκαιη σύγκριση, σε όλες τις μεθόδους, εγχέουμε τους ίδιους καταχωρητές αφού παρέλθει το ίδιο χρονικό διάστημα (που μετράται από την έναρξη της εκτέλεσης του προγράμματος). Ως αποτέλεσμα, τα ίδια σφάλματα εγχέονται νωρίτερα στη σειρά του προγράμματος για την R-SMT, σε σύγκριση με την DMR. Έτσι, τα σφάλματα που εκδηλώνονται νωρίτερα έχουν μεγαλύτερη πιθανότητα να διαδοθούν σε περισσότερους καταχωρητές μέσω εξαρτήσεων δεδομένων και, κατά συνέπεια, να αλλοιώσουν ενδεχομένως περισσότερες εντολές, και ως εκ τούτου είναι πιθανότερο να εντοπιστούν.
- Παρά το γεγονός ότι το DMR είναι η μέθοδος με το χαμηλότερο ποσοστό εντοπισμένων σφαλμάτων, παρουσιάζει επίσης τις λιγότερες αποτυχίες (crashes και hangs). Αυτό είναι αναμενόμενο, δεδομένου ότι στην DMR, το σφάλμα εντοπίζεται το συντομότερο δυνατό χρόνο, όταν εκδηλώνεται για πρώτη φορά στα αποτελέσματα των εντολών του προγράμματος, δεδομένου ότι, επειδή οι 2 επεξεργαστές εκτελούν πάντα την ίδια εντολή, οποιαδήποτε διαφορά θα εντοπιστεί όταν η πρώτη επηρεαζόμενη εντολή θα ολοκληρωθεί. Αυτό δεν ισχύει στην περίπτωση το R-SMT, όπου, όπως αναλύεται παρακάτω, ένα σφάλμα μπορεί να εντοπιστεί μετά τη ολοκλήρωση ενός αριθμού επηρεαζόμενων εντολών, λόγω του slack μεταξύ των νημάτων. Επομένως, η έγκαιρη ανίχνευση αποτρέπει την περαιτέρω διάδοση των σφαλμάτων στο σύστημα, κάτι που θα μπορούσε να οδηγήσει σε crashes ή hangs. Δηλαδή, η μικρή καθυστέρηση ανίχνευσης του DMR (η οποία επιβεβαιώθηκε και με το προηγούμενο πείραμα) συμβάλλει στα χαμηλά ποσοστά αποτυχίας.
- Ο μικρός αριθμός crashes που εμφανίστηκε σε ορισμένα benchmarks (π.χ. sha) στο DMR, μπορεί να αποδοθεί στη διάδοση του σφάλματος κατά την περίοδο μεταξύ αποκωδικοποίησης και δέσμευσης (ανίχνευση σφάλματος).



Σχήμα 3: Αποτελέσματα της έγχυσης μεταβατικών σφαλμάτων για τις 3 μεθόδους.



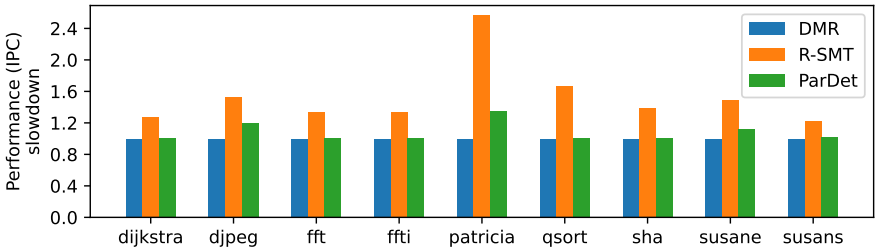
Σχήμα 4: Αποτελέσματα της έγχυσης μόνιμων σφαλμάτων για τις 3 μεθόδους.

Όσον αφορά το πείραμα των μόνιμων σφαλμάτων, από το Σχήμα 4 μπορούμε να συμπεράνουμε:

- Με την παρουσία μόνιμων σφαλμάτων, το απροστάτευτο σύστημα εμφανίζει πολύ περισσότερα crashes σε όλα τα μετροπρογράμματα, επειδή επειδή τα σφάλματα είναι μόνιμα και διαδίδονται περισσότερο στη ροή του προγράμματος σε σύγκριση με τα μεταβατικά, καθιστώντας το πρόγραμμα πιο πιθανό να καταρρεύσει.
- Το υψηλότερο ποσοστό ανίχνευσης που εμφανίζουν τόσο τα συστήματα DMR όσο και τα συστήματα R-SMT σε όλα τα μετροπρογράμματα σε σύγκριση με τις εγχύσεις μεταβατικών σφαλμάτων μπορεί να αποδοθεί και πάλι στον ίδιο λόγο, δηλαδή στα μόνιμα σφάλματα που διαδίδονται περισσότερο στο σύστημα, αλλοιώνουν περισσότερα αποτελέσματα εντολών και συνεπώς είναι πιο πιθανό να ανιχνευθούν.
- Τέλος, το ParDet έχει το μικρότερο ποσοστό εντοπισμένων σφαλμάτων, γεγονός που αποδίδεται στην αυξημένη καθυστέρηση ανίχνευσης.

0.7.3 Επιβάρυνση απόδοσης

- Όσον αφορά την επιβάρυνση απόδοσης (Σχήμα 5) του DMR, μπορούμε να παρατηρήσουμε ότι αυτή είναι η ελάχιστη στις 3 μεθόδους, καθώς διατηρεί ουσιαστικά την ίδια απόδοση με την κανονική (μη προστατευμένη) εκτέλεση. Όπως έχουμε εξηγήσει αυτή η μέθοδος δεν εισάγει επιβράδυνση στους επεξεργαστές.
- Η παράλληλη ανίχνευση έχει (αμελητέα) χαμηλότερες τιμές IPC, λόγω της επίδρασης της αντιγραφής της αρχιτεκτονικής κατάστασης που καθυστερεί τον κύριο πυρήνα.



Σχήμα 5: Επιβάρυνση απόδοσης (slowdown) με τη μετρική IPC για τις 3 μεθόδους.

- Τέλος, το R-SMT έχει τη χειρότερη απόδοση. Μπορούμε να το εξηγήσουμε με βάση τους ακόλουθους 2 λόγους:
  - στο καταμερισμό ορισμένων στοιχείων της μικροαρχιτεκτονικής μεταξύ των νημάτων SMT -ιδιαίτερα της μονάδας ανάκτησης (fetch)- εμποδίζει την απόδοση και δημιουργεί συμφόρηση.
  - στην απώλεια επιδόσεων λόγω των καθυστερήσεων που προκαλούνται από τον γεμάτων απομονωτή συγκρίσεων (comparison buffer)

0.7.4 Κόστος σε υλικό

Στον Πίνακα 1, εκτιμούμε το κόστος σε επιφάνεια κάθε μεθόδου, σε σύγκριση με ένα αprοστάτευτο σύστημα. Υποθέτουμε ότι το DMR αντιγράφει ολόκληρο τον πυρήνα του επεξεργαστή, με αποτέλεσμα 100% επιβάρυνση. Για την R-SMT, η επιφάνεια αυξάνεται λόγω δύο συνιστωσών: του SMT και του απομονωτή σύγκρισης. Ποσοτικοποιούμε την επιπλέον επιφάνεια που απαιτεί το SMT με βάση τη σχετική βιβλιογραφία [13]–[15], τοποθετώντας την σε λιγότερο από 6% της περιοχής του πυρήνα, με βάση πραγματικές σχεδιάσεις. Το κόστος του buffer σύγκρισης (10 καταχωρήσεων), υπολογίζεται σε σύγκριση με την κρυφή μνήμη L1: Ο απομονωτής 10 καταχωρήσεων αντιστοιχεί στο 0,125% της κρυφής μνήμης L1 και συμβάλλει σε πρόσθετη αύξηση της επιφάνειας κατά 0,04% [16]. Για την Παράλληλη Ανίχνευση, η επιφάνεια αυξάνεται τόσο λόγω των βοηθητικών πυρήνων όσο και του αρχείου καταγραφής. Και για τις δύο συνιστώσες επαναχρησιμοποιούμε τα αποτελέσματα από την αρχική δημοσίευση [2], καθώς μοντελοποιούμε επίσης την ίδια μικροαρχιτεκτονική.

Πίνακας 1: Κόστος σε υλικό

Μέθοδος	**Επιβάρυνση σε υλικό ανά εξάρτημα**		**Συνολική επιφάνεια**
	Εξάρτημα	***Επιβάρυνση***	
Unprotected	-	-	1x
DMR	Πλεονάζων πυρήνας	100%	2x
R-SMT	Κόστος του SMT	6%	1.0604x
	Απομονωτής συγκρίσεων (10 θέσεων)	0.04%	
ParDet	Βοηθητικοί πυρήνες (12)	20.2%	1.24x
	Αρχείο καταγραφής (36 KiB)	3.8%	

0.8 Σύνοψη

Στην παρούσα εργασία μελετήσαμε μεθόδους ανοχής σφαλμάτων που αξιοποιούν τεχνικές μικροαρχιτεκτονικής, κατάλληλες για επεξεργαστές σε διαστημικές εφαρμογές. Συγκρίναμε 3 μεθόδους, Dual Modular Redundancy, η οποία χρησιμοποιείται ευρέως σε πραγματικές διαστημικές αποστολές, Redundant Multithreading with Simultaneous Multi-

threading, η οποία χρησιμοποιεί νήματα SMT για να αυξήσει την απόδοση της επανεκτέλεσης, και Parallel Error Detection with Heterogenous Cores, η οποία είναι η πιο σύγχρονη μέθοδος ανοχής σφαλμάτων με μικροαρχιτεκτονική υποστήριξη. Οι 3 μέθοδοι συγκρίθηκαν σε 4 άξονες: i) την ικανότητα ανίχνευσης σφαλμάτων, ii) την καθυστέρηση ανίχνευσης, iii) το κόστος σε υλικό (επιφάνεια), τα οποία είναι πρωταρχικής σημασίας για τις διαστημικές εφαρμογές, και iv) την επιβάρυνση σε απόδοση, το οποίο πιστεύουμε ότι οι αναδυόμενες εξελίξεις στον τομέα του διαστήματος θα καταστήσουν εξίσου σημαντικό τα επόμενα χρόνια. Ανακαλύπτουμε ότι το R-SMT έχει σχεδόν ίση ικανότητα ανίχνευσης και καθυστέρηση με το DMR, ενώ διατηρεί ελάχιστο το επιπλέον κόστος σε υλικό και υποβαθμίζει ελάχιστα την απόδοση του επεξεργαστή. Από την άλλη πλευρά, επαληθεύουμε ότι η Παράλληλη Ανίχνευση παρουσιάζει μικρότερο κόστος σε υλικό αλλά υπολείπεται σε ικανότητα και καθυστέρηση ανίχνευσης, τα οποία είναι μη ελκυστικά χαρακτηριστικά για υπολογιστικά συστήματα στο διάστημα.

1 Introduction

Space is a hostile environment for electronics. Due to the lack of atmosphere and the presence of a plethora of radiation sources, such as cosmic rays or the Sun, spaceborne computer systems are susceptible to failures caused by the natural radiation of the space environment. Reliability and fault tolerance, are of primary importance as computer systems execute mission-critical tasks and any malfunction can turn out dangerous and destructive. From the earliest space programs until today, a variety of fault tolerance methods have been employed and successfully reduce risk and prevent failures. For computer systems, such techniques detect failures, isolate defective components, and restore the correct function of the system. In this work, we focus on methods that secure the Central Processing Unit (CPU) of any computer system in spaceflight. Even if the majority of the conventional fault tolerance techniques for CPUs exhibit high capabilities in detecting radiation-caused errors, they greatly increase the power consumption of the design and the overall chip area. That is because conventional methods usually duplicate large parts of the system to create redundant copies. Energy consumption is a significant constraint for space electronics since space vehicles have limited battery capacity. Chip area is also important since larger designs consume more power and are also more vulnerable because they have a higher probability of getting hit by a radiation particle. For these reasons, approaches of CPU fault tolerance which combine fault detection efficiency with low power and area overheads are desirable.

Computer architecture is a subdiscipline of computer engineering with microarchitecture being the internal organization and implementation of a processor. Both are traditionally offering design solutions that balance different constraints such as performance and throughput, energy consumption, cost, etc. Therefore, novel ideas from that domain can be utilized to optimize fault tolerance approaches and create robust CPUs that also satisfy the tight energy consumption and area requirements. Prior work in computer architecture has indeed already presented such approaches. However, these existing approaches have not been studied enough to be proven suitable for real-world applications in space, since in many cases, fault detection has not been evaluated, whether in other cases, uninformed assumptions about the space environment are being made, or the unique requirements of on-orbit systems are misunderstood.

In this work we seek to bridge the gap between real-world space systems and fault tolerance approaches that exploit microarchitectural trends, showing the latter can increase the reliability of computer systems and reducing energy consumption and area, compared to the conventional methods used in space today.

In this study, we compare 3 methods to demonstrate that claim:

- Dual Modular Redundancy, which is broadly used in space systems, but features high overheads in area and power.
- Redundant Execution with Simultaneous Multithreading [1], which leverages promising techniques from computer architecture but has not been adequately evaluated regarding error detection.
- Parallel Detection with Heterogenous Cores [2], which represents the state of the art regarding performance and area overheads.

We implement and evaluate all techniques in a simulated processor core, extending an open-source simulator that is widespread used in computer-system architecture research. Subsequently, we conduct a comparative assessment of these techniques, with criteria the effectiveness of error mitigation, the latency associated with error detection, the overhead introduced on processor performance and the area overhead introduced to the design, namely criteria aligned with the unique demands of spacecraft systems. For doing so, we design and employ a fault injection campaign, inserting errors into the processor.

We believe our research is timely, as it stands at the intersection of critical technological shifts (increased error rates due to transistor shrinking, penetration of cloud computing in space) and can potentially offer solutions to the current and future challenges in spaceborne computing.

1.1 Project Requirements

While designing this master thesis project, the following requirements and goals had been set:

1. The broader goal of the project is to examine techniques for enhancing the reliability of processors under the effect of radiation in space environments.
2. The methods are desired to be at the microarchitectural level and to exhibit low power overhead to the original design, minimal modifications/extensions to the existing architecture, and small additional area footprint.
3. The criteria on which the analyzed methods should be evaluated are both the impact on the performance of the processor/system and the fault coverage of the design.
4. For this, selected methods will be implemented in a simulator environment.
5. The implementation should allow the measurement of the proposed metrics.
6. More specifically, the implementation should allow error induction in the system under test, to be used for the characterization of the efficiency in error detection and/or correction and the calculation of the relevant metric.
7. The implementation should be close to reality for specific aspects related to the above metrics, producing rational results. Where the designed system diverges from real conditions, sensible and acceptable assumptions should be made.
8. The system on which the proposed methods will be realized in the implementation should also be indicative of the ones used in real space applications/missions.

The rest of the text is organized as follows: In Chapter 2, we describe typical applications of computing systems in space missions, along with the unique requirements the space environment imposes on such systems. We also present examples from real-world systems and fault tolerance methods used in recent space programs. In Chapter 3, the effect of radiation on electronics is analyzed. We describe the sources of radiation in space, the effects of radiation on electronics, and the interaction mechanisms of radiation particles with digital systems, specifically SRAM devices, DRAM devices, and logic. In Chapter 4 we give an overview and background knowledge of computer architecture concepts useful for the understanding of the remaining text and necessary for the explanation and analysis of the microarchitectural techniques we will be implementing. In Chapter 5 we present and discuss the 3 fault tolerance methods implemented, evaluated, and compared in this study. Chapter 6 describes the simulator tool used in this study, the design choices and implementation of the studied fault tolerance techniques and the implementation of the fault injection. In Chapter 7 we present the methodology behind the conducted evaluation experiments, the obtained results, and the conclusions drawn. Finally, in Chapter 8 we give some conclusions and ideas for future research directions.

2 Spaceborne Computer Systems

In this chapter, we discuss computer systems operating in spaceflight missions. We start by describing the tasks and applications computer systems perform inside a spacecraft. Next, we analyze the requirements of these systems emphasizing their differences from terrestrial computers, mainly the need for fault tolerance. We continue by describing key technologies used in space computing systems to highlight that the need for energy efficiency conflicts with systems of high computing capabilities. Based on that, we claim that in the future, more performant systems will be in increasing demand and hence solutions that combine low power consumption with performance are important. Last, we give real-world examples of fault mitigation techniques used in space and avionics programs.

2.1 Applications of Onboard Computing

In 1961, the Mercury spacecraft of the first manned spaceflight program of the United States, operated without any on-board computer [3]. Computer mainframes on Earth performed all the necessary calculations for orbit control and re-entry, which were then transmitted over radio to the spacecraft. The same thing was also true for the first 15 years of unmanned space exploration as well as for the Soviet Union's countereffortts. However, 8 years later, in 1969, the moon landing would have been inevitable without the use of onboard computer systems. As a matter of fact, the computer system prototypes for the Apollo missions from 1962 to 1968 consumed two-thirds of the global supply of integrated circuits [4]. Nowadays, computers are integral components of all spacecraft and on-orbit systems, part of every mission subsystem, and support all operations.

More specifically, some examples [17], [18] of such subsystems are:

- *Altitude and orbit control*, with computer systems acting as intelligent controllers in the navigation subsystems, calculating and adjusting the trajectory in both nominal flight conditions and non-nominal cases (such as emergency situations). Such control tasks are math-intensive, require high numerical accuracy in the produced results, and impose tight timing constraints.
- *Telecommands execution*, that is remote system control and monitoring, allowing the remote issue of commands from the mission control stations on Earth.
- *Data Acquisition and Telemetry*, by gathering the required data for determining the vehicle's position, formatting and compressing data in order to reduce the downlink bandwidth and transmitting them to the ground stations or saving them on local storage. Since the collected data needs to be manipulated and compressed before transmission, hardware systems for telemetry often include specific architectures such as signal processors, hardware accelerators, etc.
- *Time synchronization*, for ensuring that all spacecraft subsystems are synchronized with respect to Universal Time Coordinated (UTC). This is crucial for orbit determination and continuous knowledge of position as well as high timing precision.
- *Failure detection*, revealing faults or anomalies in subsystems by implementing error detection mechanisms.
- *Isolation and Recovery*, isolating faulty components, recovering from failures, and maintaining sound functionality.

- *Power Monitoring*, securing reliable, ongoing power supply to all subsystem modules, coordinating the storage, distribution, and conversion of power in the spacecraft, and orchestrating the transition between the various reserve power sources (i.e. electrochemical/batteries and solar generated)
- *Thermal control*, measuring the characteristics of the outer environment and maintaining all components within acceptable operational temperature margins.
- *Payload.* Computer systems are present also in the payload, in the management of scientific experiments, i.e. for processing collected data from scientific instruments. The cases of high-rate payloads, such as in the imaging of communication applications, impose the need for higher bandwidth data busses and also increase the requirements of computer systems performance and usually include complex operations such as Fast Fourier Transforms or image processing.

Lastly, it is worth noting that recent trends such as cloud or edge computing are penetrating into satellite systems [5], especially given the increasing recent deployments of satellite constellations. These computing paradigms, enable computation closer to data sources, and as a result, can significantly reduce the requirements of downlink bandwidth and increase availability. However, they alter the scenery of orbital computer system requirements, since require systems with higher processing capabilities. As a result, we believe that this work, which examines fault tolerance while also taking into account other requirements such as performance, is timely since spaceborne computer systems are becoming increasingly complex and potent.

2.2 Spaceborne Computer Systems Requirements

Spaceborne computing systems have a set of unique requirements due to both the nature of the space environment and the criticality and cost of space exploration. Extraterrestrial environments are harsh not only for electronics and computing systems but for most components of a spacecraft. Space-grade technology needs to withstand extreme forces and vibrational sock, a wide range of temperatures ranging from decades below zero Celsius to hundreds of Celsius, and radiation. Additionally, systems need to operate by consuming minimal power, since energy sources in space are limited: the cost and weight of batteries make impossible the abundance of stored energy and solar energy might not always be available, especially in planetary exploration where clouds or dust particles might shield solar panels from solar rays. On top of all of these, for a lot of interplanetary or space exploration programs, missions take years to reach their destination. This requires the long-term operation of all subsystems which need to function correctly and autonomously for long periods of time, unattended, without human intervention, supervision, or frequent attention from the ground stations, and without the capability of extended repairs. All these constraints are present also in the design and operation of spaceborne computer systems. It is clear that availability and reliability are the two fundamental properties computer systems in space need to satisfy, combined with the cost of manufacturing and design, testing capability, and development time. The central part of the design of computing systems that will operate in space is the minimization of risk. Along with this, systems must be able to detect failures quickly. Additionally, electronics need to have low mass and volume and consume minimal power. Especially given the recent trend of electronic components shrinking, space technology shifts towards more dense and integrated systems, reducing power even further. [19]

When it comes to minimisation of risk, robust systems are a priority. Electronic systems need to be fault tolerant against failures not only attributed to radiation but also i.e. due

to extreme vibration (important for Printed Circuit Boards), hardware malfunctions due to component aging, and software bugs. For example, in one of the training simulations for the flight of NASA's Space Shuttle, all 4 redundant flight controllers simultaneously become unresponsive because of a software bug (involving the use of a GOTO instruction) [20].

Radiation is nevertheless one of the main sources of failures. Systems in space are not protected by the atmosphere and being exposed to radiation is vulnerable to radiation-induced errors. In this work, we are focusing on errors solely caused by radiation on computer processors. However, space is not a homogenous environment with respect to radiation. Radiation environment is defined by the orbit of the spacecraft.

2.3 Technologies used in Space Computer Systems

All the embedded computer systems used in space, are real-time systems, under power and timing constraints. Real-Time Operating Systems are being used to schedule the various tasks for execution into the CPU and secondarily to run diagnostic tests to verify the correct function of the system and manage memory fault detection and correction.

Firmware -namely software loaded in Read Only Memories (ROMs)- is typically used for critical processes, since it is not susceptible to hard faults, since ROMs are more robust and less vulnerable. Along with firmware, ROMs might also store subsystem parameters but have the drawback that cannot be modified after launch.

Regarding hardware selection, space agencies often seek and use components behind the state-of-the-art time of flight. The reason behind that choice is that proven equipment and mature techniques are more thoroughly tested and safe. The same goes for design simplicity since complex designs introduce a greater risk of errors or bugs. Regarding processors, all CPUs used by NASA by 2012 [21] are simpler designs and hence less performant compared to the ones used at that time in high-performance systems or even desktop/mobile systems on Earth. Nonetheless, there are various computationally intensive operations in both manned and unmanned space applications that require potent CPUs, such as systems with high fidelity or autonomous operation, especially when it involves rapid transitions between orbits and travel to distant targets. Accurate altitude knowledge and control demands accurate sensors with complex drivers which in turn also demand potent CPUs. To give an example, to satisfy the high calculating capabilities needed for the Space Shuttle operations, the computers of the Space Shuttle were extremely energy-consuming compared i.e. to the power supply of a deep space probe [3].

To conclude, in conventional spaceborne CPUs, performance requirements are increasing, both due to the penetration of cloud computing in space as we showed before, as well as due to inherently computationally demanding tasks. There exists a tradeoff between performance (which requires more power) and energy constraints (which aim to reduce power consumption), thus this is another reason we find this study current.

Lastly, during the last few years, there has been a shift in the space industry from custom parts designed specifically for space to commercial products. Commercial-Of-The-Self-Components (COTS) are appealing since require less testing, are cheaper, and cut down time-to-market nearly to zero. Are, however, more vulnerable to radiation errors.

2.4 Real-World Examples of Fault Tolerance in Space Missions

In 1961, rocket-borne processors had an average Mean-Time-to-Failure of just 15 hours and by the flight of the Apollo program, this has been increased by over to orders of magnitude. [22]. It has been made clear that computer systems play a critical role in the success of modern space missions, enabling and ensuring proper function, data collection and transmission, and the execution of the scientific experiments of each mission. For these reasons, fault tolerance is of great importance.

At the system architecture level, space programs had been using either redundancy or distribution of processing, to mitigate faults. Redundancy replicates the computations whether distributed processing alienates single points of failure. Usually, for flight-critical components, redundancy is preferred [18]. Other methods of fault mitigation are watchdog timers with which system hangs are identified, power cycling, or (for hard faults) switching to redundant hardware. Memory is usually protected by Error Detection And Correction codes (EDAC), such as Hamming. Nevertheless, this is not a panacea, since too long scrubbing times leave the system vulnerable to errors, whether short ones consume extensive processing time.

At the hardware level, radiation hardening has been the method of choice. Radiation-hardened electronics are manufactured with processes that make them resistant to radiation-induced malfunctions. However, during the last years, an increasing number of radiation-hardened manufacturers are ceasing operations [23] which makes other approaches like the ones evaluated in this work much needed.

2.5 Summary

This chapter centered around spaceborne computing systems. We described the applications of such systems, present in every spacecraft component, from telemetry to payload experiments or power management. We tried to highlight the vastly different requirements space computer systems have, compared to systems operating on Earth, with most importantly the need for high reliability and safety. We also referred to the computing technologies used in space, that is real-time embedded systems often of previous generation, but with increasing demands for performance. Finally, we presented real examples of fault-tolerant computer systems from space and avionics flights.

3 Radiation-induced Errors on Electronics

One of the most notable differences between Earth and space is the higher radiation levels present in the latter. Although Earth is protected by the atmosphere and magnetic field and thus maintains the low radiation conditions necessary for human life, the same does not hold for extraterrestrial environments, where radiation poses a significant risk both for astronauts traveling in manned missions, as well as for spacecraft equipment and instruments.

In this chapter, we give an introduction to the origins of radiation in space and the effect it has on electronics. We structure the following analysis by differentiating between i) the sources of radiation in space, ii) the effects on electronics, and iii) the underlying mechanisms in the interaction of radiation particles with digital circuits specifically.

3.1 Sources of Radiation in Space

Radiation in space originates from 3 sources: Galactic cosmic rays, Trapped radiation, and Solar particle events.

3.1.1 Galactic Cosmic Rays

Galactic cosmic rays (GCRs) are charged particles originating outside of our solar system, traveling through space with speeds close to the speed of light. The Austrian astrophysicist Victor Hess was honored with the Nobel prize in 1936 for their discovery with high altitude balloon experiments in 1912 [24] and today it has been established that cosmic rays are emitted during supernovae eruptions within our galaxy, with the main evidence supporting this being their particle composition. GCRs consist of electrons and ionized nuclei with the latter of a mix of 90% protons (hydrogen nuclei), 9% alpha particles (helium nuclei), and 1% heavier nuclei [25]. Within that 1% (heavier particles with $Z > 2$), certain elements are found in abundance in cosmic rays but are absent in solar system material (and vice-versa). Moreover, other groups of elements that are present in the particle mix are considered to be the product of collisions with the interstellar matter [26]. Those 2 observations confirm that GSRs have traveled through the interstellar medium before reaching our solar system and are not emitted from bodies within that.

Regarding the radiation hazard of GCRs, it is worth noting that the nuclei heavier than He, have a significant impact on both astronauts and instrumentation equipment [27], despite their scarcity in the particle mix, because these particles occur typically with higher energies.

Lastly, as all charged particles, GCRs are affected by magnetic fields. This means that they are affected both by solar activity as well as Earth's magnetic field. Due to both the interplanetary magnetic field produced by the Sun's activity and the solar wind, GCR flux is varying with an anti-correlation with solar activity. Moreover, flux intensity increases with distance from the Sun, since the magnetic field is preventing from rays penetrating the inner solar system. From a mission design perspective, this means that spacecraft in the outer solar system or during low solar activity periods, are more exposed to galactic cosmic radiation and gives a first hint on the fact that radiation in space fluctuates both depending on the region and time, something which is of crucial importance and will become clearer in the rest of the discussion too.

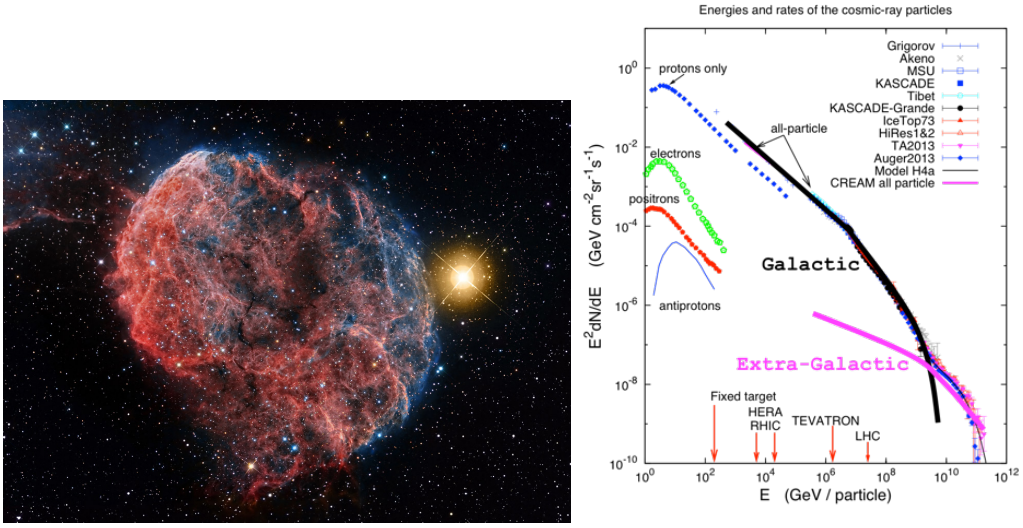


Figure 3.1: On the left, IC-443 the remnant nebula of a supernova erupted 3.000-30.000 years ago. On the right, the energy spectrum of cosmic ray data is gathered by the IceCube neutrino observatory, located deep in the South Pole ice.

3.1.2 Trapped Radiation

By the term trapped radiation, we are referring to the radiation inside the Van Allen belts, which are zones appearing around planets (in our solar system around Earth and Jupiter), consisting of charged particles being captured by the planet's magnetic field. Discovered by James Van Allen in 1958 using the Explorer satellites [28], it is today well understood that Van Allen belts are shielding Earth's atmosphere, preventing it from destruction by impacting particles, as well as protecting also the terrestrial environment from radiation, by deflecting charged particles across the magnetic field lines of belts. At the same time, since Van Allen belts surround Earth and every space vehicle needs to pass through them to get into orbit or travel in the solar system, they consist a major hazard for spacecraft, instrumentation, and astronauts, given the presence of high radiation levels. For example, for an unprotected human, the radiation dose they would be exposed traversing through the Van Allen belts is higher up to an order of magnitude than the full body dose that is considered lethal [29], however in all missions, shielding practices are employed resulting in safe radiation conditions and bringing down the radiation doses to acceptable levels.

Earth's Van Allen belts form 2 distinct regions: the inner and the outer belt [30], both consisting of trapped particles captured by Earth's magnetic field, but of different origin. The inner belt is composed mainly of protons and electrons, which are the results of collisions of neutrons from cosmic rays with particles of the outer atmosphere and are decayed into protons and electrons while bouncing back. The outer belt is composed mainly of helium ions, protons, and electrons originating from the solar wind but with substantially lower energies, which do not allow further penetration into the inner belt or the atmosphere.

3.1.3 Solar Particle Events

The Sun contributes to a significant extent to the radiation emission within the solar system, through 2 different effects: solar wind and solar flares. The solar wind is a natural phenomenon taking place in the Sun's outer atmosphere (called corona), where charged particles are projecting away from the Sun. The cause of solar wind is heated plasma from the corona layer which is energized to the extent it cannot be contained from the

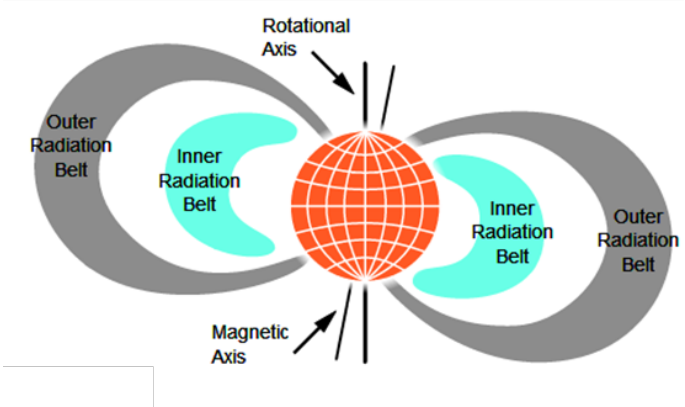


Figure 3.2: Diagram with the outer and inner Van Allen Belts

gravitational field of the Sun, and eventually erupts, and travels along the Sun's magnetic field lines. This creates a stream of low-energy electrons and protons, which traverse the solar system. Since they are of low energy, they pose a significant threat only to the components mounted on the exterior of the spacecraft. Solar flares on the other hand are more intense disruptions of the Sun's upper layers (mainly the photosphere) which emit highly energised protons and heavy ions.

3.2 Radiation Effects on Electronics

Particles from all these radiation sources when interacting with electronics cause effects which can be categorized as follows:

3.2.1 Total Ionizing Dose

Total ionizing dose refers to long-term radiation effects caused by energy transferred through ionization only, from the incident particles to the electronic device. It is hence a cumulative effect that in an atomic level moves electrons into higher energy states. As a result, macroscopically, this can cause increased leakage currents or threshold shifts in the affected devices.

3.2.2 Displacement Dose

Displacement dose is also a cumulative effect of radiation in which the highly energetic incident particles are not exciting the ions of the device -as in the case of TID- but this time are displacing whole atoms of the material, creating vacancies in the semiconductor crystalline lattice. This results in changed conducting properties of the semiconductor and leads, from a device operational perspective, to increased leakage currents or decreased conduction. For example, under long-term radiation exposure (and hence due to DD effects), diodes are becoming less conductive, and solar cells (which are essentially a diode) are becoming less efficient, again for the same reasons [31].

3.2.3 Single Event Effects

Under the term Single Event Effects, we combine all the effects caused by the flow of striking particles through an electronic device. Regardless of how these effects are explained with particle interactions at an atomic level, their effects can range from having no impact at all, to transient disruptions in circuits, or permanent damage to the IC device. It is really important and interesting that SEE appears not only in devices in space due to the presence of extra-terrestrial radiation but also in terrestrial ones, due to the higher transistor densities, which make devices more susceptible to these kinds of effects. Among the

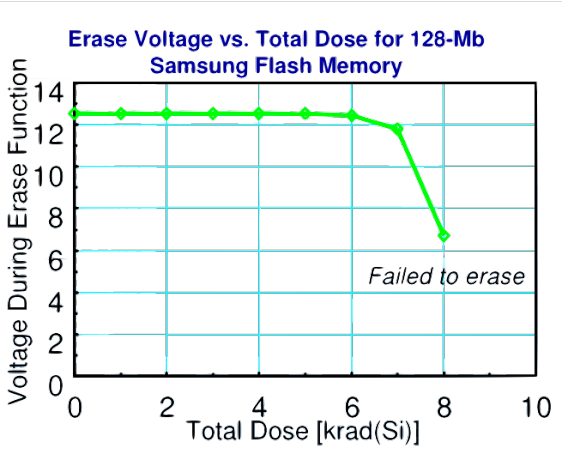


Figure 3.3: Effect of TID on a flash memory device.

various SEEs, Single Event Upsets are transient errors that appear as transient pulses in combinatorial circuits or as bitflips in memory elements.

3.3 Interaction Mechanisms

There are two technologies broadly used in storage systems: Dynamic Random Access Memory (DRAM) and Static Random Access Memory (SRAM). We will briefly analyze how radiation, and specifically Single Event Effects, are interacting with each of these types of memory cells in addition to digital logic circuits.

3.3.1 DRAMs

DRAM is usually used in main memory since it is cheaper but also slower. The simplest DRAM cell design consists of a capacitor and a transistor and is holding 1 bit of information in the charge of the capacitor. However, due to leakage current of the memory cell's transistor, the capacitor is discharging, and therefore DRAM cells need to be refreshed in order to retain their data. This -namely the fact that the information retention in DRAM cells is passive- has two implications for the behavior of DRAM devices under SEEs. Firstly, this means that there is no inherent mechanism in the operation of the memory cell which could lead to auto-correction of some SEE (as can happen in some cases with SRAM). Secondly, because again, the signal is stored in the charge of the cell's capacitor, a SEU can appear not only due to a transition from one stable state to another but also due to the degradation of the stored signal outside of some noise margins [6].

3.3.2 SRAMs

SRAM-type memory is typically used in caches as well as in most microarchitectural storage units (such as the architectural register file), due to the advantage of being faster than DRAM. Its cell (also known as 6T cell) is composed of 2 cross-coupled transistor inverters which form a feedback path, enabling data to persist on memory without the need of refreshing, as long as power is supplied. From this perspective, the key difference with DRAM cells is that in the SRAM cell, binary information is encoded not by holding a charge (as in the DRAM cell's capacitor) but by switching the current direction in the coupled transistor bridges.

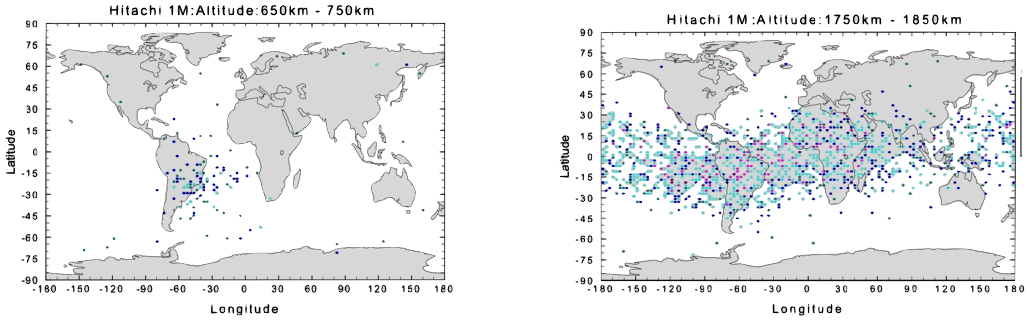


Figure 3.4: SRAM failure rates at two different altitudes inside the Van Allen zones. In lower altitudes (left diagram) most upsets occur over the south Atlantic, in the South Atlantic Anomaly (SAA). SAA is a region where the Earth's magnetic field is stronger (due to the misalignment between the magnetic and rotational axis) and their ionizing radiation is increased, penetrating into lower altitudes. From NASA APEX experiment.

The susceptibility of SRAM devices in SEEs depends heavily on the strike location of the particle within the cell circuit, that is the specific transistor of the 6T cell (actually whether it is turned on or off at the time of the strike) [7] and in general, radiation particles will accumulate charge in one of the struck transistor's terminals causing, in turn, current flow between the 2 transistors of the semibridge, leading to a voltage transient. This voltage transient because it typically occurs in the drain of the transistor can act as a write pulse, changing the written value of the affected storage cell. Also, faster SRAMs are more vulnerable to these transient errors, something which has serious implications for future technologies. Lastly, it is worth noting that since most reconfigurable systems such as FPGAs are based on SRAM technology, and given their recent wide adoption in spaceborne applications, the study of SEE interaction with SRAMs is of significant importance.

3.3.3 Logic circuits

For combinatorial logic circuits, (transient) errors caused by SEEs are manifested when a voltage transient from a signal line is being captured in some latch [7]. For this to happen the following 2 conditions must be met: a radiation particle of sufficient energy must hit an active line of the circuit, and an active path must exist from the staked line to some latch though other lines and/or circuit elements (such as gates, etc). The particle must transfer enough energy via the interaction with the device's substrate, in order to generate a voltage transient which will exceed the error margins of all the circuit elements between the struck location and the latch. This way the logic value of the latch is going to be altered and the transient error is going to manifest into the circuit.

Regarding sequential circuits, because flip flops and latches operate similarly to SRAM cells, the same principles apply, with the difference that usually larger transistor sizes are being used for the former [32], attributing to higher robustness compared to SRAMs.

For these reasons, in this study we will model all errors in processor logic as bitflips in latch elements.

3.4 Summary

In this chapter, we dived into the natural radiation present in space, with emphasis on the interaction with electronics leading to faults. We briefly described that radiation originates

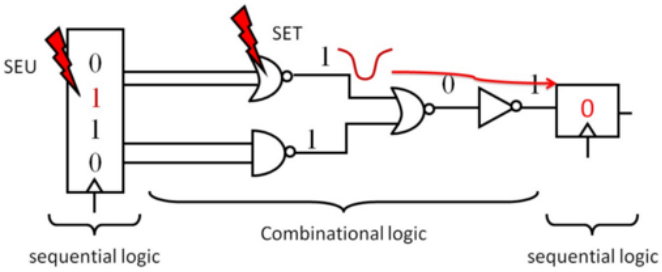


Figure 3.5: Propagation of a single event upset transient pulse to a latch element in a circuit.

from outside the solar system as cosmic rays, from interactions with Earth's magnetic field and atmosphere, and from the Sun. We tried to highlight that each of these sources is more prominent in different regions of space and hence, regarding where a mission is going to operate, different hardening measures must be employed, depending on the radiation level of that region. We discussed the effects it can have in electronics, both cumulative such as Total Ionization Dose and Displacement Dose as well as transient as Single Event Effects. For the latter, we analyzed the interaction processes with digital electronics -DRAM and SRAM memories and combinatorial and sequential circuits- where in both DRAM and SRAM technologies radiation can alter directly the stored bits, whether in logical circuits transient errors are manifesting by propagating to some latch element.

4 Computer Architecture Primer

This chapter presents the essential background information on computer architecture and microarchitecture, necessary for the understanding of the rest of the text.

A key topic in this study is microarchitecture. The architecture of a processor describes it's function, defining the interaction between hardware and software by specifying the Instruction Set Architecture (ISA), registers set, execution and exception and memory models, etc. Microarchitecture on the other hand studies the design and implementation of a processor by describing among others, the pipeline and the various pipeline stages or the cache hierarchy. One of the most important design objectives in the development of new microarchitectures is processor performance since research and design of new processors aim to improve performance and efficiency. For this reason, during the last decades, a wide range of techniques has been developed, aiming at optimizing the performance of processor pipelines. Some of them are being exploited by the fault tolerance techniques of this study and we are going to present them here.

4.1 Superscalar and out-of-order pipelines

4.1.1 Superscalar processors

An important property that greatly impacts performance is the average instruction throughput of the processor, which can be quantified by the metric of Instructions Per Cycle (IPC). Pipelines that can fetch and issue only 1 instruction per cycle can achieve the best possible throughput of 1 instruction per cycle (IPC<=1). To further increase performance, superscalar processors have been designed, which have IPC>1. Superscalar processors are able to execute more than 1 instruction at the same time in all pipeline stages by simultaneously advancing multiple instructions through the pipeline. The exact number of instructions being able to be processed simultaneously by a pipeline stage is called width of that stage.

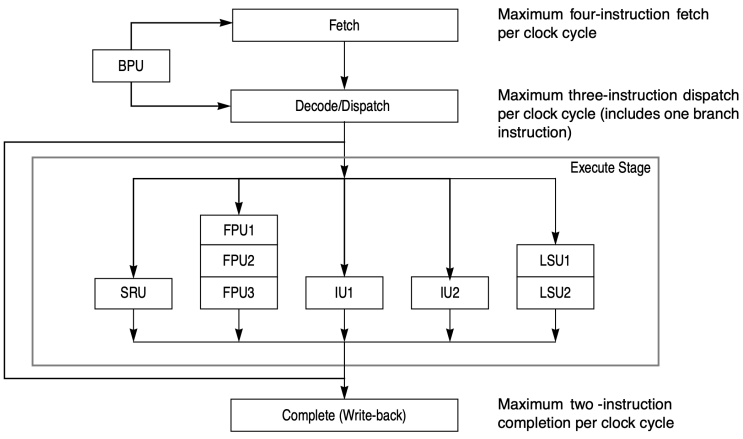


Figure 4.1: Pipeline schematic of the superscalar IBM PowerPC 750 processor, showing the widths of various pipeline stages and multiple functional units. A hardened version of that CPU was used in NASA's Curiosity rover [33].

4.1.2 Dynamic scheduling

Even with superscalar CPUs, pipeline performance can be hindered by frequent pipeline stalls. With in-order pipelines that execute instructions according to the original program

order (static scheduling), if an instruction causes the pipeline to stall, all subsequent instructions need to also stall and cannot proceed, even if they could. Out-of-order CPUs can execute instructions in a different order than the one they appear in the program (dynamic scheduling), being issued as soon as source operants become available and a suitable functional unit is free. This way, subsequent instructions can bypass a stalled instruction; dynamic scheduling creates parallelism not available at compile time (Instruction Level Parallelism (ILP)). However, in out-of-order pipelines, fetch and commit of instructions happen in-order. For accommodating in-order commit, a hardware unit called Reorder Buffer (ROB) is used. The details of dynamic scheduling implementation are out of the scope of this introduction, but the ROB assists in maintaining correct dependencies between the in-flight instructions, for example by providing instruction results as the operants to other instructions before the producer instructions are committed and written to the register file.

4.2 Branch Hazards, prediction & speculation

As we saw, in dynamically scheduled processors instruction fetch happens in-order. Therefore the throughput of the whole pipeline is restricted by the fetch bandwidth. To sustain maximum fetch throughput fetch must happen continuously and from sequential locations in the program memory (to avoid Icache accesses and misses). However, this is not possible when the next instruction is not known beforehand, such as in the case of control flow (branch) instructions which potentially alter the next fetch address. For unconditional branches, the next fetch address will be calculated in the decode stage with the decode target address, and for conditional branches in the execute stage, with the evaluation of the branch condition. This imposes stalling the pipeline in each control instruction and again negatively impacts performance. To circumvent this, a processor-specific mechanism implemented in hardware is employed, called branch prediction.

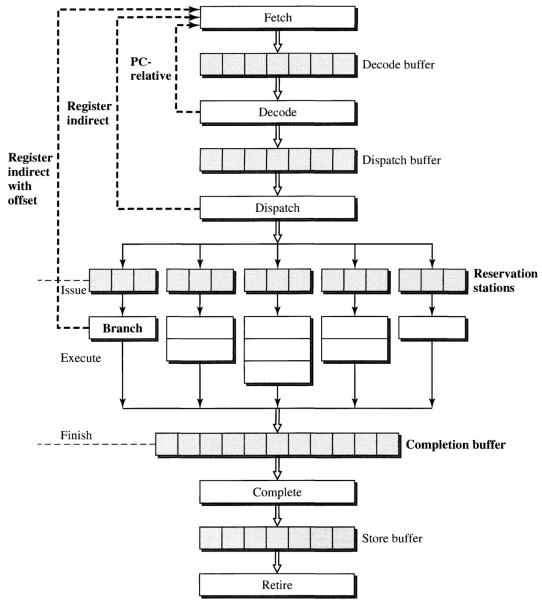


Figure 4.2: Penalty cycles (cycles spend stalling the pipeline) in the calculation of branch target address for different branch instructions.

4.2.1 Branch prediction & speculation

Branch prediction allows to minimize branch penalty and maximize instruction flow throughput. When a control flow instruction is fetched, both the branch target addresses and

branch conditions are speculated, based on an algorithm implemented in hardware. After the branch direction is predicted (taken or not taken), fetch proceeds down the predicted path (which is called branch speculation). According to branch speculation, after predicting the branch direction the execution continues along the predicted control flow path. Until the actual direction of the branch is resolved, instructions are restrained from updating the memory or the register files by disallowing commit. If upon the validation of branch direction, the prediction is found correct, the speculated instructions are committed. Otherwise, if the branch predictor mispredicted the direction of that branch, all the speculatively issued instructions need to be removed from the pipeline and the execution needs to be restarted from the mispredicted branch, following the opposite (correct) path.

4.2.2 Squashed instructions

Squashed instructions are instructions removed (flushed) from the pipeline and ROB without completing and without being committed. This can happen due to various reasons:

- Upon the discovery that a branch was misspeculated, the wrong path instructions are flushed.
- Due to a violation of the memory order during speculative memory order disambiguation, the dependent load is flushed and re-issued.
- When a trap (software exception) is raised, all the subsequent speculatively issued instructions are flushed.

4.3 Simultaneous Multithreading

Even with all these optimizations, there are cases where not all the available ILP is utilized. That is, some functional units are idling because the instruction mix of the program contains a lot of instructions of the same type or because instruction dependencies prohibit out-of-order execution. One solution to this is Simultaneous Multithreading (SMT) [8], a technique where the processor executes simultaneously more than one thread in a single core, utilizing this way all the available functional units and allows to achieve higher processor throughput without significantly impacting the performance of single-threaded applications. The most important takeaway is that by utilizing this spare (wasted) ILP, in many cases the execution of multiple threads with SMT can be peformed faster and with higher throughput compared to the execution of the same threads alone sequentially.

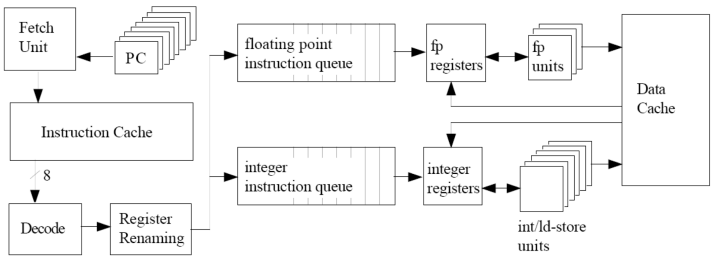


Figure 4.3: Pipeline of a simultaneously multithreaded processor, with the units being accessed by multiple threads duplicated.

SMT can be implemented by either partitioning or duplicating all the processor resources that are required to be shared by all the threads. Each thread is a separate instruction stream and the scheduling and mixing of instructions is managed by the hardware. Because duplicating the fetch stage would require also the capability for parallel access to the instruction caches, usually the fetch stage is time-shared (partitioned) between the

threads. However, this has the potential of creating a bottleneck, since it reduces the fetch bandwidth of each thread and places higher demand on specific structures such as the Translation Lookaside Buffers (TLBs), the branch predictor, the functional units, and the register files. Thankfully, the choice of thread arbitration/scheduling mechanism can mitigate that.

4.4 Summary

In this chapter, we presented the essential concepts for the comprehension of the rest of the text by explaining key computer architecture and microarchitecture concepts. We examined topics like superscalar and out-of-order processors which aim at increasing pipeline performance and are being used by the microarchitectural fault tolerance techniques which we will be analyzing in the next chapter. We showed how control flow instructions degrade performance and how prediction and speculation minimize branch penalties, concepts one of the studied fault tolerance methods will build further and which will be also needed in the implementation. Lastly, we introduced Simultaneous Multithreading, a method that maximizes processor throughput and enables the execution of different independent threads in the same processor core. SMT is also the key concept behind one of the 3 methods analyzed in the next chapter.

5 Description of the Studied Error Detection Techniques

This chapter describes the 3 fault tolerance methods studied in this work. Among these 3, the first method has been widely used in a plethora of real space missions and while it exhibits the best ability to detect errors, it demands extensively large chip area and power consumption. The other 2 methods have been proposed by computer systems researchers and even if they have not been tested in space operations, theoretically improve power and area overheads, while modestly degrading the system performance, through the use of microarchitectural novelties from computer architecture research. The error detection capability of the latter two is one of the anticipated results of this study.

Fault tolerance techniques in computing systems can be typically categorized as software-based or hardware-based, depending on the level the fault mitigation mechanism operates on. At the same time, for the subset of techniques that rely on computation redundancy for the mitigation of faults, redundancy can be achieved either spatially, by duplicating computation in multiple parallel units, or temporally, by repeating computation in time. Additionally, fault mitigation might be limited only to the detection of a system fault or include error correction with the restoration of system operation.

In this study, we focus only on error detection with hardware-based techniques. Also, we occupy ourselves with methods that secure only the processor logic and pipeline, assuming that memory is protected by other methods such as Error Correcting Codes (ECC).

5.1 Dual Modular Redundancy

Dual Modular Redundancy (DMR) is a spatial redundancy technique where each hardware component is replicated twice and computation is repeated in both copies of the system, to increase reliability. It is also the first method we will study in this work. For a computer system, redundant computation is hence performed not by running multiple repetitions of the program, but by forming a system with 2 processors instead of 1 and executing the program on each. With this scheme, errors are detected by comparing the results of the execution from the 2 processors. Memory needs to be either also duplicated, or protected, i.e. with ECC.

DMR in general is a technique that transfuses high robustness to the hardened system, as any radiation-induced error will strike only one of the two processors and hence change the results of only the one, something that will be subsequently detected. We make the (reasonable) assumption that the probability of two particles striking and producing an error in both processors (allowing an error to remain undetected and produce a system failure) is negligible. However, DMR systems have 2 important drawbacks. Firstly, DMR requires doubling all hardware components, introducing a two-times area overhead. Increased area is particularly unappealing in all hardware systems because it is accompanied by increased energy consumption as well as increased manufacturing/packaging costs. Especially in the case of radiation tolerant systems, increasing the area of the design entails also raising the probability of error occurrence. That is because designs that take up more area have a higher probability of being struck by radiation particles, compared to designs of smaller surface. Secondly, DMR systems need a voter component

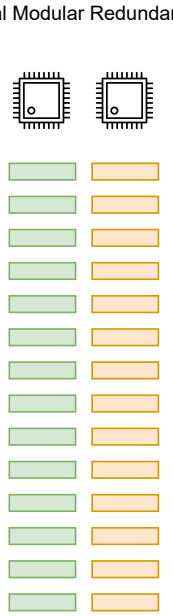


Figure 5.1: Schematic representation of Dual Modular Redundancy.

that detects errors by comparing the results from the duplicated modules (i.e. CPUs). By definition, this component needs to be unique and is thus vulnerable and unprotected. As a result, the manifestation of an error in the voter can lead to system failure and for this to be mitigated, radiation-hardened components need to be used in that particular module, raising the manufacturing costs and/or prohibiting the construction of DMR systems from solely COTS components. The last point is of great importance since as it has been previously mentioned, manufacturers of radiation-hardened components are lessening.

5.2 Redundant Multithreading

Redundant multithreading [9]–[11] is a class of time redundancy fault-tolerant techniques, where redundant execution is happening in the architectural level and realized with different threads running on the processor. With redundant multithreading, each computation is repeated not i.e. in separate CPU cores (such as in the case of the spatial DMR) but on different program threads on the same CPU core, each of them performing the same computations redundantly and then validating that all executions produced the same results, hence no errors had occurred.

The variation of redundant multithreading we are going to focus on is called redundant multithreading via Simultaneous Multithreading (SMT) [1] (denoted as R-SMT). This approach utilizes the microarchitectural technique of SMT in order for redundant multithreading to be performed efficiently, by spawning 2 SMT threads and assigning on each the execution of a copy of the same program. In that scheme, the first thread (primary thread) executes the program's instructions while the second (redundant thread) re-executes them and validates the results. It is useful to maintain the one thread slightly further along in its execution than the other, creating a leading thread and a trailing thread. This, enables a number of performance optimizations, all centering around the idea that since both threads are executing the same instructions and expect to form the same results, the leading primary thread can assist in the execution of the trailing redundant one, allowing the former (redundant) to be executed more efficiently, without repeating unnecessary work already completed by the latter (primary).

Figure 5.2: Schematic representation of Redundant Multithreading.

Some performance optimizations enabled by the primary leading and redundant trailing are the following:

Prefetching. One first optimization where the primary thread can foster the execution of the redunand one is in the case of cache misses, where essentially primary acts as a prefetcher for the redundant thread. When the primary thread encounters a cache miss, data (or instructions) are being fetched from the main memory and loaded into the on-chip caches. By having the redundant thread lagging behind in the execution when the corresponding load instruction is being re-executed, the data will already be present in the cache, avoiding the need for stalling the thread and waiting again for the same transfer from main memory.

Branch prediction. A second mechanism that can be improved and assisted by redundant threading is the branch prediction of the redundant thread. In pipelined processors, branch instructions pose delays, since their outcome (or direction, meaning whether the branch is taken or not) is not computed until the execute stage (several cycles later) and consequently, until then, the next instruction that needs to be fetched in the immediate

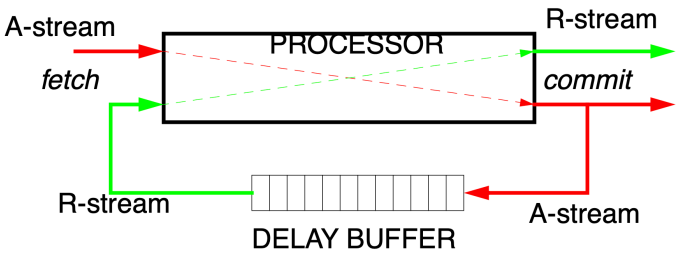


Figure 5.3: Delay Buffer, being filled from A-thread on commit with branch and operant predictions and popped by R-thread of fetch.

cycle, remains unknown. To circumvent this, a technique called branch prediction has been developed. According to that, one of two possible branch directions is being chosen (predicted) and the execution continues according to that. When the real branch direction is calculated in the execute stage, if the prediction was wrong, the execution needs to be restarted from that branch. This introduces a performance penalty in the cases of branch mispredictions. In the case of primary and redundant threads, the redundant thread is re-executing the same branch instructions as the primary one and therefore, at the time of any branch instruction fetch of the redundant thread, branch direction has been already resolved -in the execution of that specific branch on the primary thread. By passing that information for each branch from primary thread to the redundant one, the performance cost of control flow mispredictions can be eliminated.

Data (operants) prediction. A third case accelerated by that “helper” thread scheme, is called data prediction. Additionally to branch prediction, the primary thread can pass along to the redundant thread each instruction result. The motivation for doing so stems from instruction dependencies. A data/instruction dependency exists when consecutive instructions use the results of the immediately preceding ones, forming chains of instructions where the results of the one (i.e. i1) are the operants of the next (i.e. i2). The operants of i2 are not computed until the execute stage of the i1, but are required in the decode stage of i2, which due to pipelining will happen earlier. To overcome this, the decoding of i2 must be stalled until the execution of i1 forcing the whole pipeline to stall. In the context of “helper” threads, data dependencies can be fully eliminated from the redundant thread, if the instruction results of the primary thread are provided to the redundant one to be used as operants. In this manner, the operants of the re-execution of i2 will be always available from the (primary) execution of i1 instead of needing to be later computed by the redundant instance. Emphasis must be laid on the fact that this is not making the system more vulnerable. Even if an erroneous result is being passed between the 2 threads, it would have been detected before, during the formation of these results, from two (correct) operants.

As we demonstrated, optimizations require transferring branch outcomes and instruction operants from the leading to the trailing thread. For this, a hardware buffer, called Delay or Comparison Buffer (Figure 5.3), can be used, being filled from the primary and consumed by the redundant. However, the Comparison Buffer in some cases can deteriorate performance, since when it is full the redundant thread needs to be executed and the primary thread stalls (which we will call ”full comparison buffer” stalls), and when it is empty, the primary thread is forced to execute and the redundant thread stalls until is it filled.

5.3 Parallel Heterogenous Error Detection

The third approach to fault tolerance we are including in this comparative analysis is Parallel Error Detection with Heterogenous Cores. This method pertains to error correction also at the microarchitectural level and has been recently proposed [2]. The goal of the described method is the detection of both soft and hard errors while retaining low area, power, and performance overheads as well as minimal invasiveness to the original microarchitecture. Key idea to achieve that according to this approach is the parallelization of fault detection. Adjoining the main out-of-order processor, on which computation is performed, lie auxiliary processor cores that repeat the same computations, achieving the spatial redundancy needed for error detection. The checker cores are of low power, in-order, and occupy less area, enabling the desired low power and area overheads. Execution on the main core is segmented into parts consisting of a certain number of consecutive instructions and each part after being executed on the main core is offloaded to

one of the checker cores for re-execution and verification.

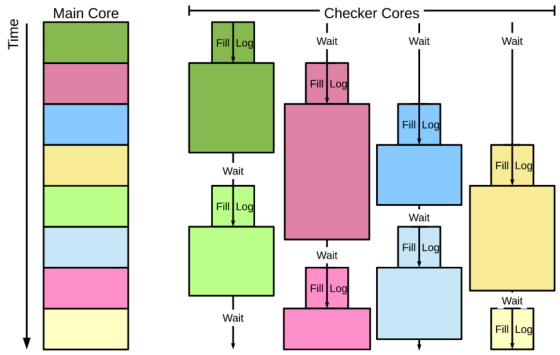


Figure 5.4: Main core execution divided into parts and checker cores verifying each one.

The verification works with the main core providing 3 elements to each auxiliary core: the instructions to be repeated, the starting architectural state of the processor, and the ending one. The state of the checker core is initialized to the provided starting architectural state and if no errors have occurred, after the execution of the segment's instructions the ending architectural state should be identical to the provided ending state. Note that this way, the various program segments are being verified independently and in parallel to each other. However, because auxiliary cores are smaller are also slower and this means that the re-execution of each segment lasts longer than the original execution on the main core. Due to this, as the main core progresses on the execution of the program, many auxiliary cores might be still verifying previous segments.

Therefore, a certain segment being verified as fault-free does not guarantee that the whole program up to this point is fault-free (because previous segments might be still under verification). Nevertheless, when the earliest segment from the ones currently verified is found correct, then the program up to this point is indeed correct. Still, the main processor will be further along in the execution than the point of the program up to which correctness has been checked, but this is the price to pay for guaranteeing low performance overhead; the main core is progressing further along before all the previous instructions have been verified.

From an implementational perspective, a couple of points are worth noticing. Firstly, the starting/ending architectural states are realized as register checkpoints. Yet, saving all the registers stalls the processor for a couple of cycles (a 16-cycle checkpointing latency is being modeled on the original paper). Secondly, as the checker cores lag behind the main one in the execution, in-memory data might get re-written by the main core before being read by the auxiliary cores, resulting in main and checker cores to have read different values. For this reason, checker cores are restrained from accessing the main memory, and each store of the main core to the main memory is being also replicated and saved in a hardware buffer called load-store log. Checker cores are therefore not reading data from the memory subsystem but from the load-store log. This is crucial for maintaining the high performance of the main core (by allowing the main core to progress further than the checkers and not stall waiting for verification) and consequently for enabling parallelism. The load-store log itself is partitioned, and each segment -consisting of a number of entries to be filled with data stored by the main core- corresponds to a particular checker core. When each segment becomes full, a checkpoint is initiated and the corresponding program part is being assigned to an auxiliary core for repetition. If all the segments are

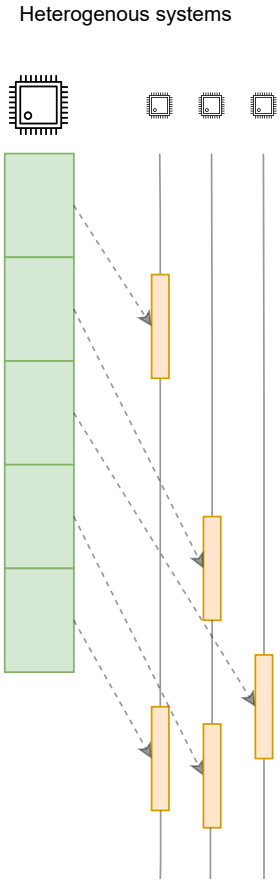


Figure 5.5: Schematic representation of Parallel Error Detection with Heterogenous Systems. Note that the main core is futher along in the execution compared to the slower checkers, because by the time a segment is being re-executed on a checker, the main core is executing subsequent program segments.

full, all the checker cores are active re-executing some program segment and in that case, the main core must stop progressing further, and stall until one checker is freed.

It should be by now evident that checkpointing of the register files occurs when a program segment is being offloaded to a checker core, which in turn happens after a load-store log segment has been filled or additionally after a certain number of timeout instructions.

Inherent to the Parallel Heterogenous Error Detection is a fundamental tradeoff between the low performance overhead of the main core and the error detection latency of the method. As we highlighted the source of performance slowdown is checkpointing. To keep checkpointing minimal and maintain high performance of the out-of-order core, the size of each load-store log segment must be large, in order for the filling of each to happen infrequently, after a lot of instructions. As a consequence, if checkpointing happens rarely and the program segments assigned to one checker core are bigger, the detection latency rises, since each auxiliary core needs to verify more instructions before the second execution of a possibly compromised instruction happens in that core. At the same time, this is also a tradeoff between area/power and detection latency, because to preserve fast detection times, not only checkpointing must happen frequently (and create a large number of program segments) but also to have an adequate number of checker cores available to verify that larger number (of smaller) program segments. In other words, to

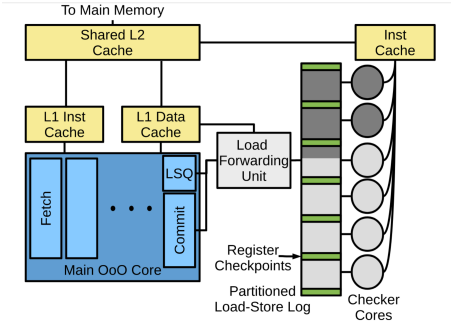


Figure 5.6: The load-store log is divided into segments, each corresponding to 1 program segment and 1 checker core. Each **load-store log** segment is getting filled during the execution of the respective **program** segment and when it is filled, a checkpoint is initiated and the re-execution on the checker core starts.

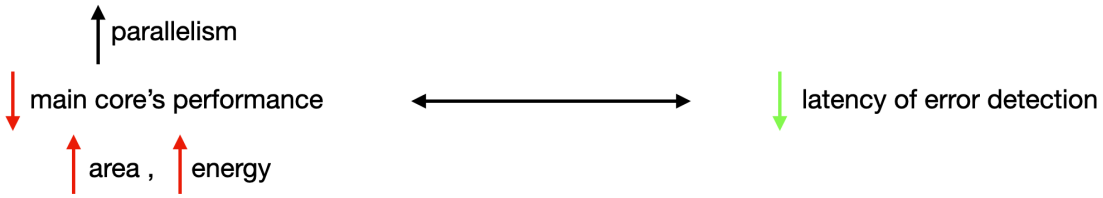


Figure 5.7: When parallelism increases, area/power also increases, the performance of the main core degrades (due to more frequent checkpointing) and detection latency improves.

achieve smaller detection latencies more segments must be verified in parallel and so, the number of main cores need to be also increased, leading to larger area and power consumption of the overall design.

Table 5.1: Qualitative comparison of the fault detection methods against various requirements. Green icons express a desired or good property such as low detection latency and low performance/area/power overheads, red ones an unappealing property such as high detection latency and high overheads, and yellow moderate.

	DMR	R-SMT	Parallel Det.	Ideal for space
Error detection latency	🙂	😐	☹	🙂
Fault coverage	😐	😐	☹	🙂
Performance overhead	🙂	😐	🙂	😐-☹
Energy consumption	☹	🙂	🙂	🙂
Area overhead	☹	🙂	🙂	🙂

5.4 Summary

In this chapter, we introduced the 3 methods which will be compared in this work. The first one, with a tradition of proven usage in space missions and critical systems, Dual Modular Redundancy, performs all computation in parallel in two identical processors

comparing the results of each. The second, R-SMT is a form of Redundant Multithreading where redundancy is achieved by executing 2 copies of the program as SMT threads in the same processor. By exploiting the well-established microarchitectural trend of simultaneous multithreading, the performance overhead of both repetitions of the program is reduced below the cost of two program executions. To further improve that, various techniques can be employed taking advantage of the one thread being further along in the execution compared to the second. The third compared approach to error detection is a recent development from the computer architecture domain, Parallel Error Detection with Heterogenous Cores. Parallel Error Detection with Heterogenous Cores proposes coupling the main out-of-order processor with tiny auxiliary cores which repeat in parallel parts of the main core computation. For each checker core to be able to verify only a part of the main core execution without having executed the previous instructions requires the checkpointing of architectural state before and verifying the instructions, and overall results in a system with low performance overheads for only a fraction more than the main core area and energy, but nevertheless with high detection latency.

6 Design and Implementation of the Studied Error Detection Techniques

A crucial step in the first stages of research and development of new hardware system architectures is software-based simulation. Indeed, in this work, the primary objectives are the implementation and the following evaluation of software simulations of the proposed architectural techniques.

6.1 The gem5 Simulator

For this work, we choose to implement the error detection methods we are studying on gem5 [34], an open-source simulator widely used for computer-system architecture research in both academia and industry. Gem5 is a cycle-level, “execute-in-execute” simulator and models many different devices and processors, enabling the simulation of both system-level components (such as DRAM out-of-chip memories, network devices, GPUs, etc) and CPU microarchitecture. Internally, it is structured as a simulator core combined with parametrized models for each modeled component. For example, it contains many processor models, ranging from simple to more accurate ones. A simple CPU model can be used for studies where modeling CPU performance is of secondary relevance like memory subsystem studies, allowing for faster simulation speeds. On the other hand, high-fidelity models have longer simulation time but can provide more realistic results when compared to real devices and are suitable for studies where the impact of the processor microarchitecture is crucial in the overall system.

6.1.1 Workflow

A typical workflow for conducting computer architecture research with gem5 can be summarised as follows. Usually, researchers will be designing a system which subsequently they will want to evaluate by measuring some metrics such as run time, memory bandwidth, or (more low level ones, like) number of committed instructions per cycle.

For that, one needs to download and build the simulator and then create their own device models on top of the already existing ones in order to construct the designed system mentioned earlier. Due to the modular design of the simulator, these models can then be plugged into the rest of the simulating system, being integrated into the simulator. After that, the user can run gem5, -which is now simulating some modified architecture- benchmark applications and measure the metrics of interest.

6.1.2 Software structure

In gem5, the description and specification of each simulated component are decoupled from the modeling of its function. This is accomplished by a synergy of Python scripts with a C++ backend. A Python-based scripting interface is used to describe the specification of the simulated system, that is, to define all the components that the system will contain (such as the choice of processor model, memory type, interconnection network, on-chip caches, etc.), specify how these components are connected and also select the parameters of each component. The C++ backend of the simulator implements the function of each component. The interlink between the Python specification and the C++ implementation is SimObjects. SimObject in the gem5 terminology are model objects of components whose parameters are specified via Python and are tied with C++ classes that specify their function. For example, the processor of any simulated system is

a SimObject. Through Python one can specify the various parameters of the processor, like the issue width, the type of branch predictor, etc., but the simulator processor logic and function are defined in the respective C++ class.

To revisit what we were discussing earlier about the workflow of simulating a custom system in gem5, more often than not, the researcher will need to create new SimObjects in order to simulate new devices or components, by specifying both their parameters as part of the Python interface as well as the implementation in the corresponding C++ class. However, there are other cases -such as the one of this project- where existing SimObjects need to be modified. In our case, because we are simulating methods at the microarchitectural level, this requires extensions of the microarchitecture itself of the processor. This proved to be a much more difficult procedure than simply adding new derived SimObjects because we had to navigate a lot of different models and understand both the codebase related to the processor SimObject that we used as well as the core of the simulation framework.

6.1.3 Simulation modes

After having specified or modified the system, when it comes to running an actual simulation, the gem5 simulator can be used in two different modes: system call emulation and full system simulation.

In System Emulation mode, system calls to the operating system are emulated, meaning that only user-space code is being modeled. Binaries will be executed on the simulated computer system but without executing the kernel-mode system calls of the operating system. This has the effect of ignoring the timing of many system-level effects including system calls, TLB misses, and device accesses, and as a result, lowers the fidelity in system modeling but gives faster simulation times and is also easier to config, since many hardware units required in a real system do not need to be instantiated.

In Full System mode, gem5 is modeling the entire hardware system, including the interaction with the operating system. In this mode for example, the simulator can boot a full Linux-based operating system running an unmodified kernel, where afterwards the user can run the application under study and gather the required statistics at cycle-level. On the contrary, in the System Emulation mode, the simulator itself is emulating the operating system.

For this study we choose to use System Emulation mode for three reasons: Firstly, the effect of the operating system in the error manifestation and in the detection methods are out of the scope of this work. Secondly, we can benefit from the extra performance of the System Emulation mode, because the biggest amount of time was spent developing the simulator extensions for supporting the studied detection methods and hence, we want to avoid long simulation times in the development phase. Lastly, we found this mode easier to configure and get started with, given it was the first time the author was using this tool.

6.2 Implementation of Dual Modular Redundancy

6.2.1 Voter Unit

For implementing Dual Modular Redundancy on gem5, the main component we added to the system is a voter unit. The voter is the hardware structure responsible for comparing instructions from the two processors in order to detect an error. Conceptually, it takes as input the two identical instructions from the two CPUs and it produces an output signal, signifying whether both executions were the same or if a fault had occurred.

In our implementation, given the two identical instructions from the two CPUs, an error is

detected if: a) the program counters of the instructions are different, or b) the results of the instructions are different.

The first case, a mismatch between the 2 program counters, can happen in one of the two following cases: if an error alters the value written in the Program Counter register (PC) of one of the two CPUs, causing that CPU to jump to a different instruction in the next cycle than the one in the correct program order, or if an error altered the direction of a branch, that is, to alter the result of a branch instruction, leading the one CPU to execute the branch as taken and the other as not taken. Note that, actually, the detection of different PCs is expected to never happen in this implementation of DMR, since firstly, fault injection in the program counter is not performed in this study and thus an error will never cause a mismatch between the 2 program counters and secondly, the different results of a branch instruction in the 2 CPUs will never lead to divergent execution between the two CPUs in any possible DMR implementation (unless slack between the 2 CPUs exists), since it will have previously detected as an error, while veryfing the results of the branch instruction. However, since we used a similar voter for the R-SMT method (where PC mismatch can occur), we choose to share some of the voter's attributes in both implementations. The second case (detecting different results), of course, can happen due to the manifestation of an injected error to the instruction result of one of the two CPUs and is the primary detection method in the setting we are simulating.

We design the voter with the following interface in order to be easily integrable into the rest of the system. On the commit stage of each CPU, before an instruction is retired, it is fed into the voter unit. The voter is equipped with input buffers, storing the instructions from each CPU. When the 2 buffers are of equal size, instructions are popped from both and compared with each other. Practically, this happens on every cycle and the buffers never store more than 1 instruction, but once again the same design will be used for the other implementations. The instructions are compared by ensuring that the pc and the result of the instructions are the same. If one of the two is found to be different, an exception is raised and the program is halted.

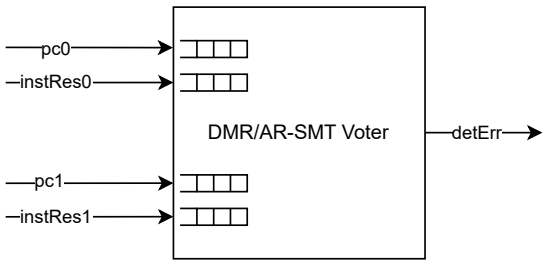


Figure 6.1: Abstract schematic diagram of the the voter interface. Voter accepts PCs and instruction results from the 2 CPUs (in the case of DMR) or from the 2 threads (in the case of R-SMT) and can detect errors based on the mismatch of PCs or results. For DMR, an error will never produce PCs mismatch in this implementation, but the interface was kept common for both implementations.

Regarding the timing of the above operations, two possible approaches were examined: the voter being simulated as a synchronous sequential circuit or as an asynchronous one. In the former case, this means that the comparison would be performed in the next clock edge, so 1 cycle after the commit of the instructions, whether in the latter the comparison would be performed concurrently with the commit cycle. We chose to model the asynchronous implementation since such a circuit would avoid the unnecessary delay of 1 cycle in the error detection.

6.2.2 Dual CPU modelling

Apart from the voter, the DMR implementation needs to model two different program executions in two different CPUs.

Initial approach

The initial approach to this was to actually compose a system with 2 identical CPUs connected to the same memory subsystem (with private L1 caches and shared L2) and run the same binary on both. This however resulted in a system with slack between the 2 cores, meaning that the 2 processors were executing instructions not in sync. This is due to the cores sharing the L2 caches, which are single-ported. As a result, only 1 processor was able to access the memory at each time. When the first core was to access the L2 cache, the second had to stall before performing the same memory access whether when the second was eventually performing the access, the first would continue the execution of the next instructions without waiting to synchronize with the second. This revealed a major drawback of DMR as a method, which is the need for addressing exactly that problem. This can be resolved by having the memory controller forward the result of each access to both CPUs, without letting both of them access the memory, since in DMR the same instructions are executed concurrently by both processors. In other words, DMR requires minor modifications to the memory controller or dual-ported caches.

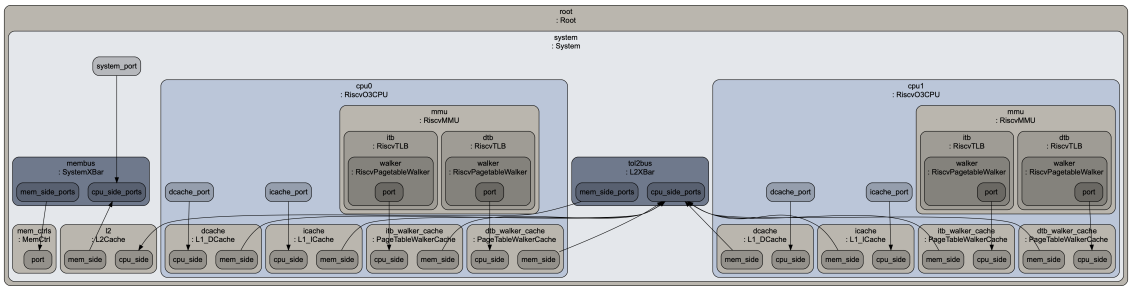


Figure 6.2: This component diagram is automatically produced by gem5 and reflects the simulated architecture of the initial approach to DMR implementation, with two CPUs in parallel. We can see that both CPUs connect to the same memory bus and L2 cache, which is single ported. As a result, sequential accesses of the 2 CPUs to the memory cause slack between them and render this DMR approach incorrect.

Pseudo-DMR

Because we believe that the design of DMR itself is out of the scope of this study, which primarily focuses on error detection comparatively between different methods, we chose to model DMR in a simpler way to alleviate the above problem, nevertheless accurate in all aspects related to error detection. Instead of simulating two hardware CPUs in the same system, we perform two executions of the same program in the same processor, with each execution acting as one of the CPUs. In more detail, the first execution (“auxiliary” execution) is gathering the results of all the instructions and the second execution (“simulation” execution) provided this result trace is able to detect an error. For this, we extended the above voter implementation to either save the results trace in a file (for the first execution) or accept a results trace from a file (for the second execution) against which the instructions of the (single) CPU will be compared. We inject errors in only one of the two executions and when measuring detection latency we measure the time difference between error manifestation and detection, as we would normally measure in a 2 CPUs system. For modeling the performance overhead, once again, because DMR adds no delay in the execution, we measure the performance of the “simulation” execution, and this approach has the same result as in the 2 CPUs approach. Therefore this approach

is equivalent to DMR as presented above for all the aspects of interest in this study (error detection, latency, overhead).

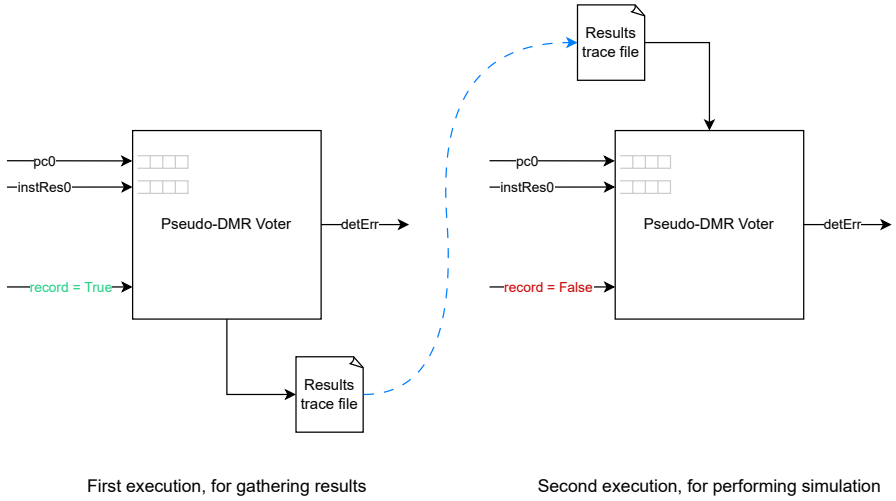


Figure 6.3: Schematic diagram of the voter used in DMR. DMR was not simulated as two parallel CPUs but through two executions of the same binary in the same CPU. In the "auxiliary" execution instruction results are recorded and provided in the "simulation" execution for comparison with the produced results. However, this approximation has the same error detection, latency, and performance overhead with a real DMR implementation.

6.2.3 Implementation & integration

Before describing the implementation of the Voter itself, first, we need to explain how it interacts with the rest of the pipeline. The CPU model used in all methods is the O3 Model, which models a superscalar, out-of-order CPU, based on the Alpha 21264 microprocessor. It has 5 pipeline stages: Fetch, Decode, Rename, Issue-Execute-Writeback (IEW), and Commit. The voter is being filled with instructions from the Commit stage. At each stage, a `tick()` function is defined, which implements the logic of this stage and updates various signals and buffers. For the commit stage, `Commit::tick()` is calling `Commit::commit()` which initiates the committing of completed instructions and in turn calls `Commit::commitHead()` which attempts to commit the head instruction of the ROB. From this function, in "simulation" executions the Voter class is being provided with instructions to compare whether in "auxiliary" executions, the Voter writes results to trace file.

The voter is implemented as a SimObject class. Internally, it defines a `comparisonElement`, which is a class that encapsulates all instruction characteristics which are going to be compared for detecting an error. Each `comparisonElement` object corresponds to one instruction and contains the gem5 representation for the PC, result, and some other auxiliary info for that specific instruction.

Methods of the `Voter` class worth discussing are:

`void Voter::compare(comparisonElement e1)`
Triggers a comparison between the provided comparisonElement and the corresponding comparisonElement acquired by the results trace file.

`void Voter::write_to_file(comparisonElement e)`
Writes that `comparisonElement` to the trace results file, and it is called only in the "auxiliary" execution.

6.3 Implementation of Redundant Multithreading (R-SMT)

In this section we are presenting the implementation of the R-SMT method. From the optimizations described in 5.2 we are implementing only perfect branch prediction.

6.3.1 SMT fetch arbitration

By default, the O3 model is using a priority based Round-Robin fetch partitioning policy, meaning that in each cycle, one thread is fetching as many instructions as possible (within the fetch width), being interrupted only by two possible scenarios:

1. a branch being identified as taken in the decode or execute stage which will change the next pc
2. exhausting the available instructions in the current I-cache line which would require accessing the next one in the following cycle

If however for some reason one thread is unable to fetch (due for example to a long-lasting I-cache miss), the next one will utilize this slot fetch instead. This is a sound choice of fetch partition algorithm even for a redundant SMT for fault tolerance design like the one implemented here, but in order to be able to better study the impact of the slack between the 2 threads in error detection, we modify the above algorithm in order to perform the following: each thread must fetch a constant number of instructions before the other one is allowed to fetch. This reduces performance slightly because in any case, the fetch stage is stalling, the whole pipeline is forced to stall too, but is the only way to create a fixed slack of certain instructions between the threads (called SMT granularity), the effect of which in fault detection, we believe is crucial to study. SMT granularity can be specified in the simulation start.

Implementation
SMT fetch arbitration policies are implemented in `Fetch::getFetchingThread()`, which returns the Thread Identifier (TID) of the next fetching thread. `Fetch::roundRobin()` method, as well as the conditions on which fetch is reading instructions from the Icache block, in `Fetch::fetch()` are modified. According to our policy, each thread needs to fetch a constant number of instructions, called fetch/SMT granularity, before the next thread is allowed to take over.

Because we are using the Oracle branch predictor and since we are not implementing the value prediction, we do not need an actual Comparison Buffer, since no information is passed on-the-fly between the threads. However, we need to model the effect of the Comparison Buffer on performance, stalling primary thread when it is full and the redundant thread when it is empty. To do so, we restrain the fetch arbitration algorithm such so, enforcing fetch from primary thread when the Comparison Buffer has less than a specific number of entries (Comparison Buffer minimum size). Also, because the redundant thread is progressing faster than the primary thread, after the enforced primary thread fetch, we further fetch a number of extra instructions (Comparison Buffer recovery size) to avoid immediately emptying again the Comparison Buffer.

6.3.2 Voter Unit

The R-SMT design also uses a voter to compare the instructions between the 2 threads and detect errors. The design of this voter is similar to the voter used for the DMR method.

Note that here, the input buffers holding instructions pending to be compared are essential, because the same instruction from the 2 threads is not added to the voter at the same time, due to the slack between the threads. However, this voter does not need to read and write results traces as the DMR variant was able to do.

Implementation
Voter class is again being filled at Commit, but this time each thread fills the respective input port of the voter. Similarly to the `comparisonElement`, the R-SMT voter class defines the `delayBufferElement` class, which also contains thread information.

Methods of the Voter class worth discussing are:

`void add_to_queue(delayBufferElement e)`
Adds a `delayBufferElement` to the input buffer of the corresponding thread.

`void compare_and_pop()`
Compares the head instructions from the two input buffers. If no error is found the instructions are removed, otherwise a detected error exception is being raised.

6.3.3 Perfect branch prediction

As we have described, the redundant thread is utilizing an oracle/perfect predictor to predict branches, fetching the actual direction of each branch as resolved in the execute stage of the primary thread. In order to implement that, we need to create our own "perfect" branch predictor for use by the redundant thread. This branch predictor will be filled by the primary thread and used by the redundant one.

The first approach to this was to implement exactly that: on-the-fly, the primary thread to be supplying branch outcomes to the predictor of the redundant thread which will be using them for the prediction of its branches. However, after implementing that, it was found that it would not be possible to operate correctly, for reasons we will explain now. In this initial approach, the "perfect" branch predictor was supplied with the actual direction of each committed primary thread's branch during the execution of the 2 SMT threads (on-the-fly). These branch directions were being stored in a FIFO buffer and the redundant thread, was probing the branch predictor for the prediction of each branch instruction. The issue that arose and deemed this approach problematic had to do with the following: This unit should be able to match instructions from the primary thread with instructions from the redundant thread, in order to pair each prediction request with its outcome. This is not straightforward to implement, because instructions do not have any sort of unique id or characteristic. The same instruction can appear multiple times in a program repeatedly. Given that, the only resort for matching primary and redundant instructions is some sort of sequential mapping (i.e. the first primary thread's instruction to be paired with the first redundant instruction, the second with the second, and so forth). This could work and lead to a working on-the-fly "perfect" branch predictor if there was not a final detail: the sequence of instructions between the 2 threads is not the same when the second is using perfect branch prediction. That is because certain instructions fetched by the redundant thread were never committed in the primary thread. The said instructions are the ones that were squashed from primary thread and will subsequently be squashed and not committed from the redundant too. Squashing is the process of removing an instruction from the CPU pipeline after it has been fetched but before being committed, and can happen for various reasons, as we have mentioned. To summarise, the reason for not being able to implement an on-the-fly branch predictor correctly was due to the fact that this would require a sequential mapping between the instructions committed primary instructions and the fetched redundant instructions and something like that is not possible

because of the instructions squashed in the primary thread.

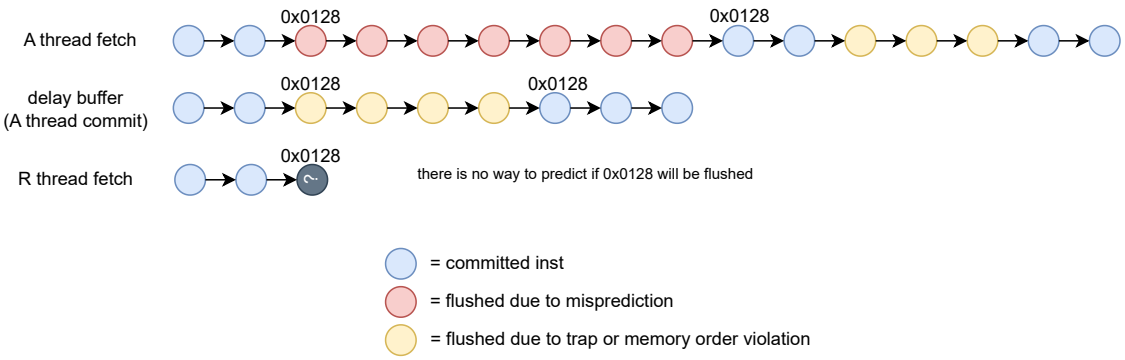


Figure 6.4: Why a perfect, on-the-fly branch predictor could not be created? Instructions committed by A thread enter the Comparison Buffer and redundant thread needs to match them with redundant fetched instructions. Because of flushed instructions, this 1-to-1 mapping cannot be formed at the time of fetch, because it is not known if the current instruction will be flushed or not. To complicate the problem even more, the same instruction (PC) might had occurred in primary thread's fetch order firstly as a flushed one and then as a committed one.

Due to all these, we resorted to creating a simpler oracle branch predictor, which utilizes a committed instructions trace from a previous execution. In other words, the simulation is run once, and the directions of all committed branches are recorded into a file. Then, the simulation is restarted. The oracle branch predictor of the redundant thread will read this file and upon each prediction supply the correct entry from the file. All these leave unaffected the primary thread's predictor which is of course, not ideal.

6.3.4 Implementation

In gem5 new branch predictors can be created by defining classes that inherit from the `BPredUnit` class. This guarantees a unified interface. This class defines the following abstract methods (among many others) that need to be implemented by each branch predictor.

`bool BPredUnit::predict(...)`
Returns whether a branch was taken or not taken.

`void BPredUnit::update(...)`
Tells the branch predictor to update any prediction information for some specific instruction.

`void BPredUnit::squash(...)`
When a misspeculation squash happens, this function provides feedback to the branch predictor about the wrong prediction that caused the misspeculation.

We create `OracleBP`, derived from `BPredUnit`. `OracleBP::predict()` is realized as follows: upon initialization of an `OracleBP` object, the trace file with the directions of all branches from primary thread is passed as a parameter. The file is parsed and the branch directions are saved in a FIFO queue. Each `OracleBP::predict()` call returns the prediction in the head of the queue. In order for the branch directions trace file to be created, a runtime parameter to the simulation executable indicates whether in the current execution primary thread is recording branches, or redundant thread is predicting branches.

In the first case, in `Commit::commitHead()`, each control flow instruction along with the branch direction is recorded in the file.

6.4 Parallel Error Detection Code Artifact

For Parallel Error Detection we use the open-source code artifact distributed by the authors [35] and augment it with our fault injection modules.

6.5 False Positives

All implementations have the ability to detect false positives. It was found that the results of certain instructions change between executions of the same program and same inputs and hence would trigger an error detection in cases where no error was present. For this, we extended the voter units in order to accept a series of program counters to ignore when detecting an error on that instruction. To identify these atypical instructions we simply execute the same binary twice without performing any injection and compare the results.

6.6 Fault Injection

Here, we describe the fault injection campaign from an implementational perspective. We have chosen to employ fault injection only in the integer register file. The register file is an important and vulnerable unit and given the available time in this project, we decided to limit the injection to this unit only. The fault injection is realized through a fault injector object we implement, which is an attribute of each CPU and it models both the timing and the location of each occurring error. In our case, the location is limited to one of the architectural integer registers, and the time of each error manifestation is specified as the number of accesses in that register (injected access time). Each error is modeled as a bitflip into the register value by applying a bitwise mask during the register read. Other parameters of the fault injector are whether the error is going to be transient or permanent and the mask which will be applied to produce the bitflips.

- Transient errors are modeled by performing the bitflips during the register access occuring at the injected access time.
- Permanent errors are modeled by performing the bitflips during all register accesses happening after the injected access time.

6.6.1 Implementation

The `RegisterFaultInjector` class, realizes fault injection. The register to be injected, the register access during which this error will occur and all other parameters are passed through the Python interface into the `RegisterFaultInjector` SimObject and can be specified via command line arguments into the simulator executable. The interface is based on the following function:

`RegVal RegisterFaultInjector::compromise(...)`
Takes a physical register id, its register value, and the current access time and if the conditions for the injection are being met (injected access time reached) returns a erroneus value, otherwise acts as passthrough.

In order for `RegisterFaultInjector::compromise()` to alter values of the integer register file, this method is called upon each register access, namely within the integer register file modeled in `PhysRegFile` class.

When `RegisterFaultInjector::compromise()` actually performs a bitflip, the current simulated time is recorded into a file, for the calculation of detection latency.

6.7 The MiBench Benchmark Suite

In computer systems research, usually, concepts and designs are tested and evaluated during the execution of certain benchmark programs. The role of a benchmark is to evaluate against sets of specific, well-known programs and through that ensure that the results are transferable in any other more specialized program, regardless of the particular characteristics or irregularities of each. For the reason specified, benchmarks must represent a wide variety of programs that are representative of all the real possible applications and produce repeatable and measurable results.

In this study we chose to use the MiBench benchmark suite [12] for evaluating the implemented methods. MiBench is a collection of commercially representative programs targeting embedded devices. The benchmarks that MiBench contains are divided into six suites, with each suite targeting a specific area of the embedded market. The six categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications.

We decided on using MiBench because we judge that this benchmark is more appropriate for a space applications system, mainly because it combines programs from Industrial Control, Networking, Telecommunications, and Security, domains which all will be present in a complex in-orbit or space probe system requiring fault tolerance hardening, such as the ones which could be utilizing the methods we are studying.

7 Results and Analysis

In this section we are presenting the methodology for conducting the experiments of this study, the obtained results and we analyse the conclusions drawn by them.

7.1 Methodology

As it has been made clear in the rest of the text, the main purpose of this study is to compare the 3 fault tolerance methods in four axes: the ability to detect errors, the detection latency, the overhead in the processor performance and the layout/area overhead. To do so, we conduct a fault injection experiment, emulating the presence of errors in 3 systems -each one employing one of the studied fault detection techniques- while running a variety of benchmarks from the MiBench suite.

The experiments we design are of the following structure: in each program execution, we inject one error in a random register. To be able to compare more fairly the results across the 3 methods, we keep the injected registers and the time of injection the same between each method. We perform a large number of executions repeatedly, injecting on different registers and at different times, and record the metrics of interest.

For detectability, we record the outcome of the execution, that is if the injected error leads to one of the following:

- a *crash*, for example to a segmentation fault, due to an illegal memory access
- a *hang* of the processor in an unrecoverable state, which is detected after a certain timeout period has passed
- the normal completion of execution without the manifestation of any error in the system (*masked*)
- the completion of execution without the manifestation of any error in the system but with erroneous results (*silent data corruprion (SDC)*)
- the successful *detection* by the fault detection method

For the error detection latency, we measure the time elapsed between the error injection and the successful detection of an error and for the overhead on the performance, we measure a metric usually used in computer architecture, Instructions per Cycle (IPC). IPC signifies the performance of a processor in the notion that more potent processors have higher values of IPC since they can execute more instructions in the same amount of time. As we have mentioned, non-superscalar CPUs which cannot process more than 1 instruction per cycle, have IPC lower than 1, and superscalar ones can have IPC higher than 1.

7.2 Experimental results

7.2.1 Detection Latency

Quantifying R-SMT slack

In R-SMT, the delay in the retirement of an instruction by the primary thread and redundant thread, is called commit slack. Since in R-SMT the error detection occures when comparing instruction results on commit, commit slack is an important factor contributing to detection latency. In Figure 7.1, we investigate how commit slack varies depending on

Table 7.1: Experimental setup

Common options for all methods	
CPU [a]	O3 CPU model, 4-way out of order, 2 GHz, 192-entry ROB, 64-entry IQ, 32-entry LQ, 32-entry SQ, 256 Int registers, 256 FP registers
Branch predictor	Tournament, 2048-entry local, 8192-entry global, 8192-entry chooser, 16-entry RAS, 4096-entry BTB
L1 Dcache	32kB, 8-way, 2-cycle hit latency
L2 cache	2MB shared, 16-way, 20-cycle hit latency
Main memory	DDR4_2400, 8GB
R-SMT specific options	
SMT Fetch policy	Round Robin
Comparison Buffer	10 entries
Parallel Det. specific options	
Load store log	36 KiB
Checker CPU	16 checker CPUs, Minor CPU model, in-order, 4 stage pipeline, 1 GHz

[a] for Paralle Det. this refers to the main CPU

the size of the comparison buffer. We find that, smaller buffer sizes result to smaller commit slack. This is expected, since when the primary thread has placed more instructions in the comparison buffer, than the redundant has consumed, the buffer becomes fully occupied and the primary thread must stall, since there are no available entries to store subsequent instruction results. As a result, the redundant thread commits instead, progressing further in the execution and converging with the primary one, hence reducing the slack. Given that with smaller buffer sizes, the primary thread is more frequently stalled, smaller comparison buffer sizes eventually lead to smaller slack between the threads.

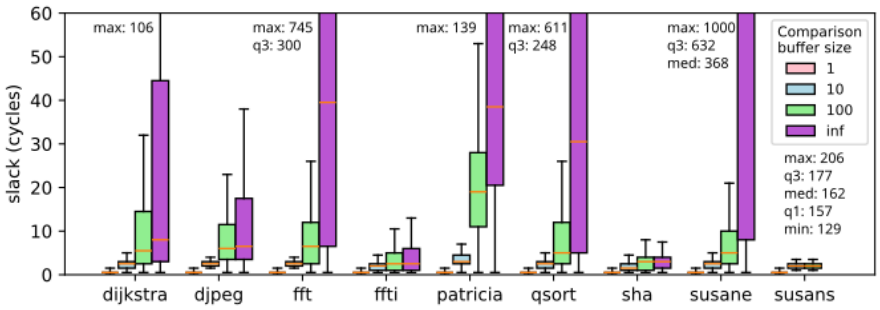


Figure 7.1: Distribution of commit slack for R-SMT, when varying the comparison buffer size. Each box is lower-bounded by the first quartile and upper-bounded by the third quartile. The median falls within the box. The inter-quartile range (IQR) is the distance between the first and third quartiles (i.e., box size). Whiskers extend to the minimum and maximum data point values on either sides of the box.

Measuring Detection Latency

Regarding the detection latency, plotted in Figure 7.2 for all methods, we draw the following conclusions:

- DMR exhibits the lowest minimum (in all benchmarks) and mean (in 8 out of 9 benchmarks) values compared to the other 2 methods. This is expected since the 2 processors are concurrently executing the same instructions. Hence, the delay for DMR is only the time between the decode and commit stages.
- For R-SMT, the slightly higher values are attributed to the slack between the 2 threads.
- Lastly, Parallel Detection consistently exhibits higher min and mean detection latency, due to the fact that checker cores are smaller and in-order are slower and the main core progresses much further along the execution.

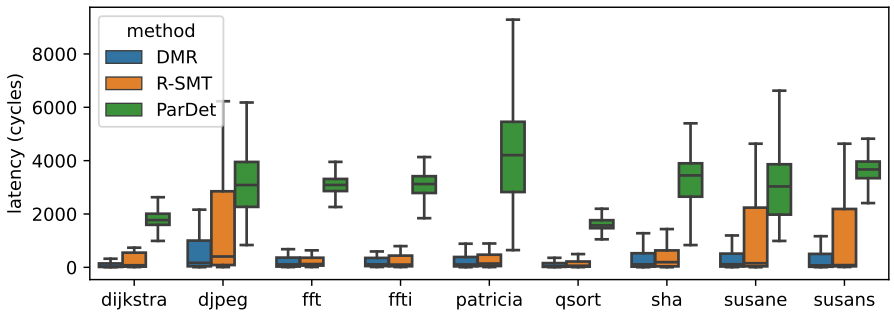


Figure 7.2: Distribution of error detection latency for the 3 methods.

7.2.2 Detectability

To evaluate the detectability of each method, we perform two series of experiments as described above, one with transient and one with permanent errors. We also conduct the same fault injections in a system with no fault detection capability, to use it as our baseline.

From Figure 7.3 regarding the transient errors experiment we recognize the following:

- On the unprotected system, we observe that all the error injections lead either to crashes, masked, or timeouts. The masked executions are own to the fact that the injection to that specific register is not resulting in any change to the instruction results. For example, this can happen if the affected register value is being masked by a later instruction (such as an AND operation with 0).
- Parallel Error detection detects the highest number of errors. This is because ParDet's implementation of architectural state comparison for error detection (as opposed to

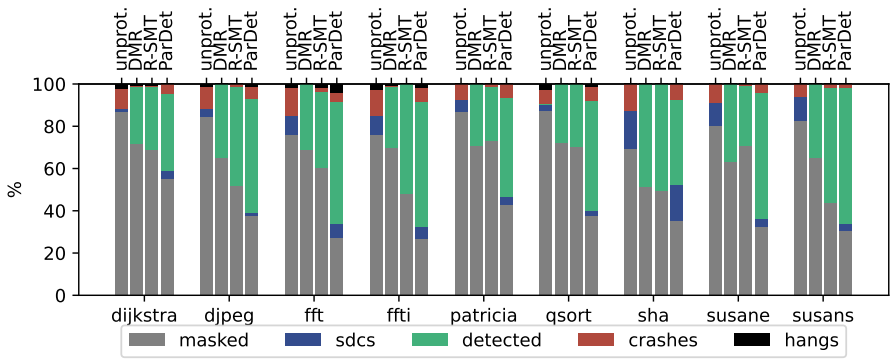


Figure 7.3: Results of transient error injection across the 3 methods

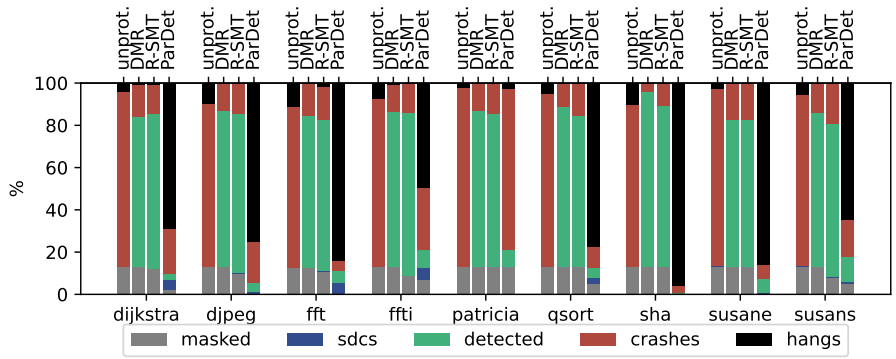


Figure 7.4: Results of permanent error injection across the 3 methods.

results comparison) which classifies as detected errors which for DMR and R-SMT would be correctly classified as masked.

- R-SMT detects more errors, compared to DMR, attributed to the structure of our fault injection experiment. To guarantee a fair comparison, in all methods, we inject the same registers after the same time inteval (measured from the start of the program execution) has elapsed. As a result, the same errors are injected earlier in program order for R-SMT, compared to DMR. Thus, errors manifested earlier have higher probability of propagating to more registers through data dependencies and consequently, potentially corrupt more instructions, and as result are more likely to be detected.
- Despite DMR being the methods with the lower percentage of errors detected, it also exhibits the fewest failures (crashes and timeouts). This is expected since in DMR, the error is being caught at the earliest possible time, when it manifests for the first time in the program instructions results, since, because the 2 processors are executing always the same instruction, any difference will be found when the first affected instruction will be committed. This is not the case for R-SMT, where, as detailed below, an error can be detected after a number of affected instructions commit, due to the slack between the threads. Therefore early detection prevents errors from propagating further into the system, something which could lead to crashes or hangs. That is to say, the small detection latency of DMR (which was also confirmed with the previous experiment) contributes to the low failure rates.
- The small amount of crashes that appeared in some benchmarks (i.e. sha) in DMR, can be attributed to the propagation of the error in the period between decode (error injection happens during the reading of the operant registers) and commit (error detection). Due to the superscalar nature of the simulated processor, the value of the corrupted register is being read by the instruction during the fetch of operant values, and stored in the ROB. After the instruction result has been formed it is being stored again in the ROB, from where it is being consumed by all subsequent instructions requesting that. This way the error has a small chance to propagate in the pipeline before committing the erroneous value to the register file and hence can propagate without getting detected.

Regarding the the permanent errors experiment, from Figure 7.4 we can deduce:

- In the presence of permanent errors, the unprotected system displays much more crashes than silents across all benchmarks, because since errors are permanent

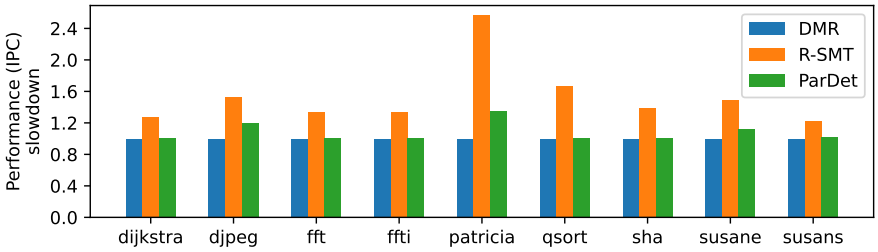


Figure 7.5: IPC slowdown across the 3 methods

they are propagating more in the program flow compared to transient ones, making it more likely to crash.

- The higher detection ratio that both DMR and R-SMT systems exhibit in all benchmarks compared to the transient error injections can be attributed again to the same reason, that is, permanent errors propagating more into the system, altering more instruction results and hence being more likely to be detected.
- Lastly, Parallel Detection has the smallest percentage of detected errors, attributed to the increased detection latency.

7.2.3 Performance overhead

- Regarding the performance overhead (Figure 7.5) of DMR, we can observe that this is the minimum across the 3 methods since it preserves effectively the same performance as in normal (unprotected) execution. As we have explained this method introduces no slowdown to the processors.
- Parallel detection has (negligibly) lower IPC values, due to the impact of checkpointing which stalls the main core. This confirms the claims of the original publication that the method has minimal performance overhead.
- Lastly, R-SMT is the least performant method across all. We can attribute this to the following 2 reasons:
  - the partitioning of some microarchitectural components between the SMT threads -especially of the fetch unit- hinders performance and creates a bottleneck
  - the performance loss due to the stalls caused by the full comparison buffer

7.2.4 Area overhead

In Table 7.2, we estimate the area overheads of each method, compared to an unprotected design. We assume DMR duplicates the whole processor core, resulting 100% overhead. For R-SMT, area is increased due to two components: the area overhead of SMT and the comparison buffer. We quantify the area overhead of SMT by relevant literature [13]–[15], placing the layout overhead in real designs in less than 6% of the area of the core. The overhead of the comparison buffer (of 10 entries), is calculated by comparison with the L1 cache: 10 entry-buffer corresponds to 0.125% of the L1 cache and contributes to an additional 0.04% area increase [16]. For Parallel Detection, area is increased both by the overhead of the auxiliary cores and the load-store log. For both components we reuse the results from the original publication [2], since we are also modelling the same microarchitecture.

Table 7.2: Area overheads

Method	Overhead per Component		Total area
	Component	***Overhead***	
Unprotected	-	-	1x
DMR	Redundant core	100%	2x
R-SMT	SMT overhead	6%	1.0604x
	Comparison buffer (10 entries)	0.04%	
ParDet	Checker cores (12)	20.2%	1.24x
	Load-store log (36 KiB)	3.8%	

7.3 Analysis and discussion

Based on all the above, we can draw some important conclusions.

First of all, regarding detection latency, DMR exhibits the lowest, the next highest is R-SMT, and then Parallel Detection, with the largest latency. As we can see in the schematic of Figure 7.6 this is something we were expecting. DMR detects the error upon commit so latency is minimal. In R-SMT detection latency is the slack between the 2 threads. For Parallel Detection, auxiliary cores are slower than the main core so the detection latency is higher. Detection latency however has an immediate impact on detectability. From our experiments, it was found that for the 3 methods, the number of undetected errors that lead to crashes is inversely proportional to detection latency. This is not a coincidence. Long detection latency (such in the case of Parallel Detection) means that in the time between any error appearance and the re-execution/verification, there is enough time for the error to propagate and hence the possibility of a crash is increased.

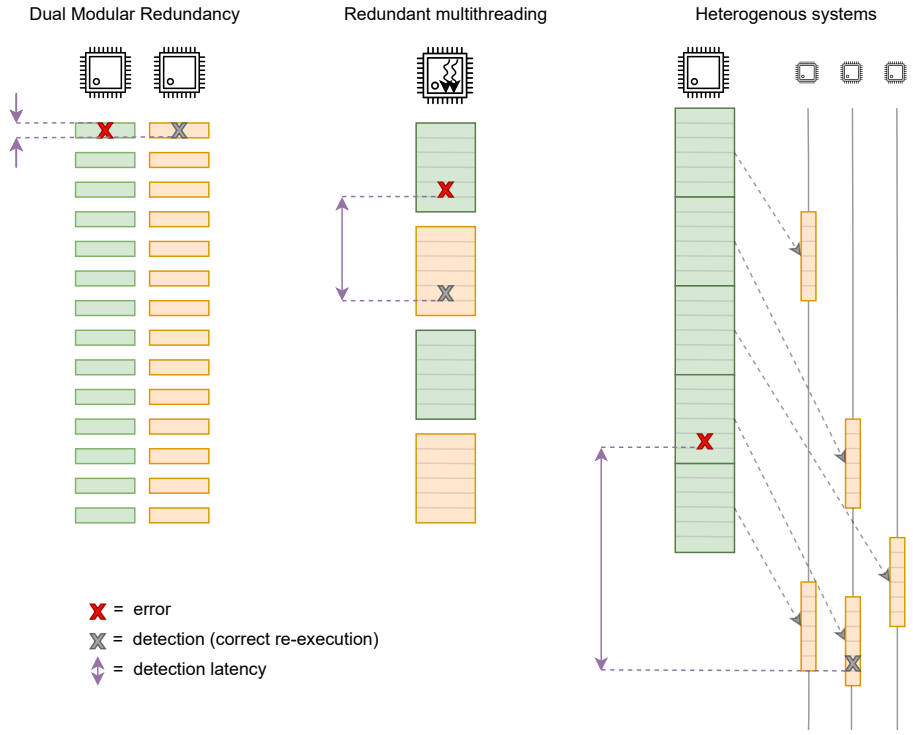


Figure 7.6: Schematic comparison of the different detection latencies for each method.

At the same time, R-SMT displayed very similar detection latency and detectability with DMR. Therefore, we can safely deduce that R-SMT is a viable alternative to DMR in

terms of detection latency, while at the same time reducing both area and energy overheads. This is an encouraging result since it shows that microarchitectural insights can successfully optimize fault detection methods in the metrics that are important in space applications (such as error detectability and latency).

Finally, when it comes to Parallel Detection, we have verified the theoretical traits of this method: small performance and area overheads with high detection latency and moderate detectability. We deem that this is an unappealing feature for spaceborne computing, where detection latency and error mitigation are more important than performance or area, which are of secondary importance. Even more, the fundamental tradeoff of this method (Figure 5.7), which states that high error detection latency is required to attain high performance and low area is also unsuitable for space. Another significant point that makes Parallel Detection not ideal for space is that it allows propagation of errors in main memory or in writes outside the system.

To summarize, the key takeaway is that R-SMT is on par with DMR in regards to latency and error detection ability while exhibiting negligible area overheads and retaining a small overhead on performance. Parallel Detection on the other hand has different strengths, outperforming R-SMT in performance but lacking in latency and detectability, rendering itself unsuitable for space applications but displaying successfully how microarchitectural insights can optimize fault tolerance methods.

8 Conclusion

8.1 Final remarks

In this work, we studied fault tolerance methods that leverage microarchitectural insights, suitable for spaceborne processors. We compared 3 methods, Dual Modular Redundancy, which is widely used in real-world space missions, Redundant Multithreading with Simultaneous Multithreading, which utilizes SMT threads to increase the performance of re-execution, and Parallel Error Detection with Heterogenous Cores, which is the state of the art method of microarchitecturally-assisted fault tolerance. The 3 methods were compared in 4 axes: the error detection ability, detection latency and area overheads, which are of primary importance for space applications, and performance overhead, which we believe emerging developments in the space sector will render equally important in the future years. We discover that R-SMT has nearly equal detection ability and latency with DMR while retaining minimal area overheads and degrading processor performance only slightly. On the other hand, we verify that Parallel Detection exhibits better performance overheads but falls short in detectability and latency which are unappealing features for computing in space.

8.2 Future work

Possible future research direction could include:

- The refinement of fault injection, by compromising more pipeline elements and thus covering larger parts of the processor.
- The examination of more methods that exploit different microarchitectural trends, like hashing for faster verification or reuse of cached instruction results.
- The evaluation of fault tolerance techniques in the presence of an operating system.

Bibliography

[1] E. Rotenberg, “AR-SMT: A microarchitectural approach to fault tolerance in microprocessors”, in *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, Jun. 1999, pp. 84–91. DOI: 10.1109/FTCS.1999.781037.

[2] S. Ainsworth and T. M. Jones, “Parallel Error Detection Using Heterogeneous Cores”, in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2018, pp. 338–349. DOI: 10.1109/DSN.2018.00044.

[3] J. E. Tomayko, “Computers in Spaceflight: The NASA Experience”, Tech. Rep. NAS 1.26:182505, Mar. 1988.

[4] P. Parker, *Integrated circuits in the Apollo manned lunar landing program*, https://-history.nasa.gov/alsj/apollo-ic.html, Dec. 2003.

[5] B. Denby, K. Chintalapudi, R. Chandra, B. Lucia, and S. Noghabi, “Kodan: Addressing the Computational Bottleneck in Space”, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 392–403, ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582043.

[6] L. Massengill, “Cosmic and terrestrial single-event radiation effects in dynamic random access memories”, *IEEE Transactions on Nuclear Science*, vol. 43, no. 2, pp. 576–593, Apr. 1996, ISSN: 1558-1578. DOI: 10.1109/23.490902.

[7] P. Dodd and L. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics”, *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583–602, Jun. 2003, ISSN: 1558-1578. DOI: 10.1109/TNS.2003.813129.

[8] D. Tullsen, S. Eggers, and H. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism”, in *Proceedings 22nd Annual International Symposium on Computer Architecture*, Jun. 1995, pp. 392–403.

[9] S. Mukherjee, M. Kontz, and S. Reinhardt, “Detailed design and evaluation of redundant multi-threading alternatives”, in *Proceedings 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 99–110. DOI: 10.1109/ISCA.2002.1003566.

[10] S. K. Reinhardt and S. S. Mukherjee, “Transient Fault Detection via Simultaneous Multithreading”,

[11] Y. Hua, C. Gang, and Y. Xiao-zong, “TRSTR: A fault-tolerant microprocessor architecture based on SMT”, *Wuhan University Journal of Natural Sciences*, vol. 10, no. 1, pp. 51–55, Jan. 2005, ISSN: 1993-4998. DOI: 10.1007/BF02828616.

[12] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite”, in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[13] D. T. Marr, F. Binns, D. L. Hill, *et al.*, “Hyper-Threading Technology Architecture and Microarchitecture”, en, *Intel Technology Journal*, vol. 6, no. 1, p. 1, Feb. 2002, ISSN: 1535-864X.

[14] D. Koufaty and D. Marr, “Hyperthreading technology in the netburst microarchitecture”, *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Mar. 2003, ISSN: 1937-4143. DOI: 10.1109/MM.2003.1196115.

[15] R. Preston, R. Badeau, D. Bailey, *et al.*, “Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading”, in *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.02CH37315)*, vol. 1, Feb. 2002, 334–472 vol.1. DOI: 10.1109/ISSCC.2002.993068.
[16] P. P. Ramon, “Caching in real-time and embedded systems: Benchmarking the arm cortex-m3 and quark soc x1000 processors”, Bsc Thesis, University College Cork, Ireland, Apr. 2015.
[17] V. L. Pisacane, *Fundamentals of Space Systems*. Oxford University Press, 2005, ISBN: 978-0-19-516205-9.
[18] J. R. Wertz and W. J. Larson, *Space Mission Analysis and Design*. Springer Netherlands, Sep. 1999, ISBN: 978-0-7923-5901-2.
[19] *Onboard Computers*, https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Onboard_Computers.
[20] A. Spector and D. Gifford, “The space shuttle primary computer system”, *Communications of the ACM*, vol. 27, no. 9, pp. 872–900, Sep. 1984, ISSN: 0001-0782. DOI: 10.1145/358234.358246.
[21] J. Culver, *The CPUs of Spacecraft Computers in Space*, https://www.cpushack.com/space-craft-cpu.html, Apr. 2012.
[22] M. Anderson, *Just What Do You Think You’re Doing, Dave?*, https://spectrum.ieee.org/just-what-do-you-think-youre-doing-dave, Jun. 2008.
[23] K. A. LaBel, “Decline in Radiation Hardened Microcircuit Infrastructure”, Tech. Rep., May 2015.
[24] *Victor Hess discovers cosmic rays | timeline.web.cern.ch*, https://timeline.web.cern.ch/victor-hess-discovers-cosmic-rays-0.
[25] *Testing at the Speed of Light: The State of U.S. Electronic Parts Space Radiation Testing Infrastructure*, in collab. with Committee on Space Radiation Effects Testing Infrastructure for the U.S. Space Program, National Materials and Manufacturing Board, Division on Engineering and Physical Sciences, and National Academies of Sciences, Engineering, and Medicine. Washington, D.C.: National Academies Press, Jun. 8, 2018, ISBN: 978-0-309-47079-7. DOI: 10.17226/24993.
[26] T. K. Gaisser, *Cosmic Rays and Particle Physics*. Jan. 1, 1990.
[27] W. Schimmerling, *The space radiation environment: An introduction.*
[28] A. S. Teitel. “Apollo Rocketed Through the Van Allen Belts”, Popular Science. (), [Online]. Available: https://www.popsci.com/blog-network/vintage-space/apollo-rocketed-through-van-allen-belts/ (visited on 07/13/2023).
[29] R. A. English, R. E. Benson, J. V. Bailey, and C. M. Barnes, “Apollo experience report: Protection against radiation”, NASA-TN-D-7080, Mar. 1, 1973.
[30] J. W. Howard, *Spacecraft Environments Interactions: Space Radiation and Its Effects on Electronic Systems*. NASA, 1999, 36 pp. Google Books: Wv4UAQAAIAAJ.
[31] M. Marinella and H. Barnaby, “Total Ionizing Dose and Displacement Damage Effects in Embedded Memory Technologies (Tutorial Notes - Draft 1).”, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), SAND2013-4379C, May 1, 2013.
[32] R. Baumann, “Soft errors in advanced computer systems”, *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005, ISSN: 1558-1918. DOI: 10.1109/MDT.2005.69.
[33] IBM Microelectronics Division, *IBM PowerPC 740/PowerPC 750 RISC Microprocessor User’s Manual*.
[34] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, “The gem5 Simulator: Version 20.0+”, 2020. DOI: 10.48550/ARXIV.2007.03152.

[35] S Ainsworth and T. M. Jones, “Research data supporting ”parallel error detection using heterogeneous cores””, 2018. DOI: 10.17863/CAM.21857.

Technical
University of
Denmark

Elektrovej, Building 327
2800 Kgs. Lyngby
Tlf. 4525 1700

www.space.dtu.dk