UNIVERSITY OF THESSALY

PHD THESIS

System software techniques to enhance reliability of modern platforms.

Author:
Konstantinos PARASYRIS

Supervisor:
Nikolaos BELLAS

Advising committee:
Nikolaos BELLAS,
Spyros LALIS,
Christos D. ANTONOPOULOS

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Department of Electrical and Computer Engineering

October 17, 2018

"Optimism is the faith that leads to achievement. Nothing can be done without hope and confidence."

"Helen Keller"

Abstract

Konstantinos PARASYRIS

System software techniques to enhance reliability of modern platforms.

Chip manufacturers introduce redundancy at various levels of CPU design to guarantee correct operation even for worst-case combinations of non-idealities in process variation and system operating conditions. This redundancy is implemented partly in the form of wide voltage margins.

This PhD dissertation is based on the concept that these conservative design runes are mostly excessive, as they account for execution scenarios that rarely appear during the lifetime of the systems. If the faults are ignored the system will result to application crashes or even system-wide failures. Our software based approach treats these faults to enable execution in such conditions.

The approach is based on the concept that many applications domains it is not the exact output that matters but a rough estimation of the output. Therefore, we propose a programming model in which the developer can define which parts of the application are more significant than others. The programming model extends a widely used parallel programming model, called OpenMP. The developer provides information about the *significance* of computations and the programming model exposes a parameter, called *ratio*, which can control the extend of quality degradation and energy efficiency.

The idea of significance-aware computing is ported into two different computing paradigms, the fault tolerant one and the approximate. In the case of fault tolerant computing we implement a significance-centric programming model and runtime support which sets the supply voltage in a multicore CPU to sub-nominal values to reduce the energy footprint and provide mechanisms to control output quality. The developers specify the *significance* of application tasks respecting their contribution to the output quality and provide check and repair functions for handling faults. On a multicore system we evaluate our approach using an energy model which quantifies the energy reduction. When executing the least significant tasks unreliably, our approach leads to 20% CPU energy reduction with respect to a reliable execution and has minimal quality degradation.

In the case of approximate computing, we implement a similar programming model that promotes the combination of the *significance* and *ratio* features. The approach using analytical models of the energy consumption of the application can

efficiently decide the degree of approximation to meet certain user defined energy requirements.

There are many application domains which require their computations to be performed without any errors. Our study on the x86-64 Haswell and Skylake multicore microarchitectures reveals wide voltage which can be removed without observing any errors. These margins can reach up to 22% and 13% of the nominal supply voltage for the Skylake and Haswell architectures respectively. The margins vary across different microarchitectures, different chip parts of the same microarchitecture, and across different workloads.

We introduce a model which can be used dynamically to adjust the supply voltage of modern multicore x86-64 CPUs to just above the minimum required for safe operation. We identify a set of performance metrics – directly measurable via performance monitoring counters – with high predictive value for the minimum tolerable supply voltage (V_{min}). We use benchmarks that vary in terms of application domain, resource utilization and pressure, and software/hardware interaction characteristics to train a V_{min} prediction model. Finally, at execution time those metrics are monitored and serve as input to the model, in order to predict and apply the appropriate V_{min} for the workload. Compared to the conventional approaches, our methodology achieves up to 42% energy savings for the Skylake family and 34% for the Haswell family for complex, real-world applications.

Last but not least, during the course of the thesis we implemented the infrastructure to observe accurately the application resiliency of faults as well as to identify voltage and frequency margins of modern processors. GemFI is a fault injection tool based on the cycle accurate full system simulator Gem5. GemFI provides fault injection methods and is easily extensible to support future fault models. It also supports multiple processor models and ISAs and allows fault injection in both functional and cycle-accurate simulations. GemFI offers fast-forwarding of simulation campaigns via checkpointing. Moreover, it facilitates the parallel execution of campaign experiments on a network of workstations. XM^2 enables the evaluation of software on systems operating outside their nominal margins. It supports both bare-metal and OS-controlled execution using an API to control the fault injection procedure and provides automatic management of experimental campaigns.

Περίληψη

Κωνσταντίνος Παρασύρης

Αύξηση ανθεκτικότητας των εφαρμογών σε σφάλματα σε
σύγχρονες πλατφόρμες

Η βιομηχανία των ημιαγωγών έχει βασιστεί τις τελευταίες δεκαετίες στο νόμο του Moore, ο οποίος προβλέπει κάθε 18 μήνες τον διπλασιασμό του αριθμού των transistors ανά μονάδα επιφάνειας σε ολοκληρωμένα κυκλώματα βασισμένα σε τεχνολογία CMOS. Σε αντιδιαστολή με το παρελθόν (πριν από το 2004), όπου οι σχεδιαστές επεξεργαστικών συστημάτων, μέσω της αύξησης του αριθμού των transistor, είχαν ως στόχο την αντίστοιχη αύξηση της απόδοσης, η νέα πραγματικότητα θέτει την μείωση της κατανάλωσης ισχύος (και ενέργειας) ως την μεγαλύτερη πρόκληση στην σχεδίαση επεξεργαστών.

Ταυτόχρονα, η μεγάλη πυκνότητα τοποθέτησης των transistors οδηγούν στην αναξιόπιστη λειτουργία των σύγχρονων επεξεργαστών. Η αναξιοπιστία αυτή οφείλεται εν μέρει στις δυναμικές διακυμάνσεις ρεύματος και τάσης (supply voltage) οι οποίες είναι πιο πιθανό να δημιουργήσουν λάθη χρονισμού σε μικρές γεωμετρίες τεχνολογίας CMOS. Επίσης είναι πιο πιθανά τα κατασκευαστικά λάθη (fabrication faults) λόγω ατελειών της διαδικασίας φωτολιθογραφίας. Επιπλέον, παροδικά λάθη (transient faults) που οφείλονται σε εξωγενείς παράγοντες, όπως alpha particles, έχουν μεγαλύτερη επίδραση σε μικρότερες γεωμετρίες τεχνολογίας CMOS. Για να επιτευχθεί αξιόπιστη λειτουργία υπό αυτές τις συνθήκες, οι σχεδιαστές σύγχρονων επεξεργαστικών συστημάτων χρησιμοποιούν συντηρητικές σχεδιαστικές τεχνικές, όπως υψηλά περιθώρια τάσης τροφοδοσίας (Vdd) και συχνότητας ρολογιού έτσι ώστε ο επεξεργαστής να προστατεύεται από κάθε πιθανότητα λαθών χρονισμού. Οι συντηρητικές αυτές τεχνικές μπορεί μεν να προστατεύουν την αξιόπιστη λειτουργία του επεξεργαστή, έχουν όμως ως αποτέλεσμα μεγάλη σπατάλη σε ισχύ και ενέργεια η οποία φτάνει μέχρι και το 35% σε αρκετές περιπτώσεις.

Η βασική ιδέα της παρούσας διδακτορικής διατριβής βασίζεται στο ότι αυτές οι συντηρητικές τεχνικές σχεδιασμού είναι σχεδόν πάντα περιττές και αντιστοιχούν σε περιπτώσεις λειτουργίας που σχεδόν ποτέ δεν πρόκειται να συμβούν ταυτόχρονα κατά την διάρκεια της λειτουργίας του επεξεργαστή. Χρησιμοποιώντας τεχνικές κυρίως στο επίπεδο λογισμικού συστήματος και εφαρμογών, η διατριβή προτείνει την λειτουργία του επεξεργαστή πολύ κοντά στις ακραίες καταστάσεις λειτουργίας του και την εξάλειψη του μεγαλύτερου μέρους του σχεδιαστικού περιθωρίου. Για παράδειγμα, η δυναμική μείωση της τάσης τροφοδοσίας ενός επεξεργαστή κατά την διάρκεια λειτουργίας του μπορεί να επιφέρει μεγάλες βελτιώσεις στην κατανάλωση ισχύος του, αλλά, εφόσον δεν ελεγχθεί, είναι δυνατόν να δημιουργήσει λανθασμένα αποτελέσματα ή και να διακόψει απότομα την

λειτουργία του. Από την άλλη, η αύξηση της συχνότητας του ρολογιού ενός επεξεργαστή, μπορεί μεν να βελτιώσει την απόδοση και να επιφέρει μείωση του χρόνου εκτέλεσης, αλλά, μπορεί να δημιουργήσει προβλήματα στην αξιοπιστία της λειτουργίας του.

Η διατριβή βασίζεται στην ιδέα ότι σε πολλές εφαρμογές (ή επιμέρους φάσεις εφαρμογών) το ακριβές αποτέλεσμα είτε δεν μας ενδιαφέρει, είτε είναι πολύ απαιτητικό σε κύκλους μηχανής και κατανάλωση ενέργειας για να μας συμφέρει να υπολογιστεί. Προτείνουμε ένα νέο προγραμματιστικό μοντέλο στο οποίο ο προγραμματιστής μπορεί να χαρακτηρίσει την σημαντικότητα των διαφορετικών τμημάτων μιας εφαρμογής και την συνεισφορά τους στην ποιότητα του τελικού αποτελέσματος. Το προγραμματιστικό μοντέλο επεκτείνει το γνωστό μοντέλο OpenMP που χρησιμοποιείται ευρέως στον παράλληλο προγραμματισμό. Μέσω της πληροφορίας της σημαντικότητας που παρέχει ο προγραμματιστής σε επίπεδο πηγαίου κώδικα (source code) και μιας επιπλέον παραμέτρου, που ονομάζεται ratio ο προγραμματιστής μπορεί και ελέγχει την αναλογία της μείωσης της ενέργειας προς την μείωση της ποιότητας του αποτελέσματος.

Η διατριβή υλοποιεί την παραπάνω ιδέα σε δύο διαφορετικούς τομείς υπολογισμών, τον προσεγγιστικό (approximate) τομέα και στον τομέα των ανθεκτικών υπολογισμών σε σφάλματα (fault tolerant). Στον τομέα τον ανθεκτικών υπολογισμών, υλοποιείται η υποδομή του λογισμικού συστήματος (προγραμματιστικό μοντέλο και σύστημα χρόνου εκτέλεσης) για να καλύψει περιβάλλοντα αναξιόπιστης υπολογιστικής. Η υλοποιήση αυτή αναφέρεται κυρίως στο ότι σε αναξιόπιστα περιβάλλοντα που μπορεί να προκαλούνται απο χαμηλότερη τάση τροφοδοσίας ή/και υψηλότερη συχνότητα ρολογιού μπορεί να συμβεί οποιοδήποτε λάθος στο υλικό το οποίο είναι πέρα από τον έλεγχο του προγραμματιστή. Αυτό επιβάλλει την ύπαρξη μηχανισμών αναγνώρισης και διόρθωσης λαθών καθώς και μηχανισμούς απομόνωσης λαθών ώστε αυτά να μην επεκταθούν σε σημεία του κώδικα που μπορεί να είναι κρίσιμα για την σωστότητα της εφαρμογής. Οι μηχανισμοί αυτοί θα πρέπει να έχουν όσο το δυνατόν υψηλότερο ποσοστό ανίχνευσης λαθών, αλλά και μικρή επιβάρυνση στην απόδοση της εφαρμογής.

Η διατριβή προτείνει και υλοποιεί αναλυτικά μοντέλα απόδοσης και κατανάλωσης ενέργειας σε πολυπύρηνους επεξεργαστές. Πολύ σημαντική επίσης είναι και η αναλυτική μοντελοποίηση των λαθών χρονισμού (fault modeling) που συμβαίνουν σε έναν επεξεργαστή σε συνάρτηση με την τάση τροφοδοσίας του επεξεργαστή αυτού. Το πλήρες σύστημα λογισμικού αξιόπιστης υπολογιστικής, μαζί με τα μοντέλα αυτά, χρησιμοποιούνται σε εκτεταμένες προσομοιώσεις εφαρμογών στον προσομοιωτή GemFI για την εκτίμηση της βελτίωσης κατανάλωση ενέργειας (λόγω μειωμένης τάσης τροφοδοσίας) χωρίς αλλοίωση των τελικών αποτελεσμάτων του υπολογισμού και χωρίς διακοπή της λειτουργίας του επεξεργαστή. Τα πειράματα έδειξαν ότι μπορούμε να μειώσουμε την τάση τροφοδοσίας ενός επεξεργαστή κατά μέσο όρο 15% μέχρι να φτάσουμε στο Point of First Failure (PoFF), κάτω από το οποίο ο επεξεργαστής εκτίθεται σε μαζικά λάθη χρονισμού και είναι αδύνατη (ή ασύμφορη) κάθε προσπάθεια λειτουργίας του. Το σύστημα λογισμικού αξιόπιστης υπολογιστικής επιτρέπει την μείωση της τάσης τροφοδοσίας μέχρι το PoFF (αλλά όχι πιο χαμηλά) αντιμετωπίζοντας επιτυχώς τα λάθη που εμφανίζονται στην περιοχή αυτή και επιτυγχάνοντας μεγάλη μείωση της κατανάλωσης ισχύος/ενέργειας χωρίς

επίπτωση στην ταχύτητα του επεξεργαστή.

Όσον αφορά τον προσεγγιστικό τομέα, προτείνουμε και υλοποιούμε ένα παρόμοιο προγραμματιστικό μοντέλο και ενα σύστημα χρόνου εκτέλεσης (run time system), το οποίο χρησιμοποιώντας αναλυτικά μοντέλα της απόδοσης και κατανάλωσης ισχύος του επεξεργαστή παίρνει ενημερωμένες αποφάσεις για τη ροή και το συντονισμό της εκτέλεσης του προγράμματος. Ο μηχανισμός αυτός επιτρέπει στον προγραμματιστή να προσαρμόσει δυναμικά την ποιότητα των αποτελεσμάτων μιας εφαρμογής με την απόδοση του συστήματος και την κατανάλωση ισχύος/ενέργειας.

Για πολλες εφαρμογές όμως, είναι απαραίτητοι οι ακριβείς υπολογισμοί καθώς δεν ανέχονται μείωση της ποιότητας του αποτελέσματος. Επομένως, μέσω μιάς σειράς πειραμάτων σε Intel x86-64 επεξεργαστές αναγνωρίσαμε την μέγιστη μείωση της τάσης τροφοδοσίας έτσι ώστε ο επεξεργαστής να παραμένει σε ασφαλή ζώνη λειτουργίας (χωρίς σφάλματα). Τα πειράματα αυτά φανέρωσαν ότι τα περιθώρια μείωσης της τάσης φτάνουν μέχρι 22% για επεξεργαστές Skylake και 13% για Haswell.

Η παραπάνω ανάλυση των περιθωρίων τάσης έδωσαν την δυνατότητα να δημιουργηθεί ένα μοντέλο μηχανικής μάθησης που χρησιμοποιήθηκε για την πρόβλεψη μελλοντικών περιθωρίων τάσης τροφοδοσίας. Το μοντέλο αυτό δέχεται σαν είσοδο τιμές των performance counters του επεξεργαστή (ενώ το πρόγραμμα εκτελείται) και προβλέπει το περιθώριο τάσης για το επόμενο χρονικό διάστημα. Με άλλα λόγια, η διατριβή αυτή έχει δείξει πειραματικά την συσχέτιση μεταξύ γεγονότων σε επίπεδο μικροαρχιτεκτονικής (όπως αυτά περιγράφονται από τους performance counters του επεξεργαστή) και του μέγιστου περιθωρίου τάσης τροφοδοσίας. Το μοντέλο αυτό χρησιμοποιείται από έναν εξελιγμένο governor τάσης/συχνότητας που ορίζει κάθε φορά την τάση του επεξεργαστή ώστε να επιτυγχάνουμε μεγάλη μείωση στην κατανάλωση ισχύος χωρίς τον κίνδυνο μειωμένης αξιοπιστίας. Σε σύγκριση με τον συμβατικό Intel DVFS (Dynamic Voltage Frequency Scaling) governor, η μέθοδός μας επιτυγχάνει μείωση ενέργειας κατά 42% σε επεξεργαστές Skylake και 34% σε επεξεργαστές Haswell.

Η διατριβή αυτή επίσης μελέτησε την επίδραση που έχουν στα περιθώρια τάσης τροφοδοσίας οι βελτιστοποιήσεις που κάνουν οι μεταγλωττιστές στον κώδικα μιας εφαρμογής. Με τη χρήση του εργαλείου XM2 διαπιστώθηκε ότι οι μετασχηματισμοί που μειώνουν τις προσπελάσεις στην κύρια μνήμη έχουν μεγάλη επίδραση στα περιθώρια της τάση τροφοδοσίας. Σε γενικές γραμμές όμως, η τάση τροφοδοσίας είναι περισσότερο συνάρτηση της μικροαρχιτεκτονικής του επεξεργαστή παρά των βελτιστοποίησεων στον κώδικα που επιφέρει ο compiler.

Η διδακτορική διατριβή δημιούργησε επίσης την ερευνητική υποδομή για τα πειράματα που διεξήχθησαν στα πλαίσια της αξιολόγησης των τεχνικών που περιγράφησαν στις προηγούμενες παραγράφους. Το GemFI είναι ένα εργαλείο δημιουργίας και προσομοίωσης λογικών λαθών σε ένα πολυπύρηνο επεξεργαστικό σύστημα. Δίνει την δυνατότητα στο χρήστη να εισάγει διάφορους τύπους λαθών, σε οποιαδήποτε χρονική στιγμή κατά τη διάρκεια εκτέλεσης του προγράμματος, και σε οποιοδήποτε τμήμα της αρχιτεκτονικής του επεξεργαστή. Το GemFI, το οποίο βασίζεται στο γνωστό προσομοιωτή Gem5, μπορεί να εκτελέσει πλήρως το λογισμικό εφαρμογών και συστήματος για μεγάλη γκάμα

διαφορετικών αρχιτεκτονικών, δίνοντας έτσι τη δυνατότητα μελέτης της αξιοπιστίας του συστήματος κάτω απο διαφορετικές συνθήκες.

Επίσης δημιουργήθηκε ένα δεύτερο εργαλείο, ο eXtended Margins eXperiment Manager (XM2) που αυτοματοποιεί τη δημιουργία πειραμάτων σε πραγματικούς επεξεργαστές που μπορούν να τίθενται σε μη-αξιόπιστες καταστάσεις τάσης και συχνότητας. Το εργαλείο χρησιμοποιήθηκε για την δημιουργία τέτοιων πειραμάτων για τον χαρακτηρισμό της αξιοπιστίας του υλικού και για τον προσδιορισμό της ανοχής σε λάθη λογισμικού εφαρμογών που εκτελούνται σε επεξεργαστές x86-64 (Skylake, Haswell) και ARM.

Acknowledgements

First and foremost I would like to thank my mentors, professors Nikolaos Bellas, Spyros Lalis, and Christos D. Antonopoulos from the University of Thessaly. They have been exceptionally good at guiding me during my initial steps, throughout my Master's and PHD, and have molded me as a researcher. Without their guidance and mentoring none of this work would be possible. They were always available to discuss and provide constructive criticism.

I would also like to thank my colleagues who provided me with help, as well as stress relief with their wit and humor to ease the burden of research. Special thanks to my friend and colleague Vassilis Vassiliadis with whom I shared, pretty much all of my research career thus far. Our joined research efforts, stimulating, and often heated, discussions were very educating and most of the time relaxing.

Last but not least, I owe great many thanks to my friends and family. Especially, I owe to my parents, Wyanda And Antonis, for their unconditional love and support all along my academic pursuits. My friends will always have a special place in my heart because they were always there during good times, and bad times to support me with patience and love. The least I can do is dedicate this thesis to them all.

Contents

List of Figures

List of Tables

Dedicated to my family and
friends

Chapter 1

Introduction

The scalability of semiconductor manufacturing process, as predicted by Moore's law, has been the driving force of the increase in the capabilities of computer systems. The 2013 International Technology Roadmap for Semiconductors (ITRS) report warns the public that CMOS, due to their constant scaling, will reach their atomistic and quantum mechanical physics boundaries in the next 3-12 years [47].

The scientific community is motivated to investigate novel designs which exploit heterogeneous technologies and emerging new information-processing paradigms. In contrast to the past, when technology was driven by the quest for higher performance, the primary goal of systems has shifted to optimize the power consumption/performance equilibrium.

1.1   The reality of power consumption

For over four decades Moore's Law, coupled with Dennard scaling [91] ensured the exponential performance increase in every process generation through device, circuit, and architectural advances. Up to 2005, Dennard scaling meant increased transistor density with constant power density. If Dennard scaling would have continued by the year 2020 [67], we would have approximately 40 times increase in energy efficiency compared to 2013. Unfortunately, Dennard scaling has ended because of the slowdown of voltage scaling due to slower scaling of leakage current.

The end of Dennard scaling has changed semiconductor industry dramatically. To continue the proportional scaling of performance and exploit Moore's Law, processor designers have focused on building multicore systems and servicing multiple tasks in parallel instead of building faster single cores. Even so, limited voltage scaling increasingly results in having a larger fraction of a chip unusable, commonly refer to as Dark Silicon [28]. It is expected by 2020 that only nine percent of the total number of transistors could be activated at any point in time due to tight power budgets [67].

To make things even worse, static and dynamic variations [13] result in adding voltage guardbands to ensure correct processor operation. The added guardbands increase energy consumption and force operation at higher voltage or lower frequency. They may also result in lower yield or field returns if a part operates at

higher power than its specification allows. The guardbands are becoming more prominent with area scaling, the use of more cores per chip, and core to core variations. The average power cost of these guardbands can be in the order of 35% [22]

For all these reasons, power consumption nowadays is a prime design parameter. If we need to go faster, we need to find ways to become more power efficient. For example, one of the major concerns of building the new generation of High Performance Computing (HPC) systems and overcoming the exascale barrier is power consumption. Such systems ideally would consume approximately 20 Megawatts and would maximize their performance by achieving greater degrees of parallelism. All other things being equal, if one design uses less power than another, then it has headroom to improve performance by using more resources or operating at a higher frequency. Simply put, a more energy efficient chip has headroom to provide more functionality and to service more tasks at given time frame.

1.2 Reliability and Power

Hardware designers go to great lengths to improve hardware reliability. They use guardbands in their designs against adverse combinations of factors that affect hardware reliability. This conservative design methodology essentially results in area, performance, power and energy overheads. Such design choices though, are not unreasonable as computation accuracy and hardware reliability have traditionally been primary concerns during the design of computing systems. After all, developers expect hardware to always behave in a reliable and predictable way. In the event that a hardware fault arises and manifests as an error in the software level it is treated as a rare scenario with developers actively spending effort to mask the errors from the user space, regardless of the magnitude of its effects.

In the terminology used in dependable and reliable operation of computer systems, a fault is an incorrect value within the internal state of the design of a system. A fault in the hardware level can result into errors and an error can result into a failure.

Possible sources of hardware unreliability are voltage droops, transistor variability, aging, temperature, or even alpha particles temporarily affecting hardware functionality. The traditional design technique addressing this problem also known as guardbanding, usually involves a combination of techniques, such as: i) higher supply voltage levels (voltage margins) ii) larger transistors iii) logic for error detection and correction iv) spare cells. Although, guardbands have successfully ensured correct operation up to date, their effectiveness in detecting and correcting all errors is questioned by researchers, as geometries and supply voltages are scaled down and circuits become more vulnerable to failures [124, 20, 111].

The extent of guardbanding that may be necessary to protect circuits against

all potential errors will lead to significant power overheads not sustainable by future systems, thus conflicting with power dissipation which is another major challenge of the semiconductor industry. Note that these guardbands are pessimistic, as they have to compensate for the worst case scenarios and combinations of nondeterminism, switching patterns, temperature and aging effects. According to [22] the average power cost of guard bands is roughly 35%. However, most of the time, these guard bands represent mere overhead, as worst case scenarios and combinations will appear very seldom during application execution. The main reason for having these pessimistic guard bands and energy inefficiency is that modern computing systems execute programs under strict correctness requirements.

Guardbanding not only increases the power consumption of hardware but it implicitly limits its performance as well. For example, [28] warns that regardless of chip organization and topology, multicore scaling is power limited. A side-effect of this issue is that at the 7nm process node, more than 50% of the transistors in a general purpose processor will have to be powered off in every cycle. This is a trend that will be visible at larger scale as well. Even though, it will be possible to fit thousands of cores in the die it will be impossible to activate simultaneously more than a few tens or low hundreds [38].

The issue at hand is quite alarming. Even if future applications have the inherent parallelism to make efficient use of thousands of cores, the performance of our computing systems is going to be restricted due to the extreme power dissipation. In other words, unless we manage to design novel architectures, technology is bound to hit again the same power wall as single core architectures.

1.3 Design Space exploration

Assuming that the quality of the output is related to the reliability of the system, there is a need to rethink the design flow and propose abstractions that shift the reliability versus energy dissipation balance towards a design philosophy in which errors may be allowed to happen and safely ignored. To this direction one should consider and exploit opportunities presented at the hardware level as well as at the software level.

1.3.1 Hardware Level

In figure 1.1 we present the possible operating points of a generic CPU. We briefly describe four different operational regions and discuss their trade-offs between energy and reliability:

Nominal Operating Points. Typically, a CPU manufacturer defines a finite number of operating points, called Nominal Operating Points (NOP) which guarantee error-less operation. These settings are the conventional operating points of modern CPUs. For the increased reliability the system designs needs to pay the overhead of the guardbands.
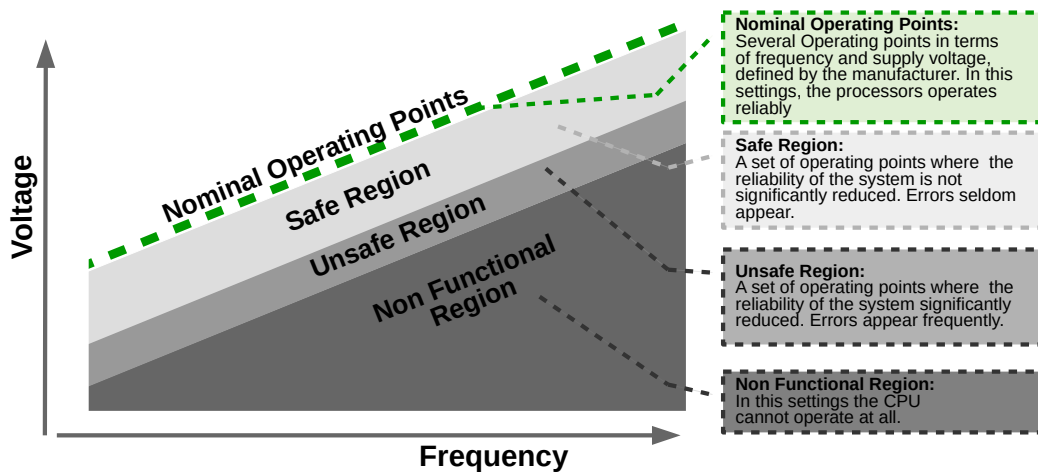
FIGURE 1.1: Description of the reliability of a generic CPU for different operating points in terms of supply voltage and frequency.

Safe Region. One can remove the voltage margins imposed by the guardbands, by decreasing the supply voltage, also referred as undervolting, or by increasing the operating frequency, also referred as overclocking, without reducing significantly the reliability of the system [95, 4, 5, 94, 74, 151]. In this region errors appear due to irregularities, which may appear only rarely or even not at all during the life cycle of a given processor. The energy reduction mainly depends on the magnitude of the margins.

Unsafe Region. It is possible, to perform even more aggressive undervolting or overclocking and operate in the unsafe region, where hardware starts to malfunction and produce timing errors during the execution time of a workload [11, 26, 22]. Since the voltage reduction is higher in comparison with the *safe* region, the energy gains are larger. However in these settings the errors occur frequently, therefore there is a need to add some protection mechanisms against these errors.

Non Functional Region. At this region the CPU is not operational at all as the majority of the paths within the netlist of the system experience timing violations.

1.3.2 Software level

For a lot of applications, not all computations and not all data of the application are equally critical, requiring to be performed or maintained at 100% accuracy or correctness. Several application domains offer the opportunity to trade-off quality of output(QoO) for significant improvements in energy consumption. For such applications, it may be possible to only approximate the final output (or part of it), rather than computing the exact output result.

Typically such applications share a common property: they have relaxed accuracy constraints. In other words, they can accept a range of possible values as "correct". Figure 1.2 lists a few examples of application domains which have intrinsic

Statistical & Probabilistic Computations (monte carlo)

Streaming Nature

(Too) Many, Noisy Measurements (big data)

Potential to Approximate

Self-Healing (iterative methods)

Perceptual Limitations (visualization, multimedia, etc)

Computational Redundancy (communications)
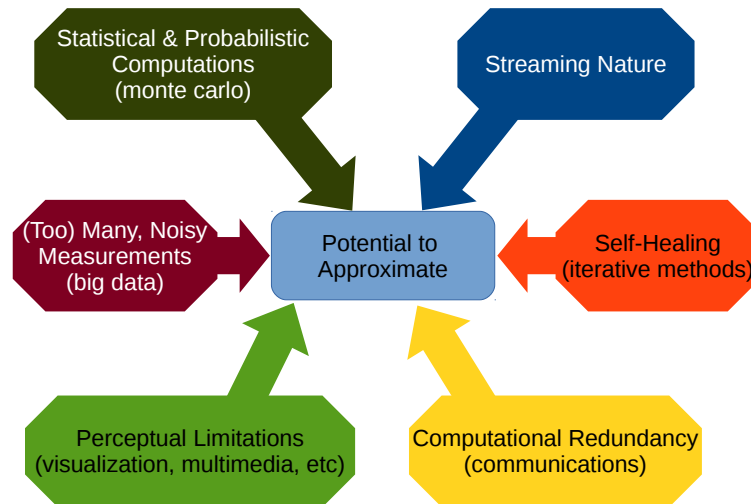
FIGURE 1.2: Application domains with intrinsic error resiliency.

error resiliency and are consequently approximation friendly. For example:

Visualization applications are amenable to approximations because their output is typically consumed by humans. Application developers can exploit the perceptual limitations of the human eye to approximate computations without inflicting noticeable quality degradation to their output.

Streaming applications are inherently amenable to approximations since they do not maintain a large state. They consume input data, perform computations, and produce output data. If an error occurs during the computation of a specific output data batch, the next batch will not be severely affected. In that sense, streaming applications inherently exhibit computational isolation.

Some **iterative methods**. For example, in the presence of errors iterative numerical methods still tend to converge to a correct solution but will typically require more iterations.

Randomized computations, If errors are random, the effect on these algorithm is part of the randomization process.

Approximate computing is an emerging paradigm, that allows a controlled decrease of the quality of the output for energy efficiency [6, 121, 149, 89]. Typically, approximate computing is used as a term to describe the disciplined aggressive optimization at the algorithmic level to gracefully trade-off computation accuracy with performance/energy efficiency.

A similar notion to approximate computing is *fault tolerant computing* in which the underlying hardware may exhibit unexpected behaviour such as computation errors, crashes, or even infinite loops. This uncertainty/unreliability can be the result of a wide variety of causes. It may be due to unreliable execution on energy efficient substrates, or even hostile environments such as space (alpha particles, cosmic rays, solar wind flux, etc). In any case, maintaining acceptable levels of reliability and maintaining an acceptable quality of the output while executing code using unreliable hardware requires the use of fault tolerance techniques. Although both

paradigms exploit similar features to achieve energy efficiency they demonstrate large differences. Approximate computing is controlled and predictable. A developer consciously designs an approximate version of the code and/or uses hardware with reduced precision. On the other hand, execution under unreliable conditions is less predictable and typically requires the application of fault tolerant computing techniques. For example, the developer should detect and correct errros before they irreversibly contaminate the application state.

One factor that contributes to the energy footprint of current computer technology is that all parts of the program are considered to be equally important, and thus are all executed with full accuracy. However, as shown by previous work [73], in several classes of computations, not all parts or execution phases of a program affect the quality of its output equally. In fact, the output may remain virtually unaffected even if some computations produce incorrect results or fail completely. Significance-aware computing [61, 90] exploits the algorithmic property of computational significance to create optimization opportunities in terms of performance and power-efficiency of applications.

1.3.3 Significance Definition

A formal definition of significance can be provided as follows. Assuming code that implements the function $y = f(\vec{x})$, where $\vec{x}$ is the vector of the function inputs, the significance of $\vec{x}$ to the output y can be defined using interval arithmetic [92] and first order adjoint analysis. The range of possible input values is the input interval vector $[\vec{x}] = [\underline{\vec{x}}, \overline{\vec{x}}] = \{\vec{x} \in I\!\!R^n | \underline{\vec{x}} \le \vec{x} \le \overline{\vec{x}}\}$, and an evaluation of f in interval arithmetic is obtained by replacing all variables and intermediate elementary functions ϕ with their interval version. The significance of an input element $x_i \in \vec{x}$ to the final result y is equal to

$$S_y(x_i) = w([x_i] \cdot \nabla_{[x_i]}[y])$$

where $w(\cdot)$ is the width of the interval. The first order derivative $\nabla_{[x_i]}[y] = \frac{\partial f[\vec{x}_i]}{\partial [x_i]}$ is the derivative of the function result $[y]$ with respect to the input variable $[x_i]$. In other words, the bounds of interval derivative $\nabla_{[x_i]}[y]$ are the steepest downward and upward slopes, respectively, of $y = f(\vec{x})$ in the interval $[x_i]$, which quantify the impact of all possible values from $[x_i]$ on the final result y. If the range (width) of S_y is large, then x_i strongly affects the value of y. As such, the code that produces the value of x_i is highly significant for the accuracy of the final output y. More information on the algorithmic property of significance and a methodology for determining the significance of computations automatically can be found in [141].

1.4 Contributions

The granularity of *significance* is not bounded in the spectrum of a single application, but it can be projected to the entire software stack. For example, the Operating
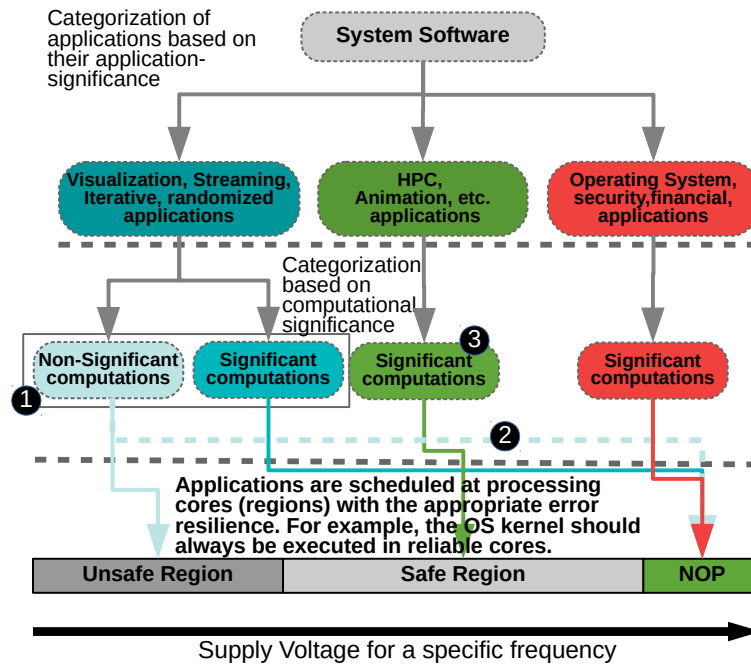
FIGURE 1.3: Vision of our approach. Applications should be categorized based on their contribution to the output quality. Using this information the applications should be scheduled in hardware with appropriate levels of reliability.

System (OS) is more important (significant) than a media application, as an error corrupting the OS could propagate to multiple applications making the entire system unstable. Likewise, a security application is more significant than a gaming application. Using a similar reasoning the quality of the output can be projected to the entire software stack, where the quality of the system takes into account all the applications within the system. This vision is depicted in figure 1.3.

This PhD dissertation aims at improving power and energy efficiency by exploiting the correctness/reliability opportunities presented both in the software level and the hardware level. Applications which can tolerate quality degradation use significance-driven approach, which aims to gracefully trade-off application output quality with improved energy efficiency by forcing the CPU processor to operate on an unsafe region (Figure 1.3 black circle 1).

The significance-driven approach can be also exploited in *approximate* computing, in which alternative more energy efficient approximate versions of the non-significant computations are scheduled for execution in Nominal operating points (Figure 1.3 black circle 2).

In any case though the significance-based computing model is not suited for all applications. For instance, the significance of computations may be highly input-dependent, hard to specify at design time and difficult or costly to extract even at run time. Also, some programs may require all tasks to be executed without inaccuracies. To this direction in this these we achieve energy efficiency by executing applications in the *safe* operational region of CPUs. In which errors are not expected to happen, but pose a rare event(Figure 1.3 black circle 3).

Finally, this PhD dissertation distributes the responsibility for energy efficiency and reliability in various design layers starting from the circuit all the way to the application layer, where available slack in output QoS can be reduced to increase energy efficiency. The realization of such a system requires modeling of the effects of potential hardware misbehaviour induced by various sources and also tools that can easily propagate such effects. To this direction, we designed, implemented and deployed experimental frameworks, which are used to study the trade-offs between energy/power efficiency and execution resilience (both in terms of quality of output and execution resilience).

The next sections describe the contributions in greater detail.

1.4.1 Significance Aware Computing

We investigate significance aware computing, based on the premise that specific phases of a computation may incur a high toll on performance and energy without the corresponding contribution to the quality of the result.

For example, Discrete Cosine Transform (DCT), a module of popular video compression kernels, which transforms a block of image pixels to a block of frequency coefficients, can be partitioned into layers of significance, owing to the fact that human eye is more sensitive to lower spatial frequencies, rather than higher ones. By explicitly tagging operations that contribute to the computation of higher frequencies as less-significant, one can leverage smart underlying system software to trade-off video quality with energy and performance improvements. Significance aware computing aims to exploit regions of an application which are amenable to aggressive optimizations that do not severely impact the final application outcome but lead to improvements in energy/power/performance.

There is need for an intuitive programming model which is user-friendly but also offers the necessary expressiveness and functionality to address all of the key challenges of the signifiance aware paradigm. To this direction, we introduce a programming model which extends the OpenMP, one of the most popular parallel programming models, and provides the following main features:

Significance Characterization: The developer can specify the significance of different parts of the computation based on how strongly each part contributes to the quality or correctness of the end-result.

Quality Control: On the one hand, the application should always terminate with results that are acceptable to the user. On the other hand, it should be elastic; different users may have varying expectations on the performance and output quality of an application. In fact, even the same user may need the same application for different execution scenarios to modularly adapt the quality/energy trade-offs. We introduce a single knob, called *ratio* that controls the proportion of computations which will be approximated or executed on unreliable

but more energy efficient hardware. The *ratio* feature can be used to effectively trade off quality for energy efficiency.

We introduce two different versions of the significance-aware programming model which correspond to (i) fault tolerant computing in unreliable hardware and (ii) approximate computing.

Significance aware fault tolerant programming model

We present a programming model which facilitates application execution on hardware within the *unsafe* region, hence multiple errors occur during the execution of an application. This necessitates mechanisms to isolate/protect unreliable computations from reliable ones, in conjunction with methods for detecting and correcting severe silent errors. Consequently, efficient significance-aware *fault tolerant* computing requires an intuitive and user friendly way to provide all of this information to an intelligent runtime system which will support and orchestrate the execution of code using mixed reliability hardware. The programming model besides the *significance* characterization and the *ratio* knob also facilitates the following features:

Sanity Control: The developer can introduce code for checking and repairing the output of the parts that are executed unreliably.

Relaxed Synchronization: Since errors may result to infinite loops or prolong the execution of the application, the programming model provides primitives to support relaxed synchronization which uses timing watchdogs to break from such rare occurrences.

Significance aware Runtime system for fault tolerant computing

Besides the expressiveness of the programming model, the runtime system should effectively support unreliable operation. Initially the runtime system should use the significance information combined with the *ratio* knob to decide which computations will be executed on nominal operating points and which computations will be scheduled for unreliable operation in the *unsafe* region. Furthermore, the runtime system should isolate the side-effects of the unreliable execution of non-significant parts, so that they do not silently propagate to the rest of the computation in an uncontrolled way. It also, should protect crucial information concerning the runtime state and the application from errors. Finally, the runtime system is responsible to hide hardware details from the programming model and consequently from the developer.

Our evaluation shows the effectiveness of different protection mechanisms provided by the runtime system. We show that traditional system software protection mechanisms are not adequate, however their combination with programmer wisdom provides effective protection against crashing and silent data corruptions,

while enabling considerable energy gains (on average 20%) . Interestingly, modern processors with the assistance of our framework can produce acceptable results until the error rate becomes too high. From that point, additional energy gains are too low, and massive failure rates defeat any software-based realistic protection mechanism.

Power/energy and fault modeling of the *unsafe* region

A key challenge is to model the operation of a core in the *unsafe* operating region. This PhD dissertation presents an analytical energy and power model of a CPU when operating in the *unsafe* region. Moreover, it associates the operation in this region with the probability of faults due to timing violations. The methodology presented combines simulation with software fault injection to emulate the unreliable operation and it uses the analytical models to estimate the energy consumption of the system in such settings.

Significance-aware programming model for approximate computing

A similar programming model is proposed that facilitates approximate computing (on reliable hardware). In this case, quality degradation is induced in the application due to the inaccuracies of the approximate versions. Besides the *significance* characterization and *ratio* the programming model should facilitate the following feature:

Approximation characterization: The programming model should provide the ability to define an approximate version of the computation. This version should be more energy efficient, but may also create a less accurate result.

Significance aware Runtime system for approximate computing

The runtime system should use the *significance* as well as the *ratio* value to decide for which computations the approximate version of the code should be used. The runtime system can select the *ratio* knob to maximize the quality of the output in energy constrained environments.

1.4.2 Exploiting Voltage Margins for Energy Efficiency

The significance aware programming models can be used to reduce the energy consumption assuming that the user has access to the source code of an application which is amenable to approximation. There are applications which can not sustain quality degradation or there is no access to the source code. However, as presented in Section 1.3.1 the hardware design already presents opportunities for energy efficiency without reducing the quality of the output. These opportunities are presented within the *safe* region of operation. To maximize the energy efficiency one should be able to predict the boundary between the *safe* and *unsafe* region and execute the applications just above this boundary.

To this end, we developed a model to predict a reduced but safe CPU supply voltage to achieve higher energy efficiency. The model is used by a dynamic voltage scaling governor which monitors the utilization of the CPU resources and uses a prediction model to apply a new safe supply voltage of the system. To train the prediction model we perform an offline characterization phase of the voltage margins, of two mainstream x86-64 microarchitectures, the Haswell i7–4790 and the Skylake Xeon E3–1220 v5 processors, using a diverse set of benchmarks which stress different components of the CPU microarchitecture. Interestingly, our analysis shows that the voltage margins depend on a multitude of factors.

The main contributions of our approach are:

- We develop a model that takes as input selected CPU performance counters and core utilization, and estimates the voltage margin of the workload on the specific CPU part for CPU base frequency. This estimation can be exploited to safely undervolt CPUs and achieve higher energy efficiency.

- We evaluate the effectiveness of our model using a dynamic voltage scaling governor, called xDVS. The governor instructed by our model dynamically adjusts the CPU supply voltage to levels below conservative nominal values. Compared to the stock Intel (P-state) DVFS governor, our approach achieves energy savings up to 40% for Skylake and 34% for Haswell CPUs.

- The model used by xDVS is verified via a long-running (consecutive 72 hours) and dynamically changing application workload on 6 different workstations, without any observable degradation of system reliability. Our approach proves able to handle cases where a single application monopolizes the CPU, but most importantly real-world scenarios, where a mixture of different applications execute simultaneously. To the best of our knowledge there is no prior work that supports dynamic undervolting for such realistic execution schemes. During this execution, the supply voltage for the Skylake and Haswell CPUs was adjusted a total of $2.6 * 10^6$ times, with the average amount of undervolting being $216mV$ and $116mV$ respectively.

We note that our dynamic V_{min} predictions are accurate and safe for conventional workloads. Our scheme still may require hardware fail-safe mechanisms to protect the CPU from worst-case, extreme voltage fluctuations, such as those induced by voltage droop viruses. Section 4.7 discusses this point in more detail. Moreover, in our evaluation we do not categorize applications as important or less important. All applications are executed in hardware operating in similar settings.

1.4.3 Experimental Frameworks for Reliability Analysis

Relaxing the hardware correctness requirements reduces the reliability of the system. A major challenge of this PhD dissertation was to understand under which circumstance do errors manifest on the hardware and how different applications mask the

respective errors. To be more precise we faced the following challenges:

- Understand how and under what circumstances modern CPU microarchitectures fail when executing code at reduced margins

- Identify the voltage margins of the system and associate them with their respective energy gains.

- Evaluate the resilience of applications – or the whole software stack

To overcome these challenges we developed the following two fault injection tools which where used by this PhD dissipation to conduct the experimentation:

GemFI: A simulation based fault injection tool. The primary objective of the tool is to enable fault injection based on different fault models and on systems with various configurations. A variety of different system configurations and architectures can be supported without affecting the implementation of fault injection in GemFI

XM^2: a fault injection tool on real hardware . It can be used to identify the margins of a system, or study the effect of real faults on the application and associate the undervolting/overclocking with the respective energy gains. We use XM^2 in several case studies in which we identify:

- The effect of common compiler optimizations on the energy efficiency and the frequency margins of four ARM Cortex-A53 processor parts;
- The effect of memory access pattern optimizations on the energy efficiency and the frequency margins;
- The interaction of SIMD instructions on the energy efficiency and the frequency margins.

1.5 Outline

In Chapter 2 we present the significance driven programming model for *fault tolerant* computing. We detail the challenges of *fault tolerant* computing and their solution through the programming model and the runtime system. We use an power-/time/energy model to estimate the energy footprint of applications when executed on the *unsafe* region. We also implement a fault injection methodology to simulate the errors that manifest during the execution in the *unsafe* region. Finally using these models we evaluate our programming model.

In Chapter 3 we present the approximate variant of our programming model and the accompanying runtime system. We evaluate our approach and present a case study, in which a model-based approach can automatically discern the appropriate levels of approximation and concurrency as well as CPU frequency for energy restricted execution of applications. An offline analysis drives the decisions of an

intelligent runtime system, which fine tunes the approximation degree of an application so that its energy consumption remains within a budget that is specified by the end-user at execution time.

In Chapter 4 we present a methodology to detect safe margins for multicore CPU operation. We create a prediction model to estimate a safe and energy efficient operating point within the *safe* region. The model is evaluated using a dynamic voltage scaling governor.

In Chapter 5 we present the experimental framework used by this work to model the various energy-reliability trade offs when operating on sub-nominal settings. We discuss related work in Chapter 6 and present our concluding remarks in Chapter 7.

Chapter 2

Significance Aware Fault Tolerant Computing

As already discussed, part of the energy inefficiency problem of modern computing systems is that all computations are treated as equally important, despite the fact that, in a lot of cases, only a subset of these computations may be critical to achieve an acceptable quality of service (QoS). In several application domains it is not the precise result that matters to the user but rather an approximation of the output [24, 33]. More generally, within each program all computations are treated as equally important for the quality of the end result, although in most cases this assumption is not true.

In this chapter we lay the foundations for not only approaching the theoretical limits of energy reduction of conventional technology, but also moving beyond those limits by accepting the quality degradation in a controlled manner. Applications or different phases of the same application have different *significance* for the quality of the end-result. System designers have traditionally made very strict assumptions about quality of the output, insisting that the application state and the processor state is completely accurate and correct. However, this requirement can be relaxed. For example the user can be satisfied with a lower but acceptable quality of the output. Even if the intermediate state of the system is not accurate. In the end, one may ask if strict, bit-exact application correctness is even required for some applications.

In any case though, the proposed significance-based computing model does not fit all applications. For instance, computation significance may be highly input-dependent, hard to specify at design time and difficult or costly to extract even at run time. Also, some programs may require all computations to be executed without any inaccuracy or any chance of data corruption. Our approach focuses on application domains that offer the opportunity to trade-off quality of output for significant improvements in energy consumption. Such applications domains include:

Visualization applications are amenable to approximations because their output is typically consumed by humans. The correction part of the result check function can exploit the perceptual limitations of the human eye to approximate

computations without inflicting noticeable quality degradation to their output. In our evaluation we use two benchmarks from this category: *DCT* and *Sobel*.

Streaming applications are inherently amenable to approximations since they do not maintain a large state. They consume input data, perform computations, and produce output data. If an error occurs during the computation of a specific output data batch, the next batch will not be severely affected. In that sense, streaming applications inherently exhibit computational isolation. *Blackscholes*, one of the benchmarks used in the experimental evaluation, falls in this category.

Iterative methods tend to be self healing. For example, in the presence of errors, Monte Carlo simulations, or iterative numerical methods still tend to converge to a correct solution but will typically require more iterations. Such applications in our evaluation are *Jacobi* and *K-means*.

2.1 Contributions

We present a programming model and the accompanying runtime system that elevates the *significance* of the computations as a first order constraint to the developers. We provide a single knob to gracefully trade-off the quality of the output to energy gains. To be more precise in this chapter we make the following contributions:

- We introduce a significance-centric programming model and runtime system, which allows computations to be executed on potentially unreliable but low-power hardware in a controlled way, to trade-off output quality for greater energy efficiency.

- We introduce a single knob, called ratio, to trade-off quality for energy gains in a flexible way. The ratio can be set at execution time, thereby allowing the user or a higher-level framework to control the energy/quality trade-off to meet dynamically varying application requirements.

- We present a power, energy and fault modeling methodology which simulates operation on the *unsafe* region.

- We evaluate our programming model and runtime system. Our approach effectively trade-offs quality of output to optimize the energy consumption of applications using the significance aware *fault tolerant* computing programming model. The evaluation also demonstrates the limitations of execution application on the unsafe region.

2.2 Programming Model Objectives & Properties

Our programming model can develop applications which target *unreliable* hardware, without uncontrolled degradation of the quality. The design objectives and properties of the programming model are the following:

2.2.1 Significance Characterization

The programming model should allow the developers to characterize computations according to their degree of significance in a straightforward and intuitive way. Significant computations will be executed correctly, at the expense of power consumption and/or performance, while computations characterized as non-significant will be executed in a way that may produce incorrect results. The significance of a code region should be expressed either statically, at compile-time, or at run-time, since significance may very well be input-related and/or context-dependent.

2.2.2 Safety Isolation

The model should be safe: Computations are by default considered significant and are thus executed correctly, unless the programmer explicitly allows imprecise computations. As a result, a silent data corruption on the output of an early part of the application can affect the execution of the following parts which are dependent on the faulty one. Such a scenario would obviously violate the isolation attribute. Our model should be equipped with mechanisms to tackle this situation by providing functionality to the developer to detect errors early and even correct the computed values of non-reliable computations. According to this scheme, the propagation of errors across the application, can be controlled/avoided.

Moreover, both the design of the programming model and its implementation should promote — and if possible guarantee — the isolation between significant and non-significant code regions. In other words, errors manifesting on non-significant tasks should not be fatal for significant tasks or the whole application. Although isolation is a desired property, it is not straightforward to guarantee it. Almost all realistic applications are characterized by data flow among their parts (such as objects, tasks, functions etc.).

2.2.3 Architecture Neutrality

The programming model should assume as little as possible about the underlying architecture, making applications portable with no or at least with reasonable effort to different architectures. It should be noted that our main focus is on functional portability. The power / performance / reliability balance is highly architecture dependent, therefore the power / performance / reliability efficiency is expected to differ when moving to different hardware.

2.2.4 Parallelism Expression

While the key objective of the programming model is to support significance-aware programming and execution, we also wish to be able to express and handle parallelism. This is important to exploit next-generation multi-core architectures, which are quite likely to include unreliable cores or allow cores to operate at power levels that may introduce faults.

2.2.5 Relaxed Synchronization

An important prerequisite for parallel execution of jobs is the existence of synchronization mechanisms to ensure correctness of the final result. Due to the uncertain behavior of jobs executing on unreliable hardware, traditional synchronization mechanisms are overly stringent and are therefore not a feasible option. To this direction we envision mechanisms offering more elastic synchronization, by extending synchronization constructs with timing watchdogs, and more flexible synchronization achievement criteria.

2.2.6 User Friendliness

A programming model must hide most of the intricate implementation and computation mapping details from the developers. In our case, we need to simultaneously support the expression of both significance and parallelism, already putting a lot of burden to the programmer. Therefore, we should make sure that migration of parallel, or even sequential codes to our model does not pose unreasonable overhead to the programmer. Moreover, it would be desirable to allow incremental porting, without widespread and intrusive changes to the original source code.

2.3 High Level Description

In the next paragraphs we will utilize an abstract figure to demonstrate the basic concepts of the model. Figure 2.1 depicts an initial, single threaded application, which however has areas (dark rectangles) that can be approximated or even dropped.

2.3.1 Task-Based Programming model

The programming model adopts a task-based paradigm, similar to OpenMP [12]. Task-based models are quite popular in both the academia and the industry. Such
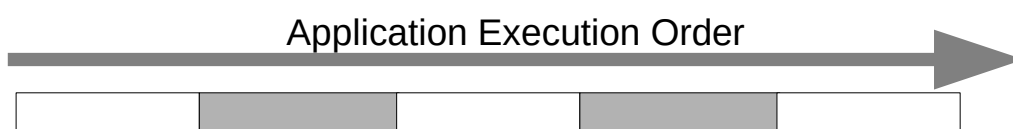
FIGURE 2.1: A single threaded execution of an abstract application, dark rectangles correspond to parts of the application that can be parallelized

models offer a straightforward way to express communication (data transfers) across tasks. Parallelism is expressed by the programmer in the form of independent tasks, however the scheduling of the tasks is not explicitly controlled by the programmer. The developer simply defines dependencies among tasks and the underlying runtime system schedules tasks on available hardware as soon as all their dependencies have been met.

Adopting a task-based model is an appealing decision:

- It can express parallelism in an efficient manner (objective 2.2.4).

- It minimizes the burden related to controlling the parallelism. (objective 2.2.4

- It provides isolation between different jobs (tasks). A task is independent from others during its execution, with well defined input and output data flows (objective 2.2.2).

- Each task is a computation with well defined entry and exit points. This facilitates error detection / recovery tests at task boundaries (objective 2.2.2).

Pragmas for the Expression of Parallelism and Significance

The proposed programming model supports task creation and significance characterization by annotating the input source code with *#pragma* compiler directives. Pragma-based programming models have the advantage of facilitating non-invasive and progressive code transformations, without requiring a complete code rewrite. Adopting compiler directives to support parallelism in combination with the task based model improves the user friendliness of our proposed model to the user (objective 2.2.6). Pragmas identify tasks and characterize them as significant or not, thus steering their execution on reliable and non-reliable cores, respectively. Each task pragma construct specifies a function which is equivalent to the task body, along with its data-flow and accompanying significance information. Thus the main granularity of significance characterization is that of a task.

Significance takes values in the range [0.0, 1.0] and characterizes the relative importance of tasks for the quality of the end-result of the application. Whether the task will be execute reliably or unreliably will be decided by the runtime system during execution time. This allows developers to flexibly assign significance values to tasks depending on information available only during execution time, for example the value of a variable.

In Figure 2.2 we show the decomposition of the abstracted example presented in figure 2.1 into computation chunks. These chunks are categorized with various significance values (shades of gray), using the programmer intuition, knowledge, and/or domain-expertise. Tasks can be executed in parallel. Note that after the parallel execution of tasks, implicit synchronization may be necessary.
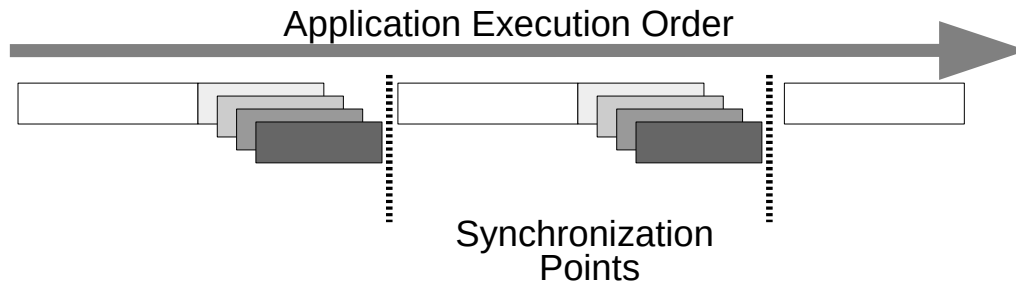
FIGURE 2.2: Application tasks are created and tagged with significance information
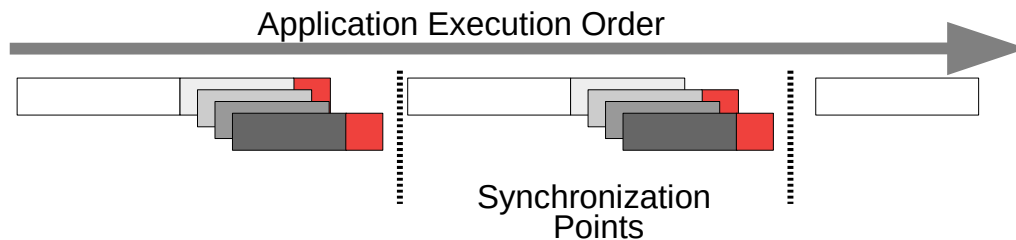
FIGURE 2.3: Non-reliable tasks may execute error identification and correction functions after their termination

Early Error Detection To minimize Fault Propagation

Non-reliable tasks are expected to have unpredictable behavior due to the uncertainty of their execution and can result into faults in the hardware level. These faults can lead to arbitrary error propagation up to the final output of the program, or total program crashes if they are not identified and isolated early on. The programming model provides mechanisms that aim at identifying errors and — if possible — even correcting them (objective 2.2.2). In the event of detected errors the developer is given the opportunity to specify the recovery strategy (such as ignoring the error , re-executing, or even assigning a default value at the task output).

The programming model allows the programmer to specify result-check functions, which will be executed right after a non-significant task completes. In this function, which is guaranteed (by contract) to be executed reliably, the user can check for possible errors or failures and even provide a default value to the calling program as an acceptable replacement of the result of a failed execution.

We expect that the computational overhead of a result-check function should be minimal, preferably orders of magnitude lower than that of the corresponding task. Otherwise, the overhead introduced by its execution would cancel the performance/power consumption benefits of unreliable execution.

The running example is now modified to include the aforementioned task-level result-check functions which are illustrated as red rectangles in Figure 2.3

Although result-check functions offer a degree of error detection and isolation, in some occasions checking a single task's output is not sufficient to decide if the computation is correct. A typical example is image and video processing applications, in which the acceptance of the result for a pixel can be judged only in relation to the values of neighboring pixels. To support error detection/correction at a coarser

granularity, tasks can be grouped, and result-check functions can also be executed at the end of each group. We extend the running example to allow for group-level result check functions as shown in Figure 2.4

Elastic Synchronization

In the context of the programming model we support a more elastic explicit synchronization model for barriers (objective 2.2.5). Barriers can be equipped with watchdog timers which terminate the wait after the specified time frame has elapsed. In the case when the synchronization was not successful — in a traditional perspective — the run-time system terminates all tasks that did not reach the barrier (if they have not already crashed) and resumes execution of the following code.

Whenever a non-reliable task fails is stack in an infinite loop the entire application will wait. The use of an elastic barrier in this case safeguards the application from the otherwise unavoidable deadlock: it ensures that the task group will terminate, either successfully, or with an error code, thus allowing the application to continue or try to recover.

The running example is now finalized as shown in Figure 2.5 and serves as an example of relaxed synchronization. Note that, due to a timing constraint which was not met, the last non-significant task was terminated by the run-time.

Significance of Data

The main point of interest in a significance-aware computing platform concerns the quality of the output results. After all, the role of computations – significant or not –
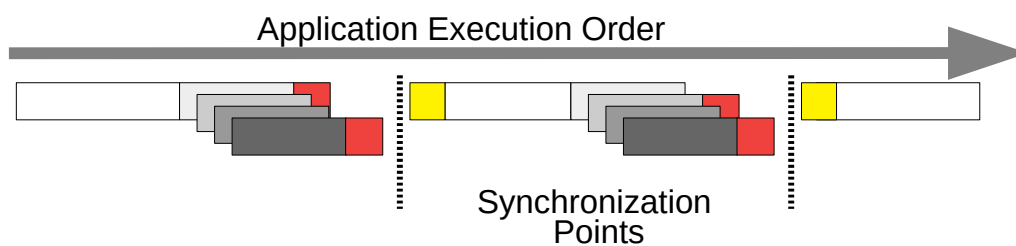
FIGURE 2.4: Result-checks at the group-of-tasks granularity
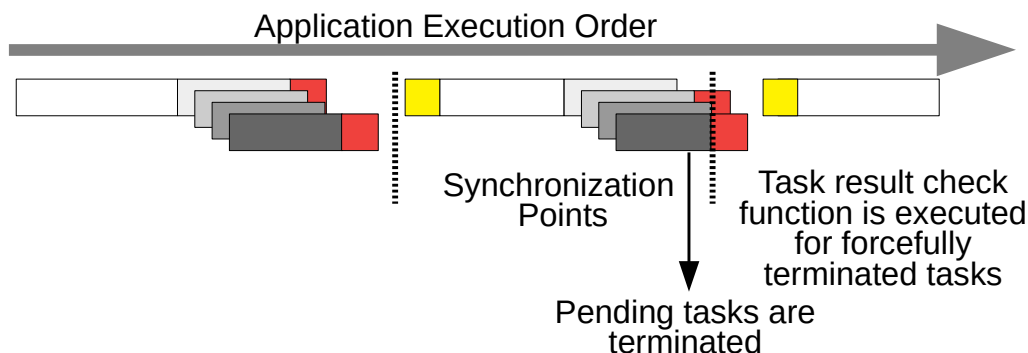
FIGURE 2.5: A case of relaxed synchronization which results to termination of late tasks

is to produce output data that are acceptable to the end user.

In this PhD dissertation we have selected to only characterize explicitly the significance of computational tasks, and not that of data. In other words, there is no explicit tagging of significance of data structures in the code. The rationale is that early error detection and correction using result-checking functions (as explained earlier in this section) precludes the flow of unacceptably erroneous data between tasks. If data are deemed unacceptable by a result-checking function, one option may be to substitute the erroneous value by a default value so that the program can continue. This approach greatly reduces the amount of effort that the programmer has to expend to express significance in the code.

2.4 Syntax

In this section we define the API of the proposed programming model which realizes concepts and ideas expressed in section 2.3.

2.4.1 Task Definition and Significance Characterization

Tasks are defined using the $task$ directive (Listing 2.1). The $significance()$ clause specifies the significance of the task, and takes values in the range $[0.0, 1.0]$, indicating the importance of the task with respect to the output quality/correctness of the result. Depending on their significance, tasks may be executed on top of reliable or unreliable hardware. The significance expression is evaluated at execution time, thus allowing the programmer to parameterize task significance depending on information accessible only at exection time (e.g values of variables).

The $taskcheck()$ clause specifies a result-check function, which is invoked only if the task is executed unreliably. The result-check function is always executed reliably, and can be used by the developer to:

- inspect the task status to see if it completed its execution normally or has crashed;

- assess whether the task output is wrong;

- assign meaningful default values to the task output;

- request a re-execution of the task.

The result-check function has implicitly access to all arguments of the corresponding task, and may return either $TRC\_SUCCESS$ or TRC_REDO to the runtime. In the latter case, the task is re-executed reliably.

```
1  #pragma omp task  [significance(...)]
2                    [label(...)][in(...)] [out(...)]
3                    [taskcheck(resultcheck())]
```

LISTING 2.1: #pragma omp task

```
1  #pragma omp taskwait [label(...)]
2                       [ratio(...)]
3                       [time(...)]
4                       [groupcheck(resultcheck())]
```

LISTING 2.2: #pragma omp taskwait

The programmer can define the inputs and outputs of the task, via the $in()$ and $out()$ clauses, respectively. This information can be used by the runtime to infer task dependencies and schedule tasks accordingly. Finally, the *label()* clause can be used to group tasks, and to assign the group a common identifier (name), which is in turn used as a reference to implement synchronization at the granularity of task groups (Section 2.4.2).

2.4.2 Synchronization

The programming model supports explicit barrier-type synchronization through the *#pragma omp taskwait* directive (Listing 2.2). A *taskwait* can serve as a global barrier, instructing the runtime to wait for all tasks spawned up to that point in the code. Alternatively, it can implement a barrier at the granularity of a specific task group, if the *label()* clause is present; in this case the runtime system waits for the termination of all tasks of that group.

Furthermore, the *omp taskwait* barrier can be used to control the quality degradation of application results. Through the *ratio()* clause, the programmer can instruct the runtime to treat (at least) the specified percentage of all tasks – either globally or in a specific group, depending on the existence of the *label()* clause – as significant and the remaining tasks as non-significant. The runtime must also respect the relative ordering of tasks with respect to their significance. In other words, a task with higher significance (value of the respective *significant* clause should not be treated as non-significant while a task with lower significance is considered by the runtime to be significant. The $ratio$ takes values in the range [0.0, 1.0] and serves as a single, straightforward knob to enforce a minimum quality in the performance / quality / energy optimization space. Smaller ratios give the runtime more opportunities for optimization, however at a potential quality penalty.

```
1  int dct_taskrescheck(...) {      /* DCT task result-check function.      */
2    if ( task_crashed() )          /* Takes the same arguments as the task, */
3      coeff = 0;                   /*  returns int.                         */
4    else if ( abnormal(coeff) )
5      coeff = 0;
6    return TRC_SUCCESS;
7  }
8  void dct_task(...) {    /* Calculate the coefficients for a specific 2x4 block */
9    ...                   /* over a number of different 8x8 blocks.              */
10 }
11 void DCT(...,double taskratio){/*DCT calculation. The taskratio is an extra
        parameter.*/
12   float sgnf[] = {1.0,  0.9,  0.7,  0.3,    /* Significance look up table   */
13                   0.8,  0.4,  0.3,  0.1};   /*for each of the 2x4 sub-blocks */
14   for each 2x4 sub-block K {/* Iterate over the blocks of the DCT coefficient
        space. */
15     #pragma omp task  significance(sgnf[K]) taskcheck(dct_taskrescheck())
16         dct_task(...);/* Task to calculate the Kth 2x4 sub-block, over all 8x8
        blocks */
17   }
18   #pragma taskwait ratio(taskratio) time(16)
19 }
```

LISTING 2.3: programming model use case: dct pseudo-code

2.5 Example

Listing 2.3 presents a task based implementation of *DCT* using our programming model. Line 15 defines a task to compute the frequency coefficients of a specific 2x4 sub-block. All tasks created in this loop have varying significance depending on their position in the 8x8 block: upper left sub-blocks have higher significance than lower right, as encoded in the $sgnf$ array. In line 15, $dct\_taskrescheck()$ is specified as the result-check function. This function checks whether the task crashed (Line 2) or whether its output is wrong (Line 4). In both cases a the corrections sets the respective coefficients to 0. Since this correction does not require task re-execution the function returns $TRC\_SUCCESS$ (Line 6).

In Line 18 of Listing 2.3, the barrier for all *dct* tasks is specified with a timeout of 16 msec; this corresponds to a target frame rate of 30 fps, assuming *DCT* corresponds to almost 50% of the computation time for each frame. Note that the $taskratio$ is an open parameter that is supplied when the program is invoked. In effect, it serves as a knob, to set the "borderline" between the most-significant sub-blocks that have to be computed reliably, and the less-significant sub-blocks that may be computed unreliably. No group-level result-check function is used in the example, because task-level result checks and repairs are sufficient.

2.6 Programmer Insight

The programming model assumes that the developer is sufficiently familiar with the application to take good decisions as to how to structure the computation in tasks, which tasks to characterize as more significant, and which result-check functions to provide. Similar to parallelism, significance is a key algorithmic aspect that requires

the programmer's full attention, but unlike parallelism task significance is orthogonal to the underlying platform architecture.

Choosing result-check functions is also important. If the result-check function is too complex, it is practically useless, as the same result could be achieved simply by declaring the task as significant, and executing it reliably in the first place. If too simple, the result-check function may erroneously mis-characterize and destroy good task output, possibly deteriorating the end result of the computation.

Finally, task granularity is an important parameter that should be considered when using this programming model. Fine-grained tasks may allow for a richer (more diverse) significance characterization, which in turn can be exploited to achieve a smoother degradation of output quality at increased energy gains. The downside is that having many small tasks will also increase the task management overhead of the runtime system, both in terms of time and energy consumption.

2.7 Significance-aware Runtime System

The runtime system is designed for a multicore shared memory platform, in which cores can be set to operate in various voltage-frequency configurations (V, f), even in ones below nominal values. Unsafe settings only apply to the cores of the CPU, including the integer and FPU pipeline logic as well as the L1 and L2 caches. Modules critical to the correct operation of all cores, such as buses, memory controllers and cache coherence mechanisms are set to a safe setting and thus always operate reliably. Our power model takes this into account and all reported energy gains are gained from undervolting the core part. A user-level library implements the runtime system and runs on top of the Linux operating system. A source-to-source compiler, which we developed based on [148], lowers programs that use the primitives of our programming model to code with calls to the runtime system API. Finally, the produced source code is compiled into machine code using the standard *gcc* tool chain.

2.7.1 Runtime Execution Management

We consider three different hardware configurations:

FastRel: This onfiguration is a high-performance nominal point of operation, with high voltage/frequency (V_h, f_h), where a core executes code fast.

SlowRel: This configuration is a slower nominal operation point in terms of supply voltage and frequency (V_l, f_l). Consequently, parts of this application have low power consumption but at the same time increased execution time.

FastUnRel In this configuration cores are set to operate in the *unsafe* region, the operating point, called $FastUnRel(V_l, f_h)$, operates with the same (low) voltage as $SlowRel$ and the same (high) frequency as $FastRel$. Code execution in $FastUnRel$ is equally fast as in $FastRel$ yet more energy-efficient. At the same
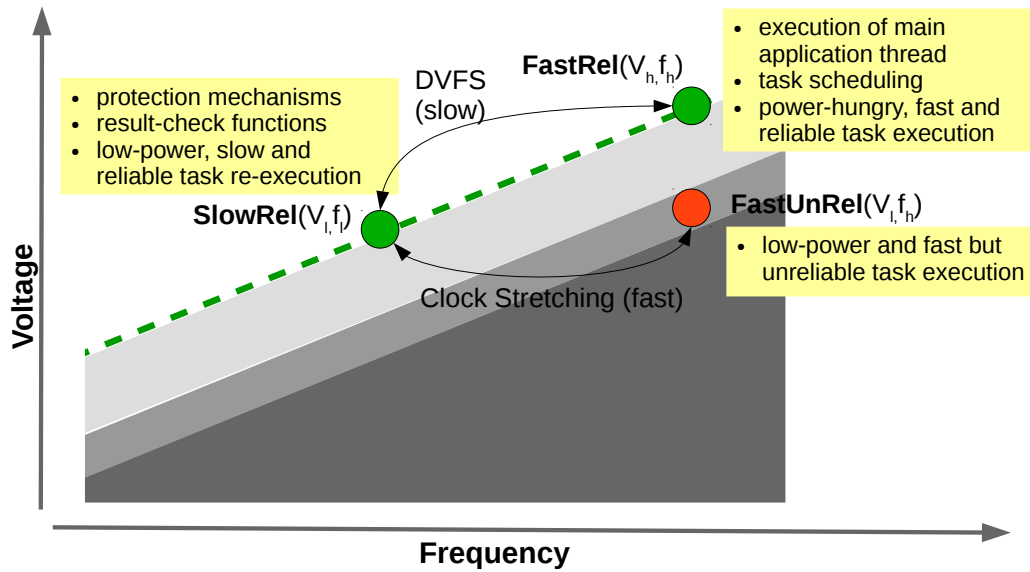
FIGURE 2.6: The configurations $FastRel$, $SlowRel$ and $FastUnRel$ used by the runtime system, to reduce the energy footprint by exploiting the significance of computations. Our approach exploits non-nominal configurations within the *unsafe* region, that are energy-efficient but unreliable.

time, execution is potentially unreliable due to timing faults, since $FastUnRel$ is outside the nominal range of operation.

We assume that the runtime system can switch the operation of cores dynamically. Due to the difference in their voltage, the transition between $FastRel$ and $SlowRel$ requires a *voltage and frequency scaling* step, which introduces significant delay. In contrast, given that $SlowRel$ and $FastUnRel$ have the same voltage, the transition between them can be done quickly via clock stretching [19]. Figure 2.6 illustrates the principle of operation.

The main application thread and the master runtime thread are executed reliably in the $FastRel$ configuration. The tasks of the application can be executed reliably in the $FastRel$ configuration, or unreliably in the $FastUnRel$ configuration, depending on their relative significance and the user-supplied task ratio (see Section 2.4.2). Task execution is done using separate worker threads, with each worker being placed in a different core. To reduce the number of voltage transitions, task scheduling is done in two alternating phases. In the first phase, workers are configured to operate in $FastRel$, and the master thread schedules all the tasks in the ready list that have been flagged for reliable execution. Before the second phase starts, all workers soft-checkpoint crucial context information to use it to recover in case of corruption from faults [1]. Afterwards, the main thread requests the memory allocator (discussed in Section 2.7.2) to protect all the memory pages as well as the stack of the main application thread. This actually forces all data, including non-significant output data, to a read only state. Reliable task input/output data can be mixed with unreliable

---

[1]We use the Linux *getcontext()* function. The state is copied to a read-only memory page to prevent it from being written accidentally.

input/output data in the same memory page. However the Operating System (OS) assigns privileges at the granularity of a page, therefore when locking a page to read only state even unreliable tasks cannot write to their output data locations. To overcome this, each worker allocates extra memory in which the non-significant tasks will store their results. These memory locations have read write permissions.

At this point the second phase starts. Workers switch from $FastRel$ to $SlowRel$, and the master thread proceeds with the scheduling of all the tasks that have been flagged for unreliable execution. When a worker is assigned with a task, it switches to the $FastUnRel$ configuration and executes the task. If during task execution an event causes the OS to take over (e.g. an I/O event), the worker switches to $SlowRel$ prior to executing the kernel code, and switches back to $FastUnRel$ mode when it resumes the execution of the application task. When the task completes or crashes, the core is switched back to $SlowRel$, the previously saved state is restored, and the result-check function of the task is invoked.

If the result-check function requests task re-execution, the worker repeats the execution but maintains the core in the reliable $SlowRel$ configuration. When all tasks have finished their execution or the synchronization timing constraint is reached, the main thread requests from the allocator to revert the protected memory privileges to their previous state. Afterwards the main thread copies the computed output data from unreliable tasks back to their original memory locations. In case the group result-check function requests re-execution of the task group, the master thread configures all workers to operate in $FastRel$. Then, all tasks in the group that have been flagged for unreliable execution are re-scheduled from scratch, and are executed reliably. The overhead of switching to a different voltage level is amortized by the execution of a large number of tasks.

The runtime supports the following levels of protection:

No Protection (NP): The runtime system does not employ any error detection/-correction mechanism or programmer-supplied significance information. All tasks of the application are executed unreliably ($FastUnRel$ configuration) and are susceptible to faults. A task crash leads to the abrupt termination of the entire application.

Basic Protection (BP): All applications tasks are executed unreliably as in NP, but the runtime system identifies and handles errors using the standard processor/OS protection mechanisms, including the internal soft-checkpointing of critical state and the memory protection mechanism. As a result, task crashes are properly caught. However, the programmer-supplied result-check functions are ignored.

Basic & Result Checking (B-RC): In addition to BP, when an application task completes its execution normally or crashes, the runtime system invokes the programmer-supplied result-check function to detect and correct possible errors.

Basic & Significance (B-SF): On top of *BP*, the runtime system takes into account the programmer-supplied significance of tasks and ratio, and schedules them for execution accordingly. As a consequence, the most significant tasks are executed reliably (in the *FastRel* configuration), while the less-significant tasks are executed unreliably (in the *FastUnRel* configuration). Task crashes are caught and handled as in *BP*, and the programmer-supplied result-check functions are ignored.

Full System (FS): The entire protection arsenal is employed, including basic runtime system protection, task scheduling based on the programmer-supplied significance information, and invocation of the result-check functions for unreliable tasks.

Full System & Re-Execution (FS-RE): Like *FS*, but if the task result-check functions detect a task crash or invalid output, they request a full task re-execution, rather than trying to repair the task output.

2.7.2 Memory Management

Our framework has been developed on a shared memory system, which is the worst case scenario in terms of reliability. Erroneous stores by code that executes on an unreliable core may affect global data structures or the memory of significant computations.

The runtime system utilizes a custom dynamic memory manager which requests memory slabs from the OS at the granularity of pages and serves dynamic allocations from either the application or the runtime system. When switching to an unreliable execution phase, the memory manager assigns read-only privileges to all used heap pages, to protect them from rogue stores from tasks executed on non-reliable cores. Should such a store be attempted, it leads to an exception which is handled accordingly by the runtime system. Besides the heap, faults can also corrupt the stack. The runtime system allocates its internal data structures dynamically, thus there is no information within the stack or the global data space to be protected from application errors. All memory pages used as stack by the main application thread are also set to be read-only prior to executing unreliable code. The current framework does not implement any global data protection as this requires compiler support.

2.7.3 Life of a group-of-tasks

Figure 2.7 illustrates the typical life of a group-of-tasks in an application implemented using our significance aware *fault tolerant* computing programming model. For each group instantiated during the life of an application the runtime system receives a collection of tasks with varying significance values (represented with different colors in Figure 2.7), a desired approximation level in the form of a taskwait *ratio*.
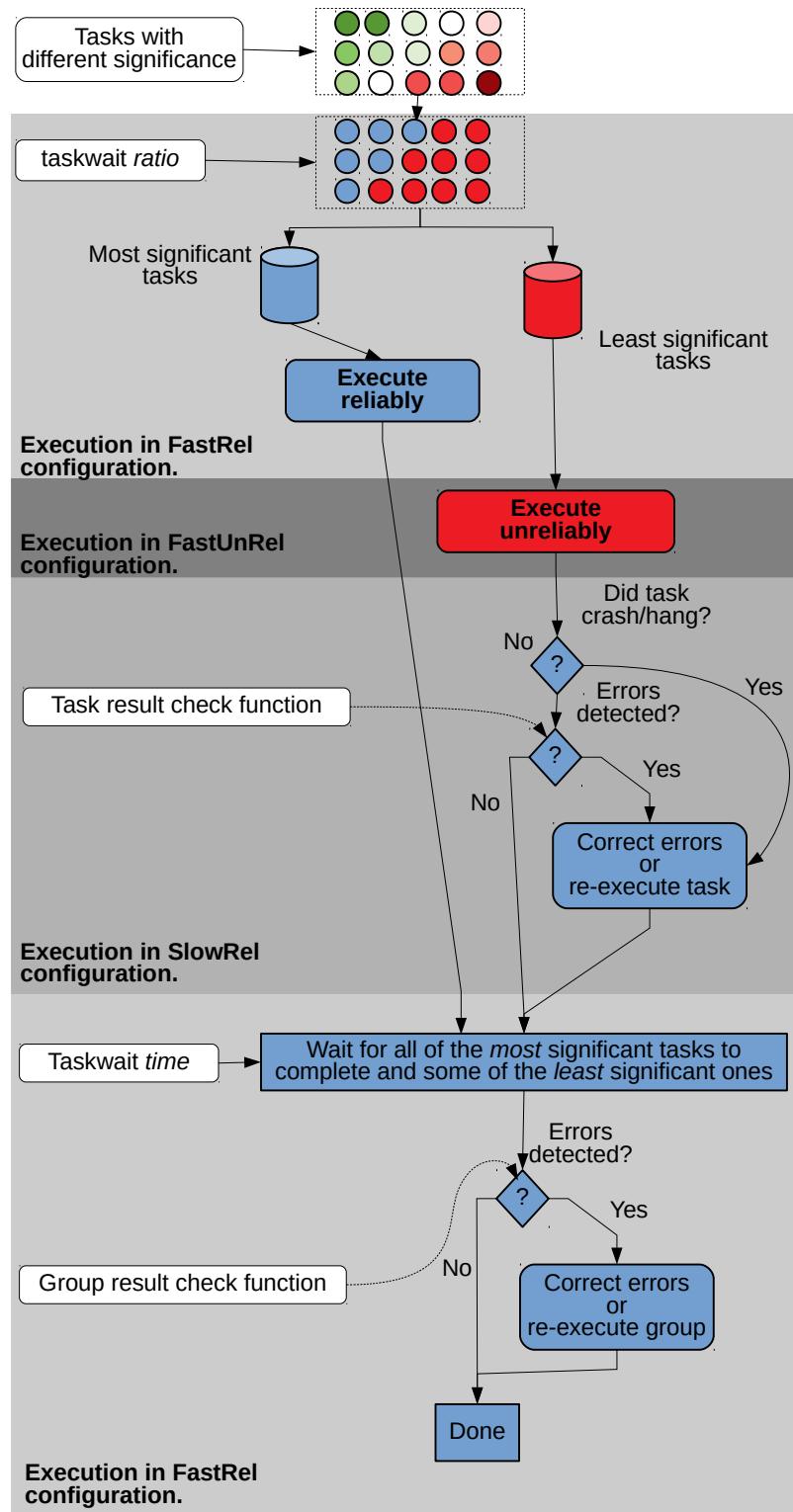
FIGURE 2.7: The typical life of a group-of-tasks in the context of significance aware unreliable computing

Based on the $ratio$ value, the runtime system partitions the tasks into two sets, the most significant tasks and the least significant ones. The most significant ones are executed under reliable conditions ($FastRel$ configuration), whereas the runtime system schedules the least significant ones on hardware that is operating unreliably

($FastUnRel$ configuration). Tasks which are executed unreliably are monitored for abnormal behaviour (crashes, and infinite loops). In the event that they complete their execution without an obvious error, their outputs undergo an error detection phase. Tasks whose outputs are considered to be invalid may then be re-executed on undergo an error correction phase. Result checking may also be performed at the granularity of group-of-tasks, this way we enable developers to assess the validity of the aggregated result of a group-of-tasks.

2.8 Evaluation

In this chapter we evaluate the programmability of the proposed programming model and the extra overhead induced by the various protection mechanisms provided by the runtime system. Section 2.12 evaluates the energy gains of the proposed programmin model.

2.8.1 Benchmarks

We use five benchmarks, listed in Table 2.1, and apply three different methodologies to perform significance characterization on them. In *DCT* we use **domain expertise** to identify the significance of different parts of the computation. The tasks that compute low frequency coefficients are close to the upper left corner of each 8x8 frequency block, and are more significant than the ones computing coefficients towards the lower right corner of the block.

In *Blackscholes* and the iterative benchmarks *K-means, Jacobi* we employed a **profile-driven** approach. More specifically, in *Blackscholes* we injected bitflips in the input data and observed the output quality. All parts of the code appear to be equally significant, since faults had similar manifestations regardless or task computations. Therefore, all tasks are assigned equal values of significance since all stock options are considered equally important.

In *Jacobi* and *K-means* we injected bit-flips in the input data of a randomly chosen iteration, and compared the relative error of the faulty execution with an error-free one. In both *Jacobi and K-means* we observe that errors in the last few iterations tend to severely reduce the output quality, and thus infer that these are the most significant ones. Finally, in *Sobel* we exploit the perceptual properties of the human eye, and **randomly** distribute the significance among tasks. This way errors are spread across the entire output image and the loss of quality is not clustered in a specific area of the image.

2.8.2 Evaluation of Programming model and Runtime System

In all benchmarks we used a very simple result-check function. The result-check function of *DCT* detects errors in the task output via a heuristic out-of-bounds check;

| | | | Lines of Code | |
Benchmark	Domain	Sgnf. Characterization	Task	TRC function
DCT	Multimedia	Domain expertise	39	34
Sobel	Image Filter	Randomly	54	42
Blackscholes	Finance	Profile-driven	117	105
K-means	Data mining	Profile-driven	141	57
Jacobi	Numerical Solver	Profile-driven	62	39

TABLE 2.1: Lines of code (LOC) for the tasks and corresponding result-check and correction functions for each benchmark. The result-check functions are implemented based on the original task code, which was modified to reduce its computational complexity.

coefficients that do not respect the bounds are set to zero. In *Sobel* the task result-check function corrects only tasks that crashed during their execution by running an approximate version of the Sobel filter, using a lightweight stencil with just 2/3 of the filter taps. *Blackscholes* is a benchmark of the Parsec suite [8]. Results are checked with the *isfinite()* macro. This is a *glibc* floating point classification macro, it returns a non-zero value if the value under inspection is not *NaN*, or *infinite*. If the check fails, the function uses a faster implementation of the *Blackscholes* formula, by substituting costly mathematical operations (such as $expr(), sqrt(), log()$) with approximate versions. In *K-means* the result-check function of non-significant tasks is minimalistic, exploiting the error-tolerant nature of this iterative application: if a point attempts to subscribe itself to cluster but miscalculates the cluster's id then it reverts to its previous cluster. Also, if the runtime system reports an error, then all points computed by the task are subscribed back to their previous clusters. In *Jacobi* it is hard to create an error detection mechanism, since assessment of the quality of results is associated with the application in which the solver is used. We implement a simple result-check function which uses the *glibc isfinite()* macro to detect obvious errors to the output of tasks. In the event of detecting such an error, the current solution estimate is replaced with that of the previous iteration.

In our benchmarks, the result-check part was simple, mostly based on range checks. For the correction part, we reused the original task code and, in some cases, modified it to perform the computation approximately. Table 2.1 shows that result-check functions are almost as big as the tasks themselves. Nevertheless, since we heavily reused the existing task code, the actual effort to implement the result-check function was minimal.

2.8.3 Runtime Overhead

Figure 2.8 breaks down the execution time of a task for each benchmark using a fixed frequency of 3.7 Ghz. The time spent by the runtime system to create and schedule tasks, to protect the memory and to checkpoint the state of each task before execution is less than 5% of the total task execution time. Task creation and scheduling overhead is practically constant, at about 5000 cycles. The same applies
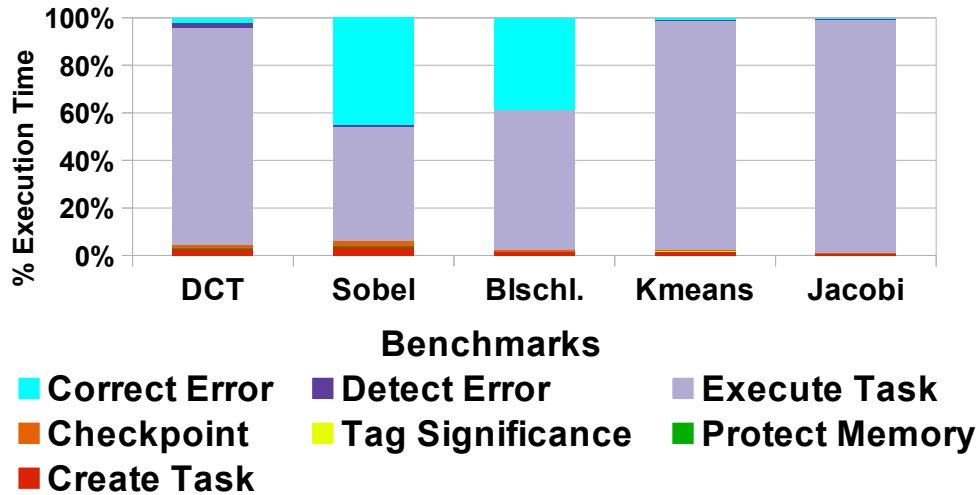
FIGURE 2.8: Breakdown of task execution time, for each benchmark.

to checkpointing, which costs approximately 2000 cycles. Noticeably, in *Sobel* and *Blackscholes* the overhead of correction is comparable with the task execution time. These two benchmarks execute an approximate version of the computation as a correction heuristic, whereas the rest simply discard the computed erroneous solutions, which incurs almost zero overhead.

2.9 Energy Reduction Evaluation methodology

Commercially available platforms do not allow individual cores to be operated below nominal settings, hence cannot be used to support the runtime model that was described in Section 2.7. We note that operation in sub-nominal (V, f) values is possible in some but not all convenitonal CPUs, but only for the entire CPU. This is not useful for the purposes of our work, because at least one core has to always work reliably in order to run the OS and the runtime system. As a consequence, we cannot take real measurements on the performance, energy consumption and fault rate/behavior of the system.

To evaluate our framework we use a suitable model for estimating the execution time and energy consumption of a computation as a function of the voltage-frequency settings of the $FastRel$, $SlowRel$ and $FastUnRel$ configurations, and the tasks that are executed in these configurations. The model also takes into account the task management overheads of the runtime system, as well as the cost for performing the voltage and frequency scaling steps needed for a switch between $FastRel$ and $SlowRel/FastUnRel$. This allows us to run computations on a real platform, trace its execution, profile the performance and power parameters used as input to the model for the $FastRel$ and $SlowRel$ configurations and extrapolate estimates for the $FastUnRel$ configuration. The performance model is described in detail in Section 2.10.
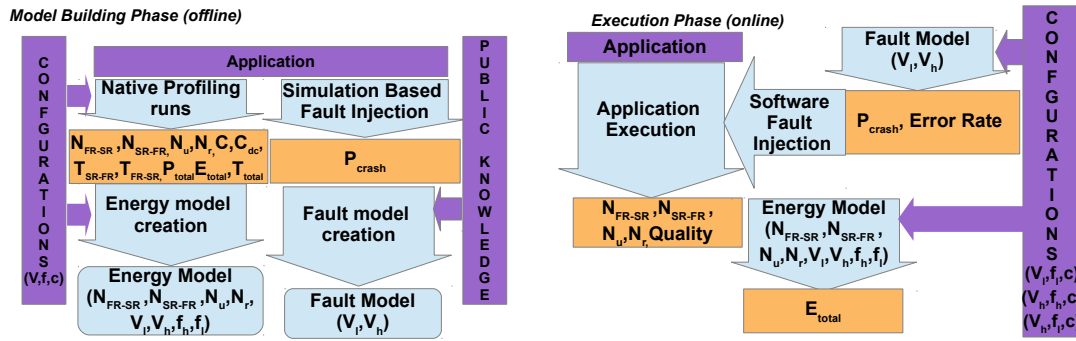
FIGURE 2.9: Evaluation approach: we build the performance, energy and fault models (left), and use these models to drive experiments and estimate energy consumption (right).

In any case, this performance modeling is insufficient. When executing in the $FastUnRel$ configuration the hardware may experience timing errors, which in turn trigger the respective runtime protection mechanisms. A major challenge is to associate the operation in the $FastUnRel$ configuration with the probability of hardware faults due to timing violations. Another issue is how to assess the impact of such faults to the actual execution and outputs of a given task, which is entirely application-specific. We use a model that estimates the fault rate as a function of the voltage-frequency setting of $FastUnRel$. We use a combination of simulation-based and software-based fault injection to observe the impact of faults on the benchmarks.

Figure 2.9 illustrates the workflow of our evaluation approach which is split into two phases. During the model building (offline) phase we run the benchmarks in the GemFI simulator (see section 2.11.2) to obtain the probability of crash (P_{crash}). We also perform native executions and measure the number of reliable and unreliable tasks (N_r, N_u), the number of transitions from $FastRel$ to $SlowRel$ and vice versa (N_{FR-SR}, N_{SR-FR}), the average time required to perform a voltage and frequency transition from $FastRel$ to $SlowRel$ and vice versa (T_{FR-SR}, T_{SR-FR}), the length (in cycles) of a task (C), the length of its result check function (C_{dc}), the total energy and power consumption (E_{total}, P_{total}) of the program as well as its total execution time (T_{total}). We create the energy model using the above information as well as the operating frequency (f), the supply voltage (V) and the number of cores (c). The energy model input parameters are the number of reliable/unreliable tasks (N_r, N_u), the number of active cores (c), the number of voltage and frequency transitions (N_{FR-SR}, N_{SR-FR}) , the supply voltage and the operating frequency of all configurations. Its goal is to estimate the energy cost of an unreliable execution. We model application error resiliency via simulation-based fault injection combined with observations in literature. The fault model input arguments are the supply voltage of the $FastRel$ and the $FastUnRel$ configuration. The energy and fault models are discussed in more detail in the following sections. During the execution phase, the models are used to inject errors and to estimate the energy consumption of the execution. After each execution, the system reports the values for the number of

tasks that were executed reliably (N_r) and unreliably (N_u), the number of transitions between the $FastRel$ and $SlowRel/FastUnRel$ (N_{FR-SR} and N_{SR-FR}) configurations. Using this data, we estimate the energy footprint of the computation.

2.10 Execution Time and Energy Consumption Model

We introduce an analytical model for the performance and energy consumption of a program as a function of the core frequency, the voltage, the number of tasks that are executed reliably and unreliably and the number of voltage and frequency transitions. Our model is agnostic to the CPU structure and captures the execution phases of an application. Therefore it accounts for both the *core* and *uncore* components of the CPU. The model is validated for our CPU platform, where it predicts the actual energy consumption of our benchmark applications with high accuracy over a wide range of different (nominal and thus reliable) core configurations.

2.10.1 Execution time modeling

As discussed, the runtime uses three different voltage/frequency configurations, $FastRel = (V_h, f_h)$, $SlowRel = (V_l, f_l)$ and $FastUnRel = (V_l, f_h)$. Equation 2.1 expresses the time for executing a given piece of code N times, where C denotes the number of cycles spent for code execution, and f is the frequency of the core depending on its configuration setting (f_h for $FastRel/FastUnRel$ and f_l for $SlowRel$).

$$T(N, f, C) = \frac{C}{f} \times N \tag{2.1}$$

Tasks can be executed in parallel by the workers of the runtime system, on different cores. Besides task execution itself, the system software spends additional time to schedule tasks and to manage unreliable task execution. Equations 2.2 give the total execution time of an application:

$$T_{FastRel} = T(N_r, C, f_h), T_{SlowRel} = T(N_u, C_{dc}, f_l), T_{FastUnRel} = T(N_u, C, f_h)$$
$$T_{vfs} = N_{FR \to SR} \times T_{FR \to SR} + N_{SR \to FR} \times T_{SR \to FR} \tag{2.2}$$
$$T_{Total} = \max_{i=1}^{Workers} \left(T_{FastRel_i} \right) + \max_{i=1}^{Workers} \left(T_{SlowRel_i} + T_{FastUnRel_i} \right) + T_{vfs}$$

The execution time for each worker in each configuration is expressed by $T_{FastRel}$, $T_{SlowRel}$ and $T_{FastUnRel}$. Variable C is the average number of cycles required to execute a task in $FastRel/FastUnRel$, while C_{dc} is the average number of cycles required by the runtime system to prepare for an unreliable task execution and to execute the result-check/repair function in the $SlowRel$ configuration. Variables N_r and N_u express the number of reliable and unreliable tasks executed by the worker, respectively. T_{vfs} captures the time required to switch between the $FastRel$ and

SlowRel configurations. Variable $N_{FR \to SR}$ denotes the number of times the runtime system switches from the *FastRel* to *SlowRel*, and $T_{FR \to SR}$ is the average time required to perform this transition. Similar parameters apply for the reverse direction. Finally, the total execution time of the application is the maximum execution time among all workers for the first task scheduling phase (in the *FastRel* configuration), plus the maximum execution time among all workers for the second task scheduling phase (in the *SlowRel/FastUnRel* configurations), plus the time spent on the respective voltage and frequency transitions.

2.10.2 Power and energy modeling

The total power dissipation of a CMOS circuit is given by Equations 2.3. P_{dyn} is the dynamic power dissipation, P_{leak} is the power dissipation due to transistor leakage current, and P_{shortC} is the power dissipation due to the short circuit formed when both the PMOS and NMOS transistor tree momentarily conduct current during CMOS switching. Since modern fabrication technologies which use high-k dielectric materials can control leakage current, it is the P_{dyn} component that dominates power dissipation. Therefore, our model considers the idle power consumption of a processor as a constant P_{idle} and equal to the sum of P_{leak} and P_{shortC}. The uncore power consumption of the CPU is included in P_{idle}. Since the P_{idle} is a constant all the energy gains are a result of the undervolted core part of the *CPU*. P_{dyn} is the product of the supplied voltage squared (V^1), the frequency (f) and the activity factor $A(\vec{c})$. We have observed that the activation of a new core in our multicore platform results to power steps. The number of cores used by the application are captured via vector $\vec{c}$, where $\vec{c}[n]$ is 1 if n cores are active , else 0. $A(\vec{c})$ is the dot product of $\vec{c}$ and a vector $\vec{w}$ containing per-core switching capacitance values which are obtained via regression.

$$P_{Total} = P_{idle} + P_{dyn}, \ P_{idle} = P_{leak} + P_{shortC}$$
$$P_{dyn}(\vec{c}, V, f) = A(\vec{c}) \times V^2 \times f, \ A(\vec{c}) = \vec{c} \cdot \vec{w} \tag{2.3}$$

The total energy dissipation E_{Total} is given by Equations 2.4. In general, this depends on the hardware/core configuration and the time spent to execute the runtime management functions, the application tasks, and their result-check/repair functions, as discussed above.
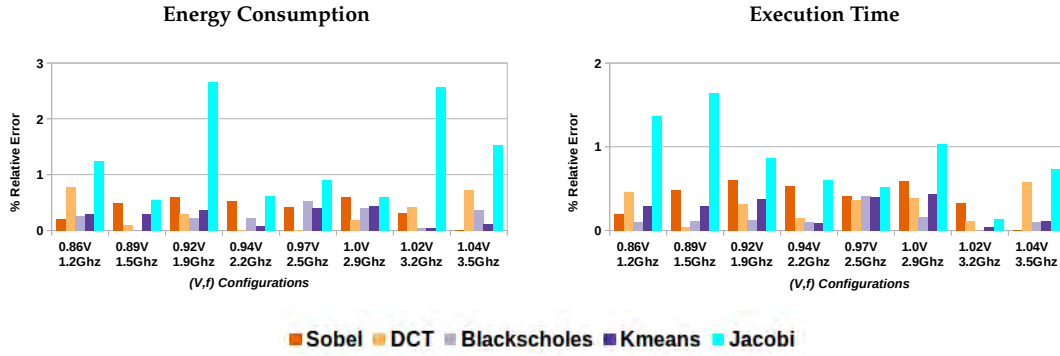
FIGURE 2.10: Relative error for the execution time and energy as predicted by our model vs. a real execution, for our application benchmarks when half of the tasks execute in the $FastRel = (3.7Ghz, 1.06V)$ configuration and the other half in a lower-power $SlowRel$ configuration. All $SlowRel$ configurations are shown in x-axis.

$$E_{FastRel} = P(\vec{c}, V_h, f_h) \times \max_{i=1}^{Workers} (T_{FastRel}), E_{FastUnRel} = P(\vec{c}, V_l, f_h) \times \max_{i=1}^{Workers} (T_{FastUnRel})$$

$$E_{SlowRel} = P(\vec{c}, V_l, f_l) \times \max_{i=1}^{Workers} (T_{SlowRel})$$

$$E_{vfs} = N_{FR \to SR} \times T_{FR \to SR} \times P(V_h, f_h) + N_{SR \to FR} \times T_{SR \to FR} \times P(V_l, f_l)$$

$$E_{Total} = E_{FastRel} + E_{SlowRel} + E_{FastUnRel} + E_{vfs}$$

$$(2.4)$$

2.10.3 Calibration and validation

We calibrate and validate the timing and energy models based on measurements taken on our platform, for the benchmarks presented in Section 2.8.1. The parameters f_h and f_l are known while N_r, N_u, $N_{FR \to SR}$, $N_{SR \to FR}$ can be measured. C and C_{dc} are profiled using *likwid* [135] by accessing the x86 performance counters. Similarly, $T_{FR \to SR}$ and $T_{SR \to FR}$ are profiled using the *FTaLaT* tool [84]. Finally, the transition overhead between the $SlowRel$ and $FastUnRel$ configurations is negligible, since clock adjustment is very fast.

As a first step, we execute all tasks of each application reliably under different configurations $V, f, \vec{c}$, and measure the power consumption. We then perform linear regression using least squares to derive the parameters $\vec{w}$ and P_{idle} of the power model. Finally, we validate the accuracy of our model by forcing the runtime system to execute tasks in different (V, f) configurations. To this end, we execute half of the tasks of each application in the $FastRel = (1.06V, 3.7GHz)$ configuration, and the other half in various lower power but still reliable configurations. The latter are different candidates for $SlowRel$. Cores enter these configurations by applying a software-driven voltage and frequency transition.

Figure 2.10 depicts the relative error of model-based estimates vs. the execution time and energy that was measured using *likwid*. Our model closely matches the

real numbers for various *SlowRel* configurations, with the relative error ranging from 0.004% to 2.7%. In *Jacobi* the increased error is due to load balancing issues. Different executions of the benchmark result in different tasks to worker assignment. This impacts the execution time of the benchmark, hence there is an increase in the relative error.

Note that our x86 platform does not allow placing individual cores in a non-nominal configuration, where actual timing violations and faults might occur. Thus it is impossible to validate the execution time and energy consumption estimates of the model for non-nominal $FastUnRel$ configurations. Still, the accuracy of the model for this wide range of real operating points gives us sufficient confidence to use the model to extrapolate for non-nominal $FastUnRel$ configurations as well.

2.11 Fault Model and Fault Injection Methodology

This section introduces the fault model we use for different unreliable execution. We discuss how we combine simulation-based and software-based fault injection to map the fault rates derived from the model into actual errors at the application level.

Note that it was impossible to conduct our full evaluation using a purely simulated execution platform. Given the vast number of fault injection experiments required to acquire statistically significant results, we would have to limit executions to non-realistic input sizes, despite using a large compute farm for the simulations. Therefore, we adopt a hybrid approach. Initially, we use detailed simulations for injecting faults at the architectural level of an x86 CPU model, and observe the impact they have on each of our benchmarks. Afterwards, we use these observations to drive fault-injection via software when running the benchmarks and our runtime system natively, on our platform. The latter setup, in conjunction with the performance model discussed in Section 2.10, makes it possible to evaluate the fault-tolerance mechanisms that are provided by our framework for different $SlowRel/FastUnRel$ configurations.

2.11.1 Fault modeling

A key challenge is to associate the operation of a core in the unreliable $FastUnRel$ configuration with the probability of hardware faults due to timing violations. Besides undervolting (or overclocking), the number and distribution of faults in a CPU also depends on the type of instructions executed: instructions which activate long paths, which are close to the critical path, tend to fail more frequently [103]. The failure probability of each instruction is also closely related to the micro-architectural design of the CPU, optimizations used by synthesis, placement & routing CAD tools, the manufacturing process and process variability, ambient temperature, IR drops, aging etc. Even identical chips with the same micro-architecture, using the same technology libraries, and running identical code, can have highly different behavior [22]. Moreover, whether a fault manifests as an error does not only depend on

the paths which are activated during the current cycles, but also on the paths which were activated in previous cycles [137].

It is almost impossible to model such complex phenomena in a practical way, as the conclusions are specific to the particular system used to make the observations and create the model, and cannot be generalized to other systems. To the best of our knowledge there is no modern-CPU fault model which combines all the observations in a unified and applicable method. For these reasons, we abstract out the instruction mix of applications, by taking into account only the effects of voltage scaling.

The Point of First Failure (PoFF) is used to indicate the point at which circuits start to exhibit massive errors (one error every $\sim$10 million cycles). Prior to this point errors still occur, however at rates that are orders of magnitude lower [22]. If one goes beyond the PoFF, the fault rate increases exponentially, by one order of magnitude for every $10mV$ drop of the supply voltage [11, 22].

To guarantee functional correctness, designers typically account for parameter variations by imposing conservative margins that guard against worst case scenarios. The extent of the voltage margins required for fault-free operation for all operating conditions of the chip is on average around 15% [36, 108, 50]. We determine the *PoFF* based on Equation 2.5, where ρ is the percentage of the extra voltage margin to guarantee fault-free operation, and V_n is the nominal supply voltage. A CPU part with tight margins has a low ρ and, therefore, low energy benefits when using our approach. We select the average case, $\rho = 15\%$, which is consistent with several observations in the literature [36, 11, 22]. Based on the same reports, we model the fault rate as shown in Equation 2.6. The parameters are the voltage V_{PoFF}, which can be obtained using equation 2.5 using as input argument the nominal voltage V_n and the voltage of the requested (unreliable) operating point V_u. In our case, $V_n = V_h$, the voltage setting of $FastRel$, and $V_u = V_l$ is the voltage setting of the $FastUnRel$ configuration. Our model obtains the constants β, γ via regression on the data provided in [11, 22]. Note that Equation 2.6 is CPU agnostic.

$$V_{PoFF} = \frac{(100 - \rho)}{100} \times V_n \tag{2.5}$$

$$Err(V_{PoFF}, V_u) = \beta \times 10^{\gamma * (V_u - V_{PoFF})} \tag{2.6}$$

$$V_n(f) = \delta \times f + \eta \tag{2.7}$$

Finally, the nominal supply voltage V_n is linearly dependent on the operating frequency, as modeled by Equation 2.7. Parameters δ and, η depend on the system configuration. We deduce their values by monitoring the supply voltage of the CPU of our x86 platform, while commanding changes of the operating frequency.
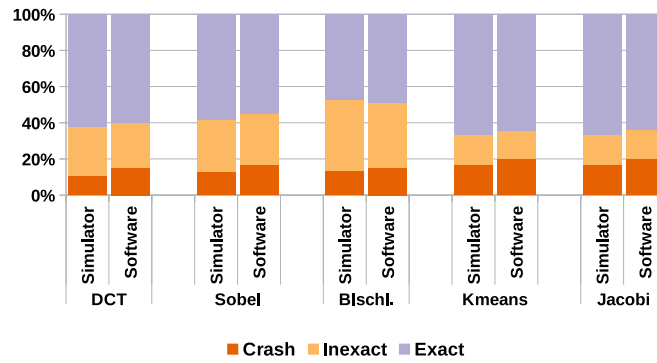
FIGURE 2.11: Effects of single fault injection, using the GemFI simulator at the architectural CPU level, and the software-based approach during native execution.

2.11.2 Simulation-based fault injection

We use the GemFI framework [96] to execute our benchmarks on a simulated out-of-order CPU supporting the x86 ISA. GemFI injects faults at different CPU pipeline stages. In the fetch stage, a fault corrupts a single bit of the instruction. In the decoding stage, the selection of registers is corrupted so that the instruction in question reads from, or writes to a different register. In the execution stage, faults corrupt a single bit of the computed result. Finally, faults in the memory stage corrupt a single bit of the value being transferred from/to memory. Even though we only inject faults to a subset of the CPU modules, these faults can be propagated to the majority of the CPU modules. For example, when a fault is injected during the execution stage of an integer instruction, the fault corrupts the result of the operation. If the result is stored in a register, the fault propagates and corrupts the register file. Also, when injecting a fault to a memory write, the fault can corrupt the data cache hierarchy and even propagate to the main memory. Note that we model transient faults, i.e. the injection of the fault only lasts for one clock cycle.

Simulated fault injection captures the "default" impact of faults on an application executed on top of unreliable hardware, without employing any of the protection mechanisms provided by our framework. Consequently, all application tasks are susceptible to faults, and the result-check functions are ignored. The number of experiments for each application and pipeline stage (see above) is determined based on the methodology described in [75], for a 99% confidence level and 1% error margin. For the purpose of our evaluation, we categorize the outcome of program execution in three bins: (i) crash if the program did not terminate normally, (ii) inexact if the result is not bit-wise identical to that of a reliable execution, and (iii) exact if the result is bit-wise identical to that of a reliable execution. The output of this phase is the probability for a single fault to result in a crash (P_{crash}) for each benchmark. This probability is used by the software fault injection mechanisms during native execution.

2.11.3 Software-based fault injection during native execution

For the native (fast) executions of the benchmarks on our platform, we use software-based fault injection. This is designed to have two possible effects: (i) it forces a crash, and (ii) it corrupts a randomly chosen register of the CPU. The former is done with the probability P_{crash} computed in GemFI simulation, and the latter with probability $1 - P_{crash}$. As in the simulation experiments, all protection mechanisms are disabled, and faults are injected in all application tasks. To validate that software-based fault injection yields realistic results, we compare the outcome of the native executions with the respective outcomes of simulated executions on GemFI. Figure 2.11 which summarizes the results for all benchmarks shows that the software-based fault injection has practically equivalent effects to simulation-based fault injections using GemFI.

Finally, we support native execution scenarios with multiple fault injection. The runtime selectively executes application tasks in reliable or unreliable mode, and where the different protection mechanisms of our framework come into play. The voltage and frequency settings for the $FastRel$ and the $FastUnRel$ configurations are decided as follows. We pick f_h in order to maximize performance, and derive respective nominal voltage V_h from Equation 2.7. We then set $V_l = \epsilon \times V_h | \epsilon < 1.0$. Frequency f_l is derived from Equation 2.7, and the fault rate of the $FastUnRel$ configuration is derived from Equation 2.6, using V_l and V_h as parameters. The rate increases for smaller values of ϵ. Given a target fault rate, we randomly generate a set of fault injection intervals, expressed as number of cycles between faults, using a uniform distribution with a mean value equal to the target fault rate. We then use the performance counter infrastructure of x86 CPUs to interrupt application execution at those intervals and invoke the software-based fault-injection logic. For each application, combination of protection mechanisms, and voltage level (fault rate) we perform $10,000$ multiple fault injection experiments, for a confidence interval of 95% and an error margin of 2.5%.

2.12 Experimental Evaluation

We study the behavior of benchmarks for the different protection levels supported by our runtime system (Section 2.7), using the fault modeling methodology discussed in Section 2.11.1 on our CPU clocked up to 3.70 GHz. We fix the $FastRel$ configuration to the highest performance configuration, with $V_h = 1.06V, f_h = 3.7GHz$. To determine proper $FastUnRel$ configurations, we run our benchmarks for different values for V_l while keeping frequency to f_h, observe their behavior and compute the corresponding energy gains.

Figure 2.12 demonstrates the energy gains for a single task of *Sobel* when executed at different $FastUnRel$ configurations, in comparison with an execution in the $FastRel$ configuration. The "sweet spot" is around $0.88V$. If we further undervolt, inducing faults at higher rates, tasks are practically certain to crash the CPU.
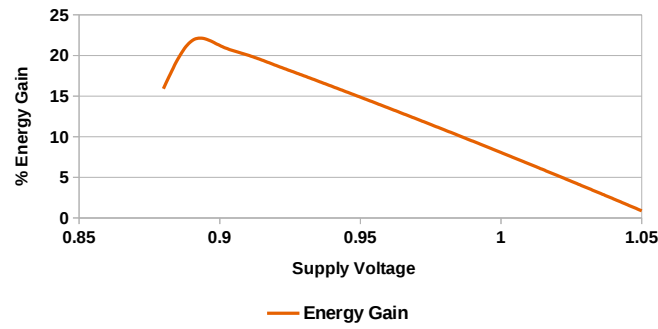
FIGURE 2.12: Energy gains of a single task for *Sobel* executed at voltages $V_l < V_h$ for constant frequency $f_h = 3.7GHz$.

SlowRel		FastUnRel		
Freq.	**Voltage**	**Freq.**	**Voltage**	**Fault Rate**
1.67 GHz	0.90V		0.90V	10^{-7}
1.54 GHz	0.89V	3.7 GHz	0.89V	10^{-6}
1.41 GHz	0.88V		0.88V	10^{-5}

TABLE 2.2: *SlowRel* and *FastUnRel* configuration settings used in our evaluation, and average fault rates of the *FastUnRel* configurations.

This increases the overhead due to the activation of protection and task correction mechanisms in the *SlowRel* configuration, and cancels any energy gains. In contrast, when a core operates in voltage regions higher than the PoFF, the failure rate is very small, and the functionality of our framework is rarely activated. Since these effects are observed in all the application benchmarks in our evaluation , we focus on the "promising" voltage range from $0.88V$ to $0.90V$. In our evaluation we set $FastRel = (1.06V, 3.7Ghz)$. Table 2.2 summarizes the configurations used in our experiments.

Figure 2.13 summarizes our experimental results for a range of voltage settings and protection mechanisms. For each benchmark we present two diagrams. The

Benchmark.	C	N_r	N_u	$N_{FR \to SR}$	$N_{SR \to FR}$
DCT	133K	4096	28672	1	1
Sobel	50K	410	3684	1	1
Blscls	197K	90	10	1	1
Kmeans	283K	1500	13500	83	83
Jacobi	594K	830	7470	83	83

TABLE 2.3: Average task execution time in cycles (thousands), number of tasks executed reliably/unreliably, and number of voltage and frequency transitions, for each benchmark.
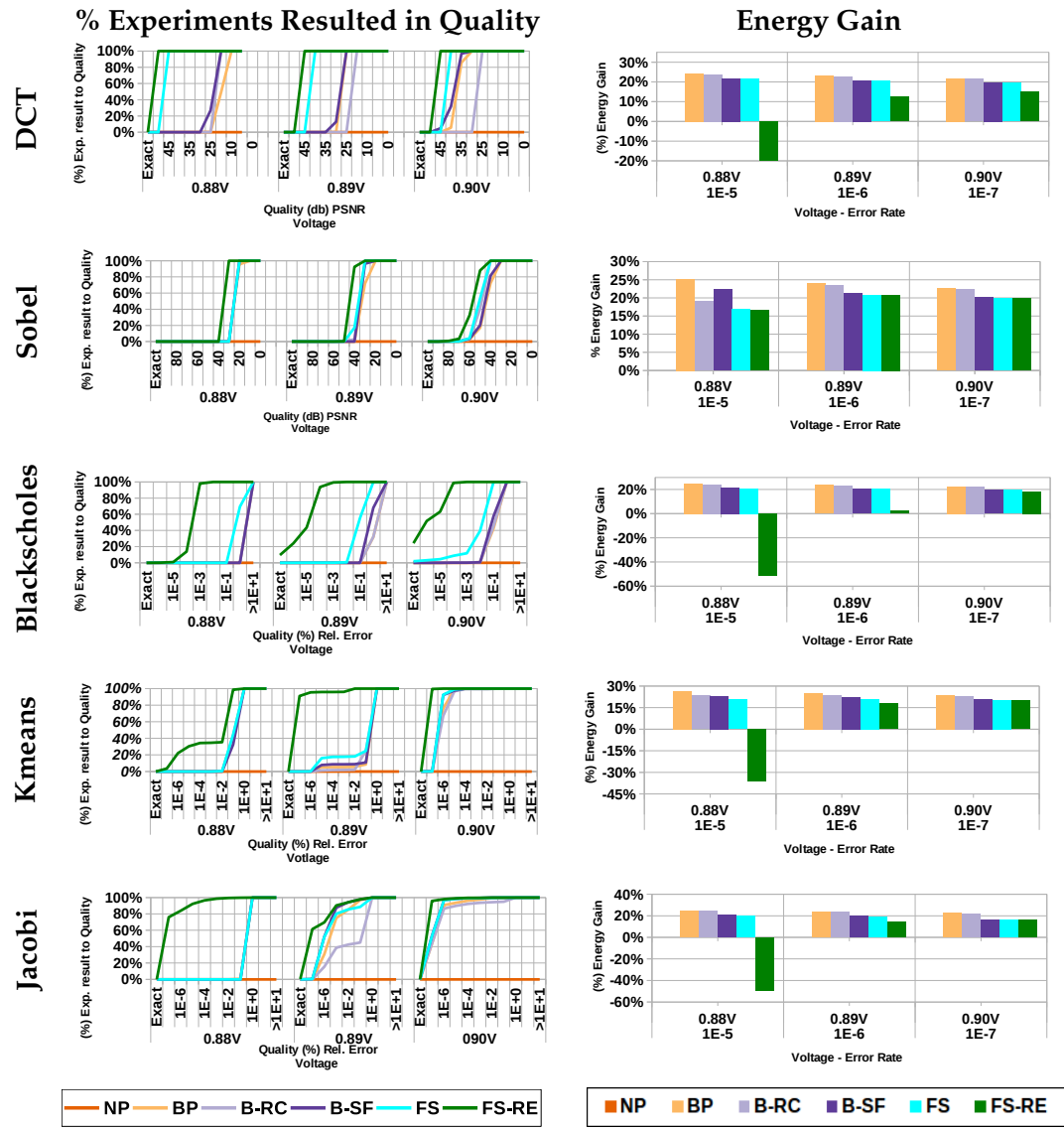
FIGURE 2.13: Experimental results for different V_l values for the *SlowRel* and *FastUnRel* configurations. Percentage of experiments which achieved a certain quality (left), and energy gains with each protection scenario (right).

left one depicts the *cumulative distribution function (CDF)* of the percentage of experiments (y-axis) achieving a specific quality threshold (x-axis) under different protection mechanisms (different lines). For the media benchmarks (DCT, Sobel) the quality metric is PSNR (higher value is better). For the remaining benchmarks quality is quantified by the relative error w.r.t the fully reliable execution (lower value is better). The two extreme bins of the x-axis correspond on the one side to experiments which resulted in bitwise exact results, and on the other side to experiments producing very bad output quality. The percentage of crashed experiments can be deduced by subtracting the percentage of worst quality experiments from 100%. The percentage of experiments which resulted in bitwise exact results are equal to the percentage of experiments which provide the best quality in the *CDF*. For a specific quality target, the height of each CDF line at the specific quality corresponds to the

percentage of experiments which achieve the specific quality of results. Intuitively, the sooner (to the left) and the higher the lines raise, the better the respective protection configurations.

The diagrams to the right depict the average energy gains against a fully reliable execution ($FastRel$ state) using our runtime in the NP configuration. The number of voltage and frequency transitions, the average execution time of a task in cycles as well as the number of reliable and non reliable tasks are given in Table 2.3. In all scenarios where task significance information is taken into account, the task ratio is fixed to 10%, except DCT in which the requested ratio is 13%. In DCT all tasks which compute the upper left coefficient corner need to be executed reliably. These tasks correspond to the 13% of the total number of tasks. In scenarios that do not exploit significance information, all tasks are executed unreliably.

When no protection mechanism is active, all experiments result in crashes. Basic protection (BP) eliminates crashes, and can even lead to satisfactory behavior as long as the fault rate remains moderate. As expected, error resilience increases as more protection mechanisms are employed. As an exception, result-check functions (*B-RC*) may produce worse results compared with *BP*, by discarding partially good results produced by tasks before they crash. On the other hand, energy gains are typically reduced as the amount of protection increases. Therefore, we select naive result check (*RC*) functions, which do not spend a lot of time to detect and correct an error. This increases the energy gains, however it decreases the quality of the end result. Another interesting observation is that task re-execution (*FS-RE*) does not guarantee perfect results, as is clearly visible from the CDFs in Figure 2.13. A task is re-executed reliably only if it crashes or the result check function requests a re-execution. Since the result check functions are simple they miss silent data corruptions, which in turn lead to imperfect results. Finally, when combining all protection mechanisms, the application error resiliency is pushed to significantly higher fault rates. In the following paragraphs, we discuss the behavior of each application in more detail.

The two image processing benchmarks demonstrate a similar behavior. The transition from *NP* to *BP* completely eliminates any program crashes. However, there is no guarantee for the quality of the output. The produced outputs are of unacceptable quality when executed in all $FastUnRel$ configurations. Even the addition of a *result check function (B-RC)* doest not increase the quality; the same is observed for the *B-SF* scenario. In DCT *B-RC* the detection part of the result check function is able to detect many errors, however, when errors corrupt tasks that should had been significant, there is no efficient way to correct them. This motivates the usage of the significance information by our runtime system. On the other hand, in *Sobel* the detection part is incapable of detecting many errors. In the *B-SF* scenario significant tasks are protected by the software stack, however there is no increase in the quality of the output. In the case of *DCT* the absence of a result check function allows faults that manifest on non-significant tasks to negatively impact the end quality.
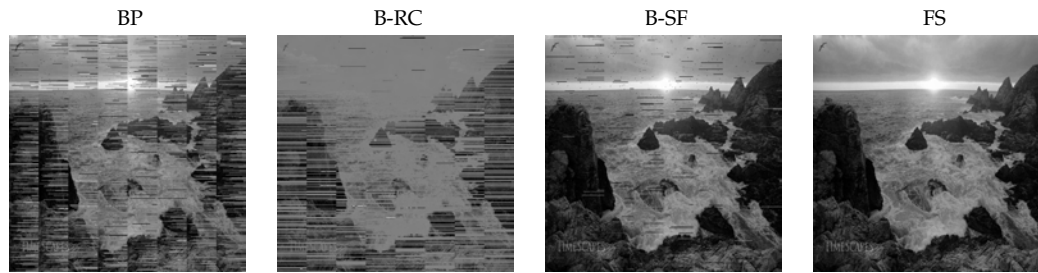
FIGURE 2.14: *DCT* output at 0.89V, with one fault injected every 100,000 cycles. The images correspond (from left to right) to the *BP*, *B-RC*, *B-SF* and *FS* protection configurations, resulting to PSNRs of 12, 13, 15 and 37 dB respectively. A fault free execution leads results in a PSNR of 43 dB. *NP* deterministically leads to crashes.

Figure 2.14 illustrates the output of four protection configurations (excluding *NP* and *FS-RE*) for the *DCT* benchmark. The corrupted images show the effect of faults when protection is not adequate, while the rightmost image shows that even in a highly faulty environment, our approach almost eliminates visible artifacts. In *Sobel* the significance characterization of tasks simply spreads unreliability uniformly within the output, however *PSNR* does not capture such effects. It is interesting to note that for *Sobel* at $0.88V$ the *B-RC* leads to smaller energy gains than *B-SF*. Under such high error rates tasks tend to crash frequently, which is detectable by the runtime system and therefore the correction part is invoked. However in *Sobel* the correction part of the result check function is almost as costly as the task itself (Table 2.3), so correcting a large number of tasks incurs excessive overhead. The combination of the result check function with the significance values (*FS* scenario) results in increased quality. Even in the highest fault rates the *FS* scenario delivers quality higher than *35* dB for *DCT* and *30* dB *Sobel*, respectively, for all experiments. Similar behavior is observed for the *FS-RE* scenario. In the case of *Sobel*, the detection part of the result-check function is unable to detect most faults except the ones which lead to task crashes. Therefore the correction part (re-execution in FS-RE) is rarely executed. Consequently, the negative (energy-wise) impact of the re-execution is not captured in this benchmark.

In the *FS* configuration when the voltage is decreased from $0.90V$ to 0.88, the energy gains of *DCT* slightly increase from 18% to 21% whereas in *Sobel* the energy gain is reduced from 20.0% to 16.0% The result check function of *DCT* sets a default value (0) to the erroneous output. For *Sobel*, an approximate version of the task is executed. Therefore, the energy gains due to undervolting are eliminated by executing the result check function more frequently due to the higher fault rate. A similar trend is observed for *DCT* in the *FS-RE* configuration. Re-executing the entire task every time its output is detected as erroneous outweighs all energy savings and results in energy losses.

The computationally intensive *Blackscholes* uses mathematical functions, such as logarithms, square roots, etc. which return *NaN* or *inf* when arguments are outside
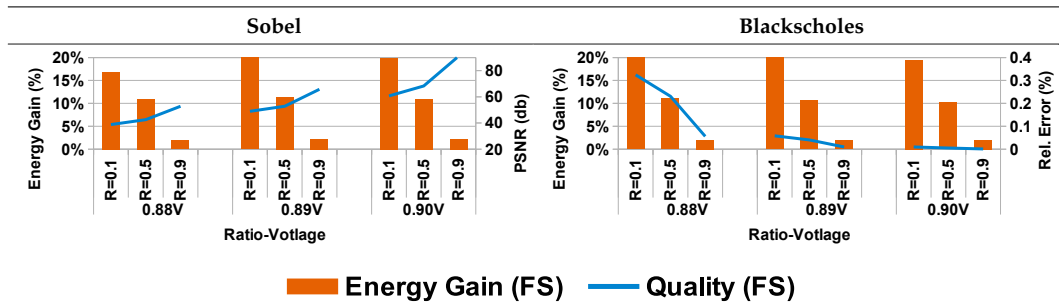
FIGURE 2.15: Quality vs. energy trade-offs using the *ratio* parameter in the FS configuration.

their definition range. Detecting such errors is easy. Since many errors can be detected, *FS-RE* computes exact outputs 24% of the time when operating at $0.90V$. The resulting output quality is exceptional with a relative error less than 0.03% across all experiments. However, at high fault frequencies the application results in energy losses since a large number of tasks need to be re-executed in the *SlowRel* domain.

K-means and *Jacobi* demonstrate similar characteristics. At $0.90V$, in all protection scenarios, both applications result in a relative error less than $10^{-6}\%$. In *K-means* the quality decreases rapidly for higher fault rates. Neither the result check function nor the significance values increase output quality. The result check function has no efficient way to correct errors and the small subset of the last significant iterations is unable to assign the points to the correct centers. For *Jacobi*, at $0.89V$ *BP* has better quality than *B-RC*. In *B-RC*, when an error is detected, the current solution estimate is replaced with that of the previous iteration. At high fault rates errors are frequently detected and therefore the respective iterations are discarded. In *Jacobi* it is better to rely on the self healing attributes than correcting the result.

Figure 2.15 presents experiments in which we vary the *ratio* parameter and record the energy savings and output quality for different values, for the *Sobel* and *Blackscholes* benchmarks under the FS configuration. The *ratio* knob allows the user to select the percentage of reliably executed tasks and can effectively control the trade off between energy savings and quality loss. A similar behavior is observed in all benchmarks.

Note that energy gains are best obtained when shaving voltage guard bands to the point of first failure. More aggressive voltage scaling diminishes improvements in energy efficiency, due to the overhead of error detection and correction.

Chapter 3

Significance Aware Approximate Computing

In this chapter we present the approximate version of our significance aware approach. Although, the significant-aware *fault tolerant* programming model provides significant energy gains it is hard for the developer to completely comprehend the interactions between significance identification, ratio, error manifestation and error detection and correction. Developers are more familiar with the *approximate* computing, as it presents a more intuitive trade-off between quality and energy. To this direction we port our *fault tolerant* programming model to support *approximate* computing. The semantics of our approach, which is the *significance* clause and the *ratio* clause remain the same but we allow the developer to define alternate approximate more energy efficient functions. We vision the *approximate* programming model as an intermediate step prior adopting the *fault tolerant* one. Developers, using the *approximate* variant, can grasp the interactions between the significance tagging and the *ratio* clause.

3.1 Contributions

In this chapter we make the following contributions:

- To reduce the energy footprint of applications we port the significance - aware *fault tolerant* programming model to support approximate computing. In this variant of the programming model the developer can supply approximate versions of non-significant tasks; The basis of the programming model is similar with the *fault tolerant* programming model, however the mechanisms to reduce the energy consumption are slightly different.

- We experimentally evaluate our approach and show that it is superior to loop perforation [125], a compiler-based approximation technique.

3.2 Programming Model

The significance-aware for approximate computing programming model is similar to the programming model described in the chapter 2. Both programming models use the same syntax and semantics to define the significance of the computations, parallelism and the synchronization of the tasks. The main difference is that the developer instead of providing functions for error-detection and correction she should provide an approximate version of the original task version.

```c
int sblX(byte *img, int y, int x) {
  return img[(y-1)*WIDTH+x-1]
    + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
    - img[(y-1)*WIDTH+x+1]
    - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
}

int sblX_appr(byte *img, int y, int x) {
  return /* img[(y-1)*WIDTH+x-1]   Ommited taps */
    + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
    /* - img[(y-1)*WIDTH+x+1]   Ommited taps *//
    - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
}

/* sblY and sblY_appr are similar */
void row_acc(byte *res, byte *img, int i) {
  unsigned int p, j;
  for (j=1; j<WIDTH-1; j++) {
    p = sqrt(pow(sblX(img, i, j),2) +
             pow(sblY(img, i, j),2));
    res[i*WIDTH + j] = (p > 255) ? 255 : p;
  }
}

void row_appr(byte *res, byte *img, int i) {
  unsigned int p, j;
  for (j=1; j<WIDTH-1; j++) {
    /* abs instead of pow/sqrt,
       approximate versions of sblX, sblY */
    p = abs(sblX_appr(img, i, j) +
            sblY_appr(img, i, j));
    res[i*WIDTH + j] = (p > 255) ? 255 : p;
  }
}

double sobel(void) {
  int i;
  byte img[WIDTH*HEIGHT], res[WIDTH*HEIGHT];
  /* Initialize img array and reset res array */
  ...
  for (i=1; i<HEIGHT-1; i++)
    #pragma omp task label(sobel) approxfun(row_appr) \
      in(img[i*WIDTH:(i+1)*WIDTH-1]) \
      out(res[i*WIDTH:(i+1)*WIDTH-1]) \
      significant((i%9 + 1)/10.0)
    row_acc(res, img, i); /* Compute a single
                             output image row */
  #pragma omp taskwait label(sobel) ratio(0.35)
}
```

LISTING 3.1: Programming model use case: Sobel filter

Listing 3.1 depicts our programming model in the implementation of the *Sobel* filter, which we use as a running example.

```
#pragma omp task [significant(...)] [label(...)]
    [in(...)] [out(...)] [approxfun(function())]
```

LISTING 3.2: #pragma omp task

Tasks are specified using the *#pragma omp task* directive (Listing 3.2), followed by the task body function. The *significant, label, in, out* clauses have the same semantics as the ones used in the *fault tolerant* version.

For tasks with significance less than 1.0, the programmer may provide an alternative, approximate task body, through the *approxfun()* clause. This function is executed whenever the runtime opts to approximate a task. It typically implements a simpler version of the computation in the task, which may even degenerate to setting default values for the task output. If the runtime system decides to execute a task approximately and the programmer has not supplied an *approxfun* version, the task is dropped. The *approxfun* function implicitly takes the same arguments as the function implementing the accurate version of the task body.

As an example, lines 41-46 of Listing 3.1 create a separate task to compute each row of the output image. The significance of the tasks gradually ranges between 0.1 and 0.9 (line 45), so that there are no extreme quality fluctuations across the output image. The approximate function *row_appr* implements a lightweight version of the computation. All tasks created in the specific loop belong to the *sobel* task group, using *img* as input and *res* as output (lines 43-44).

```
#pragma omp taskwait [label(...)] [ratio(...)]
```

LISTING 3.3: #pragma omp taskwait

Explicit barrier-like synchronization is supported via the *#pragma omp taskwait* directive (Listing 3.3). The synchronization primitive has the same semantics as the *unrelible* one, however the user can neither define a group result check function nor specify a timing constraint.

As an example, line 48 of Listing 3.1 specifies a barrier for the tasks of the *sobel* task group. The runtime needs to ensure that at a minimum, the most significant 35% of the group tasks are executed accurately. Note that the runtime may opt for a higher ratio, provided this is feasible with the energy budget of the program.

3.3 Runtime support for significance aware approximate computing

We demonstrate how to extend existing runtime systems to support our programming model for approximate computing. To this end, we extend a task-based parallel runtime system that implements OpenMP 4.0-style task dependencies [136].

Our runtime system is organized as a master/slave work-sharing scheduler. The master thread starts executing the main program sequentially. For every task call encountered, the task is enqueued in a per-worker task queue. Tasks are distributed across workers in round-robin fashion. Workers select the oldest tasks from their queues for execution. When a worker's queue runs empty, the worker may steal tasks from other workers' queues.

The runtime system furthermore implements an efficient mechanism for identifying and enforcing dependencies between tasks that arise from annotations of the side effects of tasks with *in(...)* and *out(...)* clauses. Dependence tracking is however orthogonal to our approximate computing programming model. Therefore, we provide no further details on this feature.

The job of the runtime system is to selectively execute a subset of the tasks approximately while respecting the constraints given by the programmer. The relevant information consists of (i) the significance of each task, (ii) the group a task belongs to, and (iii) the fraction of tasks that may be executed approximately for each task group. Moreover, preference should be given to approximating tasks with lower significance values as opposed to tasks with high significance values.

The runtime system has no a priori information on how many tasks will be issued in a task group, nor what the distribution is of the significance levels in each task group. This information must be collected at runtime. In the ideal case, the runtime system knows this information in advance. Then, it is straightforward to execute approximately those tasks with the lowest significance in each task group. We have designed two runtime policies which work without this information, and estimate it at runtime [138]. *Global Task Buffering (GTB)* is a globally controlled policy based on buffering issued tasks and analyzing their properties. *Local Queue History (LQH)* estimates the distribution of significance levels using per-worker local information[1]. In *GTB* the master thread stores a number of tasks as it creates them in a buffer, postponing the issue of the tasks in the worker queues. When the buffer is full it sorts the tasks based on their significance. Given a per-group ratio of accurate tasks R, and a number of B tasks in the buffer, then the $R * B$ tasks with the highest significance level are executed accurately. The *LQH* policy avoids the step of task buffering. Tasks are issued to worker queues immediately as they are created. The worker decides whether to approximate a task right before it starts its execution, based on the distribution of significance levels of the tasks executed so far, and the target ratio of accurate tasks (supplied by the programmer).

3.3.1 Life of a group-of-tasks

Figure 3.1 illustrates the typical life of a group-of-tasks in an application implemented using our significance aware approximate computing programming model. For each group instantiated during the life of an application, the runtime system

---

[1]Both policies (GTB, LQH) are implemented by Charalampos Chalios.
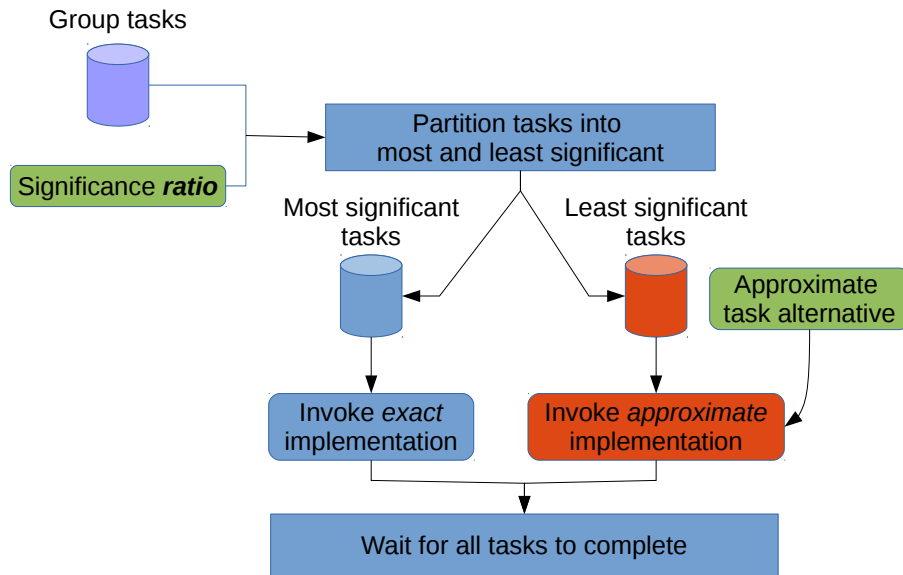
FIGURE 3.1: The typical life of a group-of-tasks in the context of significance aware approximate computing

receives a collection of tasks with varying significance values and a desired approximation level in the form of a significance *ratio*. Afterwards, the runtime system partitions the tasks into two sets, the most significant tasks and the least significant ones. The most significant ones are executed in an accurate way, whereas the runtime system invokes the approximate implementation for the least significant ones.

3.3.2 Approximate vs Fault Tolerant Runtime Support

A large portion of the approximate and fault tolerant runtime systems are identical, as they support similar features. The *approximate* variant does not require any protection mechanism such as memory management (2.7.2), relaxed synchronization, soft check-pointing or taking into account the operating settings of the CPU. Regarding the task scheduling, since the *approximate* variant does not require specific CPU settings in terms of operating frequency and voltage, tasks are flagged as ready for execution right after their final significance is decided. On the other hand, to minimize the number of transitions between the $FastRel$ and the $FastUnRel$ settings the *fault tolerant* runtime needs to wait for all tasks of a group to be instantiated prior starting the execution of tasks.

3.4 Experimental Evaluation

We performed a set of experiments to investigate the performance of the proposed programming model and runtime policies, using different benchmark codes that were re-written using the task-based pragma directives. In particular, we evaluate our approach in terms of: (i) The potential for performance and energy reduction; (ii) The potential to allow graceful quality degradation; (iii) The overhead incurred

Benchmark	**A**pproximate or **D**rop	Approx Degree			Quality
		Mild	Med	Aggr	
Sobel	A	80%	30%	0%	PSNR
DCT	D	80%	40%	10%	PSNR
MC	D, A	100%	80%	50%	Relative Error
K-means	A	80%	60%	40%	Relative Error
Jacobi	D, A	10^{-4}	10^{-3}	10^{-2}	Relative Error
Fluidanimate	A	50%	25%	12.5%	Relative Error

TABLE 3.1: Benchmarks used for the evaluation. For all cases, except *Jacobi*, the approximation degree is given by the percentage of accurately executed tasks. In *Jacobi*, it is given by the error tolerance in convergence of the accurately executed iterations/tasks (10^{-5} in the native version).

by the runtime mechanisms. In the sequel, we introduce the overall evaluation approach, and discuss the results achieved for various degrees of approximation under different runtime policies.

3.4.1 Approach

We use a set of six benchmarks, outlined in Table 3.1, where we apply different approximation approaches, subject to the nature/characteristics of the respective computation.

The approximate version of the *Sobel* tasks uses a lightweight *Sobel* stencil with just 2/3 of the filter taps. Additionally, it substitutes the costly formula $\sqrt{sbl_x{}^2 + sbl_y^2}$ with its approximate counterpart $|sbl_x| + |sbl_y|$. The way of assigning significance to tasks ensures that the approximated pixels are uniformly spread throughout the output image.

We assign higher significance to tasks that compute lower frequency coefficients for the tasks of Discrete Cosine Transform (*DCT*) [141].

For Monte Carlo (*MC*) a modified, more lightweight, methodology is used to decide how far from the current location the next step of a random walk should be.

Approximated K-Means tasks compute a simpler version of the euclidean distance, while at the same time considering only a subset (1/8) of the dimensions. Only accurate results are considered when evaluating the convergence criteria.

In *Jacobi*, we execute the first 5 iterations approximately, by dropping the tasks (and computations) corresponding to the upper right and lower left areas of the matrix. This is not catastrophic, due to the fact that the matrix is diagonally dominant and thus most of the information is within a band near the diagonal. All the following steps, until convergence, are executed accurately, however at a higher target error tolerance than the native execution (see Table 3.1).

In *Fluidanimate*, each time step is executed as either fully accurate or fully approximate, by setting the *ratio* clause of the *omp taskwait* pragma to either 0.0 or 1.0.
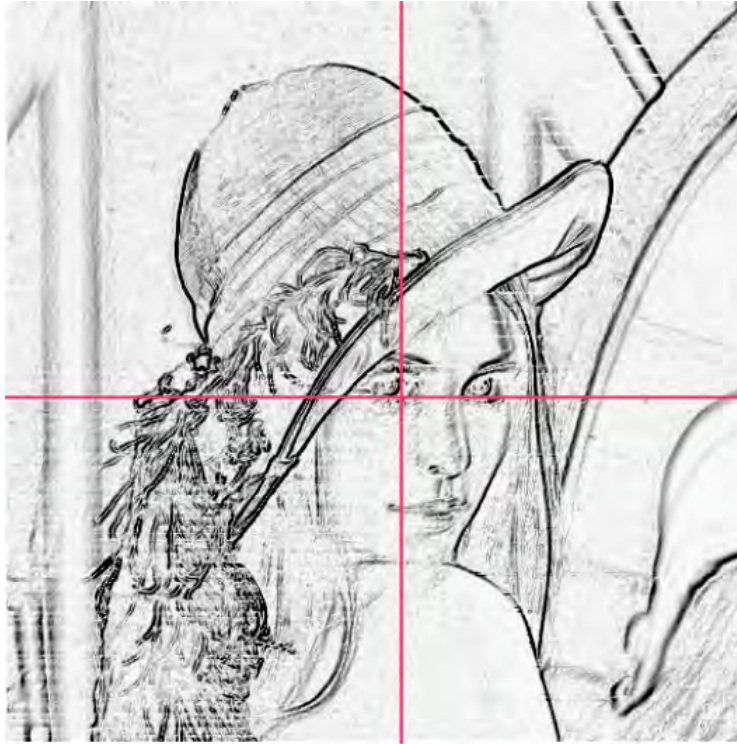
FIGURE 3.2: Different levels of approximation for the *Sobel* benchmark

In the approximate execution, the new position of each particle is estimated assuming it will move linearly, in the same direction and with the same velocity as it did in the previous time steps.

Three different degrees of approximation are studied for each benchmark: *Mild*, *Medium*, and *Aggressive* (see Table 3.1). They correspond to different choices in the quality vs. energy and performance space. No approximate execution led to abnormal program termination. It should be noted that, with the partial exception of *Jacobi*, quality control is possible solely by changing the *ratio* parameter of the *taskwait* pragma. This is indicative of the flexibility of our programming model. As an example, Figure 3.2 visualizes the results of different degrees of approximation for *Sobel*: the upper left quadrant is computed with no approximation, the upper right is computed with *Mild* approximation, the lower left with *Medium* approximation, whereas the lower right corner is produced when using *Aggressive* approximation.

The quality of the final result is evaluated by comparing it to the output produced by a fully accurate execution of the respective code. The appropriate metric for the quality of the final result differs according to the computation. For benchmarks involving image processing (*DCT, Sobel*), we use the peak signal to noise ratio (*PSNR*) metric, whereas for *MC, K-means, Jacobi* and *Fluidanimate* we use the relative error.

In the experiments, we measure the performance of our approach for the different benchmarks and approximation degrees, for the two different runtime policies GTB and LQH. For GTB, we investigate two cases: the buffer size is set so that tasks

are buffered until the synchronization barrier (referred to as Max Buffer GTB) ; the buffer size is set to a smaller value, depending on the computation, so that task execution can start earlier (referred to as GTB).

As a reference, we compare our approach against:

- A fully accurate execution of each application, using a significance agnostic version of the runtime system.

- An execution using loop perforation [125], a simple yet usually effective compiler technique for approximation. Loop perforation is also applied in three different degrees of aggressiveness. The perforated version executes the same number of tasks as those executed accurately by our approach.

The experimental evaluation is carried out on a system equipped with 2 *Intel(R) Xeon(R) CPU E5-2650* processors clocked at 2.00 GHz, with 64 GB shared memory. Each CPU consists of 8 cores. Although cores support SMT execution (hyperthreading), we deactivated this feature during our experiments. We use Centos 6.5 Linux Operating system with the 2.6.32 Linux kernel. Each execution pinned 16 threads on all 16 cores.

Finally the energy and power are measured using likwid [135] to access the Running Average Power Limit (RAPL) registers of the processors.

3.4.2 Experimental Results

Figure 3.3 depicts the results of the experimental evaluation of our system. For each benchmark we present execution time, energy consumption and the corresponding error metric.

The approximated versions of the benchmarks execute significantly faster and with less energy consumption compared to their accurate counterparts. Although the quality of the application output deteriorates as the approximation level increases, this is typically done in a graceful manner, as it can be observed in Figure 3.2 and the '*Quality*' column of Figure 3.3.

The GTB policies with different buffer sizes are comparable with each other. Even though Max buffer GTB postpones task issue until the creation of all tasks in the group, this does not seem to penalize the policy. In most applications tasks are coarse-grained and are organized in relatively small groups, thus minimizing the task creation overhead and the latency for the creation of all tasks within a group. LQH is typically faster and more energy-efficient than both GTB flavors, except for *K-means*.

In the case of *Sobel*, the perforated version seems to significantly outperform our approach in terms of both energy consumption and execution time. However the cost of doing so is unacceptable output quality, even for the mild approximation level as shown in Figure 3.4. Our programming model and runtime policies achieve graceful quality degradation, resulting to acceptable output even with aggressive approximation, as illustrated in Figure 3.2.
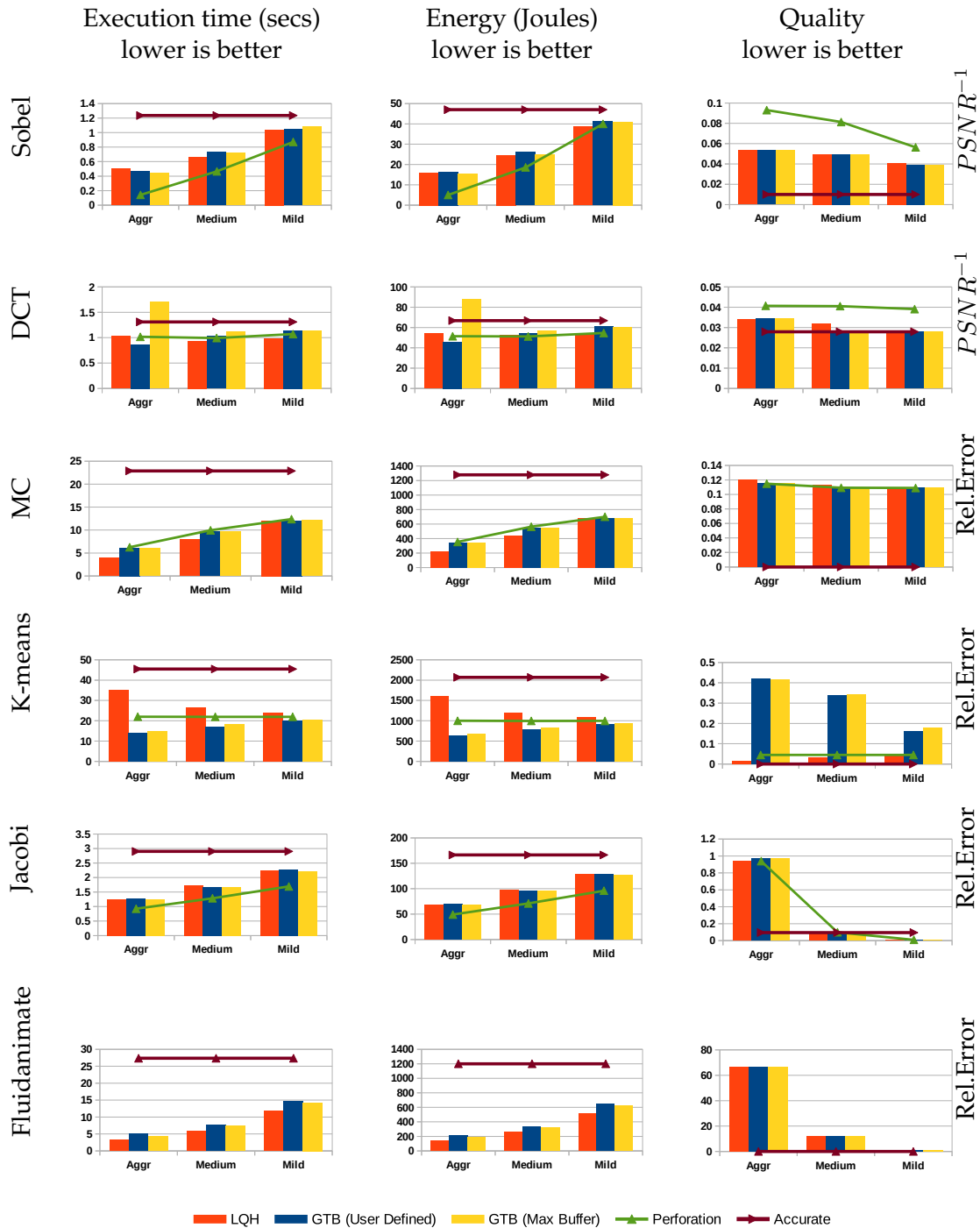
FIGURE 3.3: Execution time, energy and quality of results for the benchmarks used in the experimental evaluation under different runtime policies and degrees of approximation. In all cases lower is better. Quality is depicted as $PSNR^{-1}$ for *Sobel* and *DCT*, relative error (%) is used in all others benchmarks. The accurate execution and the approximate execution using perforation are visualized as lines. Note that perforation was not applicable for Fluidanimate.

DCT is friendly to approximations: it produces visually acceptable results even if a large percentage of the computations is dropped. Our policies, with the exception of the Max Buffer version of GTB, perform comparably to loop perforation in

FIGURE 3.4: Different levels of perforation for the *Sobel* benchmark. Accurate execution, Perforation of 20%, 70% and 100% of loop iterations on the upper left, upper right, lower left and lower right quadrants respectively.

terms of performance and energy consumption, yet resulting to higher quality results[2]. This is due to the fact that our model offers more flexibility than perforation in defining the relative significance of code regions in *DCT*. The problematic performance of GTB(Max Buffer) is discussed later in this Section, when evaluating the overhead of the runtime policies and mechanisms.

The approximate version of *MC* significantly outperforms the original accurate version, without suffering much of a penalty on its output quality. Randomized algorithms are inherently susceptible to approximations without requiring much sophistication. It is characteristic that the performance of our approach is almost identical to that of blind loop perforation. We observe that the LQH policy attains slightly better results. In this case, we found that the LQH policy undershoots the requested ratio, evidently executing fewer tasks [3]. This affects quality, which is lower than that achieved by the rest of the policies.

K-means behaves gracefully as the level of approximation increases. Even in the aggressive case, all policies demonstrate relative errors less than 0.45%. The GTB policies are superior in terms of execution time and energy consumption in comparison with the perforated version of the benchmark. Noticeably, the LQH policy exhibits slow convergence to the termination criteria. The application terminates

---

[2]Note that PSNR is a logarithmic metric

[3]4.6% and 5.1% more that requested tasks are approximated for the aggressive and the medium case respectively.
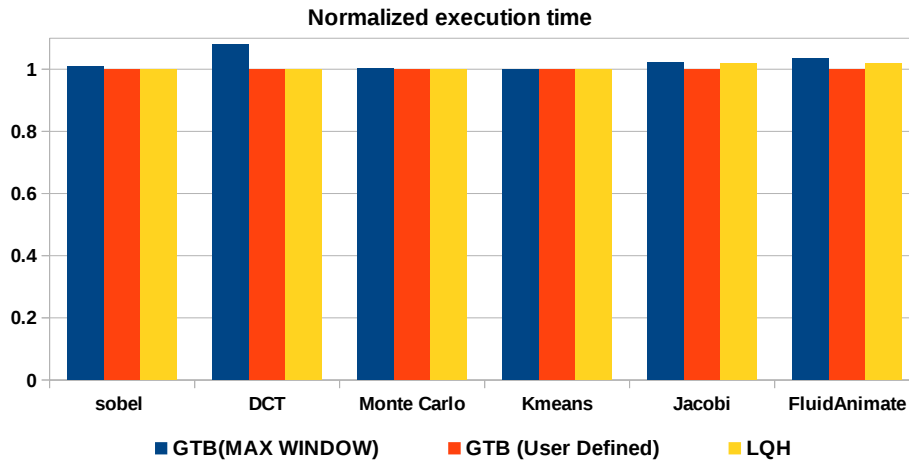
FIGURE 3.5: The normalized execution time of benchmarks under different task categorization policies, with respect to that over the significance-agnostic runtime system

when the number of objects which move to another cluster is less than 1/1000 of the total object population. As mentioned in the Section 3.4.1, objects which are computed approximately do not participate in the termination criteria. GTB policies behave deterministically, therefore always selecting tasks corresponding to specific objects for accurate executions. On the other hand, due to the effects dynamic load balancing in the runtime and its localized perspective, LQH tends to evaluate accurately different objects in each iteration. Therefore, it is more challenging for LQH to achieve the termination criterion. Nevertheless, LQH produces results with the same quality as a fully accurate execution with significant performance and energy benefits.

Jacobi is an application with unique characteristics: approximations can affect its rate of convergence in deterministic, yet hard to predict and analyze ways. The blind perforation version requires fewer iterations to converge, thus resulting to lower energy consumption than our policies. Interestingly enough, it also results to a solution closer to the real one, compared with the accurate execution.

The perforation mechanism could not be applied on top of the *Fluidanimate* benchmark. This is because if the evaluation of the movement of part of the particles during a time-step is totally dropped, the physics of the fluid are violated leading to completely wrong results. Our programming model offers the programmer the expressiveness to approximate the movement of the liquid for a set of time-steps. Moreover, in order to ensure stability, in is necessary to alternate accurate and approximate time steps. In our programming model this is achieved in a trivial manner, by alternating the parameter of the *ratio* clause at *taskbarrier* pragmas between 100% and the desired value in consecutive time steps. It is worth noting that *Fluidanimate* is so sensitive to errors that only the mild degree of approximation leads to acceptable results. Even so, the LQH policy requires less than half the energy of the accurate execution, with the 2 versions of the GTB policy being almost as efficient.

Next, we evaluate the overhead of the two runtime policies and mechanisms.

We measure the performance of each benchmark when executed with a significance-agnostic version of the runtime system, which does not include the execution paths for classifying and executing tasks according to significance. We then compare it with the performance attained when executing the benchmarks with the significance-aware version of the runtime. All tasks are created with the same significance and the ratio of tasks executed accurately is set to 100%, therefore eliminating any benefits of approximate execution. Figure 3.5 summarizes the results. It is evident that the significance-aware runtime system typically incurs negligible overhead. The overhead is in the order of 7% in the worst case (*DCT* under the GTB Max Buffer policy). *DCT* creates many lightweight tasks, therefore stressing the runtime. At the same time, given that for *DCT* task creation is a non-negligible percentage of the total execution time, the latency between task creation and task issue introduced by the Max Buffer version of the GTB policy results to a measurable overhead.

Chapter 4

Modeling and Prediction of Voltage Margins in Multicore CPUs

In this chapter we describe an end-to-end methodology to exploit CPU margins to increase energy efficiency of modern CPUs. The methodology identifies and sets a new CPU supply voltage which is lower than the nominal one. The objectives of the methodology are:

1. **Error avoidance.** The new CPU supply voltage should be lower than the nominal one to reduce the power consumption of the system but should be high enough so that errors do not manifest during the execution of various workloads.

2. **Source code agnostic.** The methodology can be applied without any knowledge about the source code of the applications.

In contrast to the previous chapters in this chapter we avoid errors and source code modifications. Many applications are not error resilient and they require 100% quality of the output and the source code of multiple applications is not always available. Our methodology, presented in this chapter, overcomes these difficulties.

4.1 Background

In this chapter we seek to identify an operating point for the CPU in which the CPU consumes less energy and applications execute without errors. Any CPU operates with a supplied voltage (V) and an operating frequency (f), the pair (V, f) is referred as an operating point. Typically, a CPU manufacturer defines a finite number of operating points, called Nominal Operating Points (NOP) which guarantee errorless operation as presented in Equation 4.1. The M denotes the total number of nominal operating points for a given CPU. In the context of this thesis we refer to any of the nominal points as (V_n, f_n). Executing a workload[1] (W) on top of an operating point (V, f), is defined using the operator presented in Equation 4.2. The Quality of

---

[1]The workload depends on the executable code and the inputs to the code.

Output *QoO* of a workload when executed on a NOP is always considered as *Bitwise Exact (BE)* (Equation 4.3).

$$NOP = \{(V_0, f_0), \ldots, (V_m, f_m) \mid 0 < m < M\} \tag{4.1}$$

$$output = Exec(W, (V, f)) \tag{4.2}$$

$$output = Exec(W, (V_n, f_n)) \implies QoO(output) = BE \tag{4.3}$$

Creating a more energy efficient operating point of a CPU is feasible using two methods. For a given nominal operating point (V_n, f_n) one can create a new more energy efficient operating point by:

Undervolting: (V_u, f_n), reduce the supply voltage V_u below V_n ($V_u < V_n$), while keeping the operating frequency unchanged.

Overclocking: (V_n, f_o), increase the operating frequency f_o beyond f_n ($f_o > f_n$), and keep the supply voltage unchanged.

In the context of this thesis we refer to the amount of undervolting or overclocking as margin. The voltage margin (V_{margin}) is defined in Equation 4.4 and is the distance between the nominal supply voltage (V_n) and the undervolted supply voltage (V_u). Similarly, in the case of overclocking (f_{margin}) as defined in Equation 4.5 refers to the distance of the nominal operating frequency (f_n) with the overclocked (f_o) frequency.

$$V_{margin} = V_n - V_u \mid V_u \in [0, V_n) \tag{4.4}$$

$$f_{margin} = f_o - f_n \mid f_o \in (f_n, \infty) \tag{4.5}$$

Executing applications on CPUs operating at reduced margins comes with risks as the reliability of the system is reduced. Figure 1.1 depicts the various operating points regions CPU. The more the voltage reduction or the frequency increase from the NOP (green dashed line) the higher the reliability reduction. The boundary between the *safe* and *unsafe* region is dependent on the microarchitecture, the specific chip part and the executed workload. Equations 4.6, 4.7 define the maximum V_{margin} and f_{margin} respectively, which provide bit-wise exact *QoO* for a specific workload, architecture and chip part as $maxV_{margin}$ and $maxf_{margin}$.

$$maxV_{margin} = \max_{0 \leq V_u \leq V_n} V_n - V_u \mid QoO(Exec(W(V_u, f_n))) = BE \tag{4.6}$$

$$maxf_{margin} = \max_{f_n \leq f_o \leq \inf} f_o - f_n \mid QoO(Exec(W(V_n, f_o))) = BE \tag{4.7}$$

4.2 Contributions

In this chapter we show the feasibility of executing applications within the *safe* region without reducing the reliability or the quality of output (QoO) of the executing applications. To be more precise we make the following contributions.

- Motivated by the extend of the $maxV_{margin}$ of two Haswell and four Skylake processors, we present a modeling methodology that takes as input selected CPU performance counters and core utilization, and estimates the voltage margin of the workload on the specific CPU This estimation can be exploited to safely undervolt CPUs and achieve significant energy gains.

- Our model is used by a dynamic voltage scaling governor, called xDVS[2], and extensively verify our model via a long-running (consecutive 72 hours). The system dynamically changed the application workload on 6 different workstations, without any degradation of system reliability.
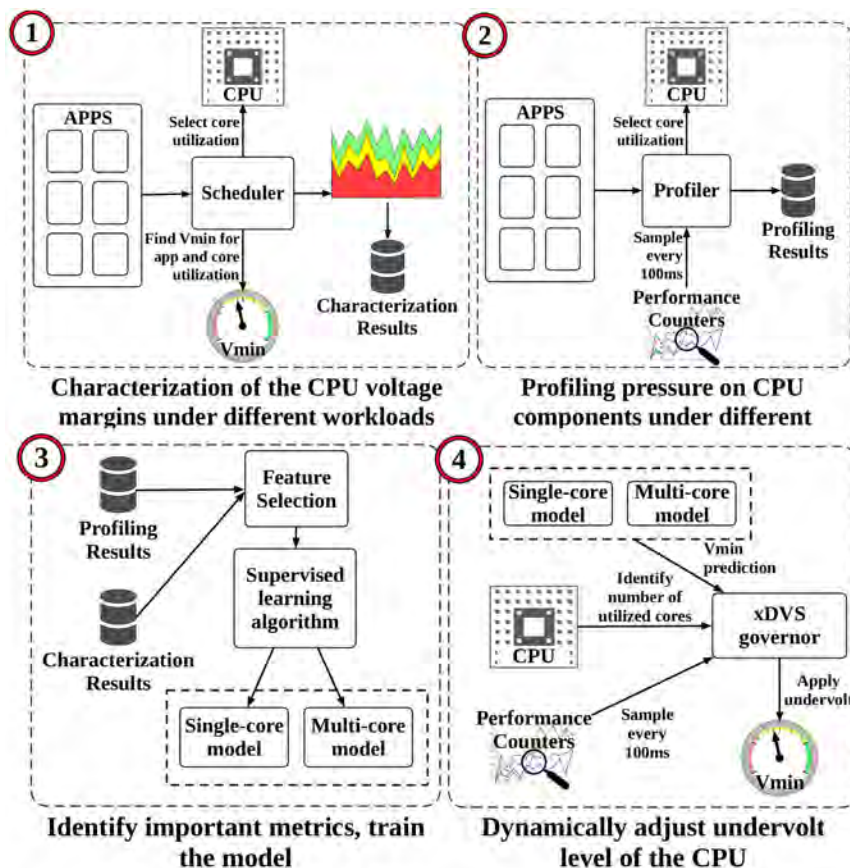
FIGURE 4.1: Overview of our approach for margin characterization, modeling and dynamic prediction.

---

[2]The xDVS governor was implemented by Panos Koutsovasilis

4.3 Methodology

Figure 4.1 illustrates the steps of our approach:

1. The $maxV_{margin}$ is determined for different cores (and all cores together) of the target CPU part for different workloads via offline characterization;

2. The same workloads are profiled to quantify their interaction with the CPU using online performance counters;

3. We combine the $maxV_{margin}$ characterization and the profiling data to fit two models for each CPU part, one for single core execution and one fore multi core execution, that predict a reduced, yet still safe supply voltage V'_{dd} ($V_{min} < V'_{dd} < V_{dd}$).

4. The models are used by a dynamic voltage scaling governor to adjust at execution time the supply voltage of the CPU below nominal values.

4.4 Offline Characterization Background

In this section we describe the offline characterization steps 1,2 presented in figure 4.1.

4.4.1 Methodology to identify $maxV_{margin}$

Figure 4.2 summarizes the methodology we followed to characterize the $maxV_{margin}$. To identify the $maxV_{margin}$ of each of the applications A_i presented in table 4.1 we execute each benchmark in different core configurations (C_j): either occupying a single core, or all cores of the target CPU. Single-core experiments are executed once per core, with the running thread pinned on the respective core while the rest of the cores are idle. To fully utilize the CPU, multi-threaded benchmarks are executed with a degree of parallelism equal to the number of cores, whereas in the case of sequential (single-process) benchmarks we achieve full utilization by co-executing
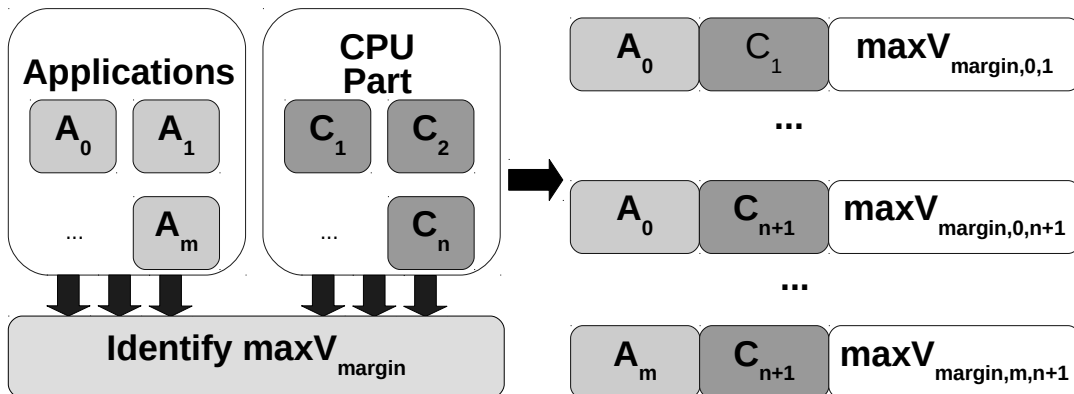
FIGURE 4.2: Offline characterization of $maxV_{margin}$ methodology

Name	Domain	Name	Domain	Benchmark Suite
Bodytrack	Computer Vision	**Facecim**	Animation	
Blackscholes	Financial Analysis	**Swaptions**	Financial Analysis	Parsec [9]
Freqmine	Data mining	**Fluidanimate**	Fluid Dynamics	
Bwaves	Fluid Dynamics	**Leslie3D**	Fluid Dynamics	
Lbm	Fluid Dynamics	**Bzip2**	Compression	
H264ref	Video Compression	**DealII**	Finite Analysis	
Gcc	C Compiler	**Gamess**	Quantum Chemistry	
Gobmk	Artificial Intelligence	**Gromacs**	Molecular Dynamics	
Hmmer	Search Gene Sequence	**Libquantum**	Quantum Computing	
Milc	Chromo-dynamics	**Mcf**	Combinatorial Optimization	SPEC2006 [40]
Namd	Molecular Dynamics	**Omnetpp**	Event Simulation	
Perlbench	Programming Language	**Povray**	Ray-tracing	
Sjeng	Artificial Intelligence	**Soplex**	Linear Programming, Optimization	
Sphinx3	Speech recognition	**Tonto**	Quantum Chemistry	
Xalanbmk	XML Processing	**Zeusmp**	Physics / CFD	
Prime	Mersenne prime stress test			GIMPS [145]
Linpack	HPC			Linpack [49]
Firestarter	Processor stress test			Firestarter [37]
Stress-NG	Linux Stress Test			Stress-NG [132]

TABLE 4.1: Benchmarks used to characterize the voltage margins of the CPUs.

as many copies of the benchmark as the number of cores. We wait initially for all processes/threads to perform their initialization routines (performing I/O etc.) and then we apply the instructed V_{margin}. In the end, we experiment with $n + 1$ application mappings to core configurations, where n is equal to the number of cores within each CPU. The first n configurations are single core executions, and in the case of the $n + 1$ core configuration we use all the available cores. In the end, for each combination of application (A_i), chip part and core configuration (C_j) we have a single $maxV_{margin,i,j}$.

We use XM² presented in section 5.3 to identify the $maxV_{margin}$. For each benchmark and core configuration we determine $maxV_{margin}$ using a binary search algorithm, which looks for the maximum applicable $maxV_{margin}$ within a range $[low, high]$. Initially, $low = 0mV$ and $high = 500mV$. In the first step of the search we set V_{margin} equal to the middle of this interval. During execution we monitor the system for Machine Check Exceptions (MCEs), application crashes, kernel panics etc. When an experiment terminates, the output is compared against the correct, "golden" output in order to detect any Silent Data Corruptions (SDC). To account for potentially non-deterministic behavior, we experiment with every configuration and V_{margin} combination 10 times. If all experiments complete successfully, the region $[low, V_{margin}]$ is marked as safe, the low bound is increased to $low = V_{margin}$, and V_{margin} is adjusted accordingly. If problematic behavior is detected during any experiment, the region $[V_{margin}, high]$ is marked as unsafe, and the high bound is decreased to $high = V_{margin}$. The algorithm terminates when the interval width

becomes less than $5mV$. At this point $maxV_{margin} = V_{margin}$.

4.4.2 Results of offline $\mathbf{maxV_{margin}}$ Characterization

We perform the experimental analysis on 6 workstations, two of them featuring an Intel Haswell i7 CPU, called *Haswell 1, Haswell 2*, and four featuring an Intel Skylake Xeon CPU called *Skylake 1 - 4*. In all experiments, we set the operating frequency to the maximum nominal frequency of the respective CPU. We also disable Intel Turbo Boost technology and the Intel P-state DVFS governor. All workstations run Ubuntu 16.04LTS with Linux Kernel version 4.10.0-38-generic. Table 4.2 outlines the characteristics of each workstation as well as the maximum nominal supply voltage for both architectures under maximum utilization.

Figure 4.3 illustrates the experimentally identified $maxV_{margin}$ for the four Skylake and two Haswell CPUs, running 34 benchmarks. Since the SPEC2006 benchmarks are single-threaded, the respective margins for the fully utilized CPUs are determined by executing simultaneously four instances of the benchmark on each 4-core CPU. The $maxV_{margin}$ spans from 17% to 24% and from 9% to 13% of the nominal V_n for the Skylake and Haswell microarchitectures. The difference between min and max $maxV_{margin}$ values (7% and 4% of the V_n for Skylake and Haswell, respectively) is the workload dependent margin.

Notably, unlike previous studies on ARMv8 [95] and Itanium [4] CPUs that revealed intermediate voltage regions of unsafe operation where indications of erratic behavior may be observed, for the architectures investigated in this study the transition to unreliable operation when the voltage reduction is larger than $maxV_{margin}$ is abrupt and always leads to crashes. Even in the few cases where *SDCs* or *MCE* errors were observed, these errors were accompanied by an immediate system or application crash.

We also observe margin variations across different cores of the same part (difference between margins of the strongest and weakest core of each CPU). The $maxV_{margin}$ variation when executing the same single-threaded benchmark with different cores can reach up to $45mV$ and $32mV$ for Skylake and Haswell, respectively. Also, in contrast to the findings of the characterization of ARMv8 and Itanium, the CPUs

Parameters	Skylake Workstation	Haswell Workstation
CPU	Xeon E3–1220 v5	Core i7–4790
Technology	14nm	22nm
# of Cores	4	4
CPU Base Freq.	3.00 GHz	3.60 GHz
CPU Max Turbo Freq.	3.50 GHz	4.00 GHz
Supply Voltage (V_{dd})	1.15V	1.07V
L1 D-Cache	128KB	32KB
L1 I-Cache	128KB	32KB
L2 Cache	1MB	256KB
CPU LLC Cache	8 MB	8 MB
CPU TDP	80 W	84 W
RAM Size	8 GB	16 GB

TABLE 4.2: Characteristics of the workstations.
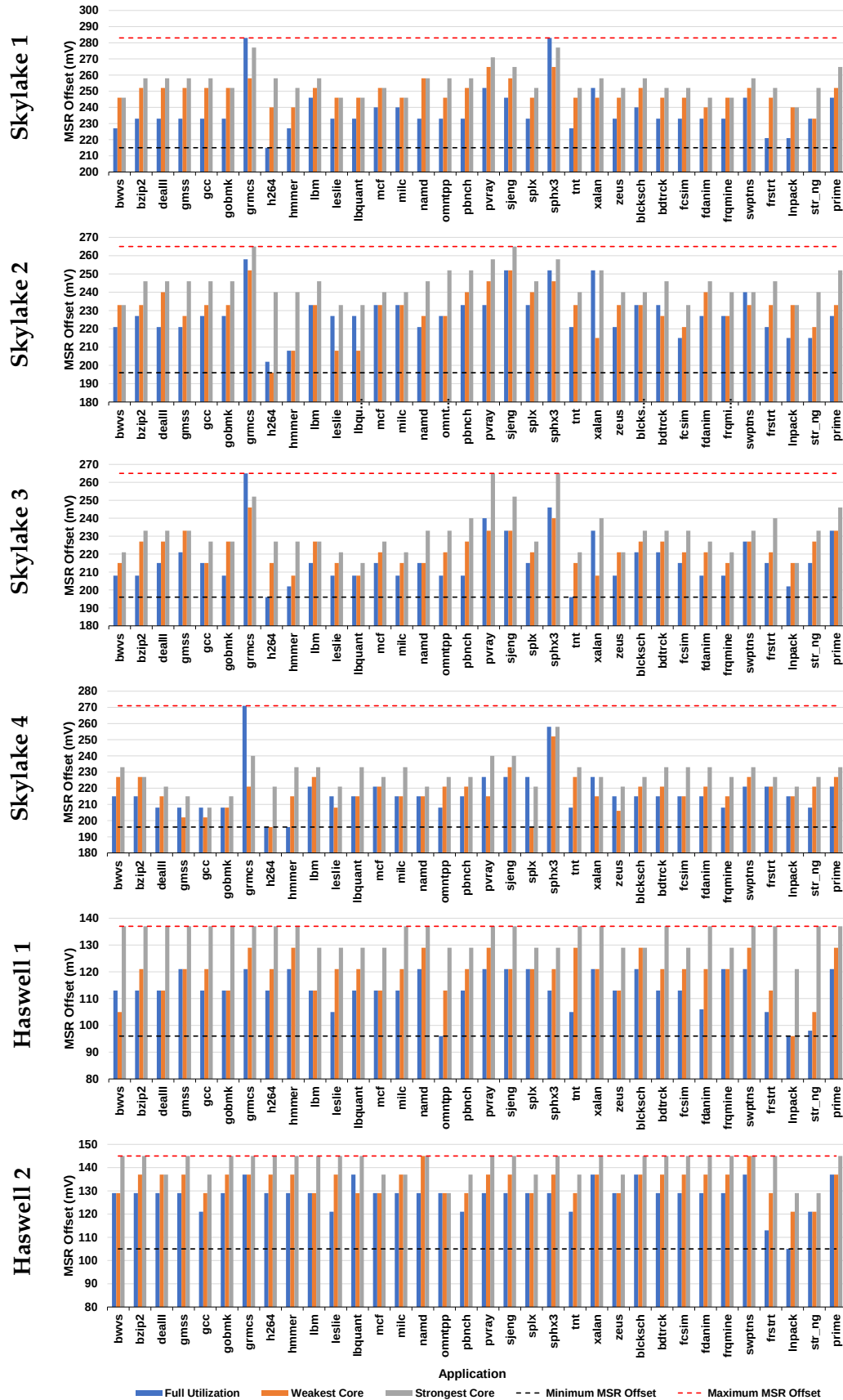
FIGURE 4.3: Evaluation of $maxMargin$ settings for 34 benchmarks (10 runs each) in each workstation; the higher the bar, the wider the exploitable voltage margin. The horizontal dotted lines show the maximum (red) and minimum (black) values of $maxMargin$.
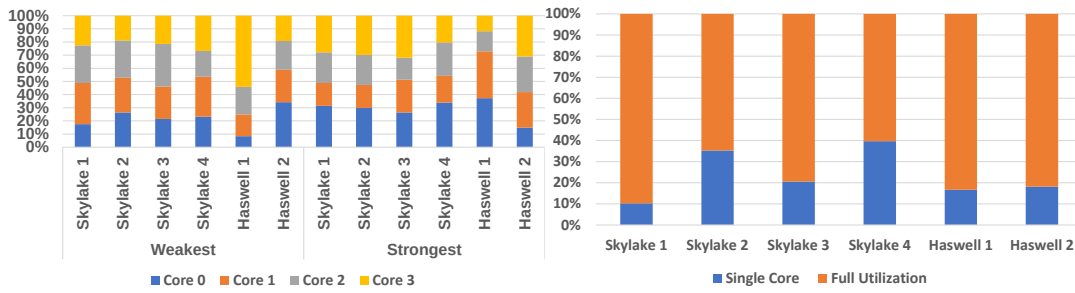
FIGURE 4.4: In the left we present the percentage of the total single core experiment in which the respective core was ranked as weakest or strongest. On the right we present th percentage of the total experiments in which the core configuration was considered as the weakest.

in this study do not exhibit the pattern of a consistently weakest and a consistently strongest core. Figure 4.4(left) shows for each chip the percentage of single-core experiments in which each core was classified as strongest or weakest. In our case the strongest/weakest core varies across chips and even on the same chip across benchmarks.

In most experiments, the exploitable margins on fully utilized CPUs for both families is narrower than the corresponding ones when a single core is utilized (with a few exceptions such as *gromacs* on the Skylake architecture). In Figure 4.4(right) we compare the most conservative single core $maxV_{margin}$ with the respective full utilization $maxV_{margin}$. Typically the full utilization configuration demonstrates narrower margins. This observation strengthens our approuach to execute, single-core workloads as well as full core utilization ones.

Figure 4.5 shows the cumulative distribution function (CDF) of the average (across all *configurations*) failure probability of each CPU, as a function of the applied V_{margin}. The Skylake family exhibits wider $maxV_{margin}$ than the Haswell family, by $103mV$
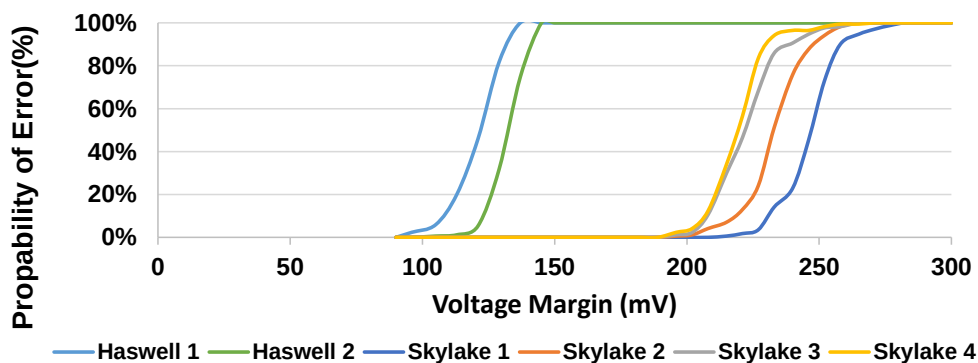
FIGURE 4.5: Average (across all configurations) faulure probability CDF for each CPU, with respect to the applied V_{margin}

on average. Note that lower slope CDF curves indicate a broader range of under-volting opportunities, depending on the characteristics of the workload and the resource pressure exercised. For example, *Skylake 4* offers an $maxV_{margin}$ range between $196mV$ to $271mV$ in which different benchmark configurations will run successfully. On the contrary, *Haswell 2* has a narrower dynamic range (105 to $145mV$) exhibiting a stepwise behavior. All 4 parts of the Skylake family have similar margins, with *Skylake 1* being able to operate at lower V_u values than the rest.

4.4.3 Performance Counter Profiling

Figure 4.6 summarizes the methodology we followed to quantify the utiliazation and pressure of an application-core configuration to the microarchitectural components. This quantification is performed through a set of performance metrics observable through respective Performance Monitoring Unit (PMU).

Each application (A_x), is described by two collections of profiling information, one for single core execution (C_s) and one for the full utilization execution (C_f). Each collection consists of several samples (S_i) and each sample consists of several performance metrics (m_j), also called features. The total number of samples depends on the execution time of the application-core configuration, as we sample the performance metrics every $100ms$ using the Linux *Perf* tools [101]. Finally, the number of performance metrics collected for each sample depends on the architecture. We profile 84 and 79 performance metrics for Skylake and Haswell, which are the ones also used in Intel's Top-down Microarchitecture Analysis Method (TMAM) [83]. Since only up to 8 performance counters per core can be monitored simultaneously, to collect data for all respective counters, we perform multiple executions for each benchmark configuration and in each execution we record a subset of performance counters, until all counters are covered.
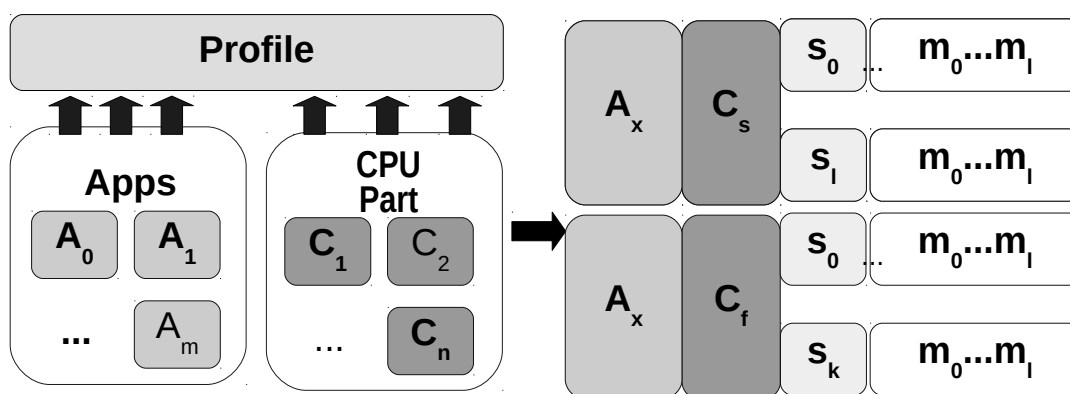
FIGURE 4.6: Profiling performance metrics
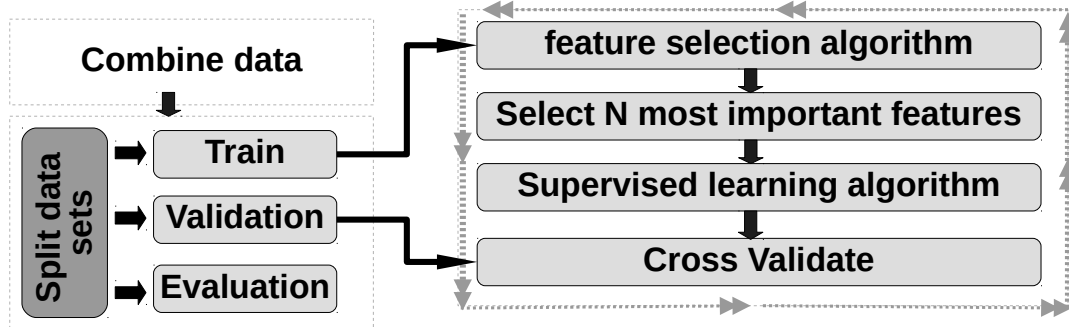
FIGURE 4.7: Methodology Used during the model training phase

4.5 Modeling phase

The modeling phase consists of several steps used in Machine learning approaches presented in Figure 4.7. Initially we need to combine the observations of the offline phases ($maxV_{margin}$ characterization and profiling). Then the applications are splitted into a training, a validation and a testing set. Splitting the applications into sets allows the our methodology to assess how the results of our offline phase will generalize to an independent/unseen application. The training and the validation sets are used during the model training phase. The evaluation is used during the evaluation of the dynamic voltage scaling governor. Finally we perform an iterative methodology to train our prediction model. The iterative produces multiple models and selects 2 models for each chip, one for single core execution and one for multi core executions. Both models are the ones that minimize the RMSE.

4.5.1 Combine Offline Data

The results of the $maxV_{margin}$ characterization (Section 4.4.2) demonstrate that the $maxV_{margin}$ besides being dependent on the workload also depends on the resilience of the specific cores (inter-core variation) and degree of CPU utilization. Moreover, in a realistic scenario, the operating system may, independently of our mechanisms, modify the topology of active cores via thread migration. Our model should provide a safe setting irrespective of workload mapping to cores. To capture these variations, we construct two models for each CPU part.

Single Core Model The model covers the case of single core execution and is trained using the $maxV_{margin}$ of the weakest core for each benchmark. In other words we map the profiling collection C_s for each application (A_i) to the minimum observed $min(maxV_{margin,i,x}) \mid x \in [0, n)$ of single core executions. This design decision misses opportunities for more aggressive undervolting as we always respect the weakest core for each application. In any case though, the penalty is on average $5mV$ and therefore negligible.
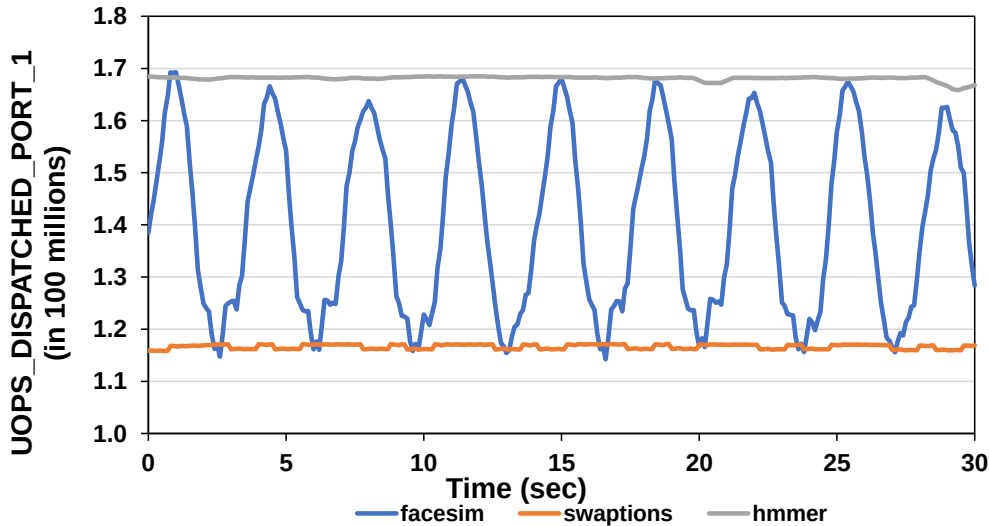
FIGURE 4.8: The number of dispatched uops in port 1 during the execution time of an application.

Full Utilization Model The model covers the case where all cores are occupied (full utilization model). For each application Ai we use the $maxV_{margin,i,n}$ and the profiling collection C_f.

Although we create two models for each part, the methodology used is the same for all parts and all models.

4.5.2   Data Splitting

Most applications exhibit different execution phases in terms of CPU resource pressure and performance characteristics. For example, Figure 4.8 shows the number of uops dispatched for execution in port 1 for the first 30 secs of execution for two applications with a single phase (*swaptions*, *hmmer*) and one application with multiple phases (*facesim*). However, our offline characterization determines the single worst-case $maxV_{margin}$ across all execution phases of an application. This can negatively affect the effectiveness of training our model as it will overgeneralize, trying to correlate wildly varying performance counter patterns with the same $maxV_{margin}$.

To provision for such cases, we bias the training input set to include mainly applications with few execution phases such as *hmmer* and *swaptions*. We first rank the 34 benchmarks according to the number of phases they exhibit, normalized to their execution time. Phase change detection is performed by monitoring large changes (more than 10%) on any of the Level-1 performance metrics of TMAM [83]. The smaller the number of phase changes, the higher the ranking of the application. Then, we select the top 90% (most stable) applications for training and validation. 90% of the selected applications are used for training and 10% for validation. The remaining 10% of applications (the ones with the largest number of phase changes) serve as the testing (evaluation) set. The validation set includes *bodytrack, freqmine, gcc*, and the testing set includes *facesim, zeusmp, fluidanimate, stress_ng*.

4.5.3 Model fitting

To build an accurate model you need to identify several parameters listed below:

i *Number of features* How many features the model should use.

ii *Feature Selection Algorithm* Which feature selection algorithm to use since different algorithms rank features in a different order.

iii *Model Type* What kind of model should you use, for example linear models decision trees, random forest etc.

iv *Hyper-parameters* What kind of hyper-parameters to use for this model. In Machine Learning (ML) techniques, a hyper-parameter is a parameter whose value is set before the learning process begins. Different model types require different hyper-parameters, for example, simple algorithms such as ordinary least squares regression require none, whereas *Ridge*[42] requires a regularization hyper-parameter.

We performed an exhaustive search over this four dimensional space to discover the combination which minimizes the Root Mean Square Error (RMSE) of the validation set. Due to the limitations of Intel PMU, at execution time we are limited to concurrently measuring up to 8 PMU events per core, hence we limit the search algorithm to test up to 8 features. We use all the feature selection algorithms provided by the Scikit-learn [100] ML library. Moreover, we tested several different classification and regression supervised learning algorithms such as linear regression, nearest neighbour, Support Vector Machines (SVM), desision trees, and ensemble methods. Finally, depending on the supervised learning algorithm we searched for different hyper-parameters.

Below we present the modeling parameters which reduce the RMSE as identified by the exhaustive search.

Feature Number

To identify the optimal number of features we start from one feature and we test up to eight features, as the PMUs of our architectures can measure up to eight performance counters simultaneously. In the end, the optimal number of features were estimated to be the maximum (8).

Feature Selection Algorithm

The optimal feature selection algorithm was mutual information (MI) which is an algorithm commonly used for feature selection in machine learning [69]. It ranks different features by assigning weights so that the higher the weight the more important the feature for modeling. The algorithm assigned the highest importance to the metrics listed in Table 4.3 in decreasing order of importance. Note that the highest ranked metrics essentially characterize the instruction mix of the workload.

Skylake	Haswell
UOPS_DISPATCHED.PORT_0:	*IDQ.ALL_DSB_CYCLES_4_UOPS:*
Uops dispatched for execution in port 0 (Port 0 is responsible for Int., FP, vector ALU, mult, div and branch operations).	Cycles in which Decode Stream Buffer (DSB) is delivering 4 Uops.
UOPS_DISPATCHED.PORT_4	*UOPS_EXECUTED.PORT_0:*
Uops dispatched for execution in port 4 (Port 4 is responsible for Store operations)	Uops dispatched for execution in port 0 (Port 0 is responsible for Int., FP,vector ALU, mult, div and branch operations).
UOPS_DISPATCHED.PORT_1:	*UOPS_EXECUTED.PORT_1:*
Uops dispatched for execution in port 1 (Port 1 is responsible for Int., FP and vector ALU operations).	Uops dispatched for execution in port 1 (Port 1 is responsible for Int., FP and vector ALU operations).
UOPS_DISPATCHED.PORT_5	*UOPS_EXECUTED.PORT_5:*
Uops dispatched for execution in port 5 (Port 5 is responsible for Int. and vector ALU operations).	Uops dispatched for execution in port 5 (Port 5 is responsible for Int., vector ALU operations).
UOPS_DISPATCHED.PORT_2	*UOPS_EXECUTED.PORT_2:*
Uops dispatched for execution in port 2 (Port 2 is responsible for Load operations).	Uops dispatched for execution in port 2 (Port 2 is responsible for Load operations).
EXE_ACTIVITY.PORTS	*UOPS_EXECUTED.PORT_6*
Cycles for which one uop began execution on any port, and the Reservation Station was not empty.	Uops dispatched for execution in port 6 (Port 6 is responsible for Int., and branch operations).
UOPS_EXECUTED.THREAD	*UOPS_EXECUTED.PORT_3*
Number of Uops executed by this hardware thread.	Uops dispatched for execution in port 3 (Port 3 is responsible for Load operations.)
MEM_UOPS.ALL_STORES:	*MEM_UOPS.ALL_STORES:*
Number of store operations retired.	Number of store operations retired.

TABLE 4.3: Most influential performance metrics for V_{min}, as ranked by the MI algorithm.

During the offline profiling phase, the performance metrics are collected through multiple executions for the same configuration of each experiment. After the model training procedure identifies all the parameters for the optimal model, we repeat the experiments so that the selected metrics (Table 4.3) are collected during the same execution and we retrain the model. These data are normalized to take values between 0 and 1. As a normalizer, we use the sum of all available slots during the sampling period, which is equal to the number of pipeline slots (4 in our architectures) multiplied by the number of clock cycles (the respective counter does not fall into the limitation of 8 PMU events per core).

Supervised Learning Algorithm

The values obtained by offline characterization do not always exhibit a simple, monotonic behavior with respect to the obtained performance metrics. Consequently, linear regression models do not adequately capture the $maxV_{margin}$ of the applications. Instead, to predict $maxV_{margin}$ as a combination of the aforementioned metrics, we employ a machine learning ensemble technique, called Random Forest Regression (RFR) [78]. A random forest is a collection of regression decision trees, each used to independently predict a value based on an input vector. The model predicts by averaging over the predictions of all regressions trees.

Hyper-parameter selection

Typically, RFR is defined by a predefined number of different simple estimators (the decision trees) and by the maximum depth of each decision tree. Using a large number of estimators and/or using deep trees for prediction can both incur high performance penalties, and result to overfitting. We detect overfitting by cross-validating the models. In the end, the models that minimized the RMSE for both the training and validation set consist of only three estimators, with the maximum depth of each estimator being equal to three.

In the end our model resulted to an average RMSE of $7.01mV$ and $5.45mV$ for the Skylake and Haswell families, respectively. Generally, there is no single optimal model-parameter, but rather the combination of the parameters that minimize the RMSE. For example, when using 8 performance counter, with the MI feature selection algorithm, decision trees with a maximum depth equal to 8 resulted to overfitting.

4.5.4 Safety Margin

Over- or under-prediction is a common side-effect of many modeling approaches. In our case, over-predicting the $maxV_{margin}$ would result in reducing the supply voltage below V_{min} leading to unreliable operation. Therefore, as a last step, we introduce a small *safety margin* to the estimated $maxV_{margin}$ value. For each individual model, the safety margin is set equal to the RMSE between the value that is predicted for the validation data, and the $maxV_{margin}$ value that was observed during the offline characterization for the respective applications in the validation set. Equation 4.8 provides the final offset that is applied on the MSR registers of the CPU (where $\vec{X}$ denotes the input vector to the model).

$$maxV_{margin}(\vec{X}) = maxV_{margin}(\vec{X}) - safetyMargin \qquad (4.8)$$

The safety margin controls the aggressiveness of our methodology. Using very small values would result to aggressive undervolting at the risk of reduced reliability, whereas too large safety margins would merely decrease the energy gains. A
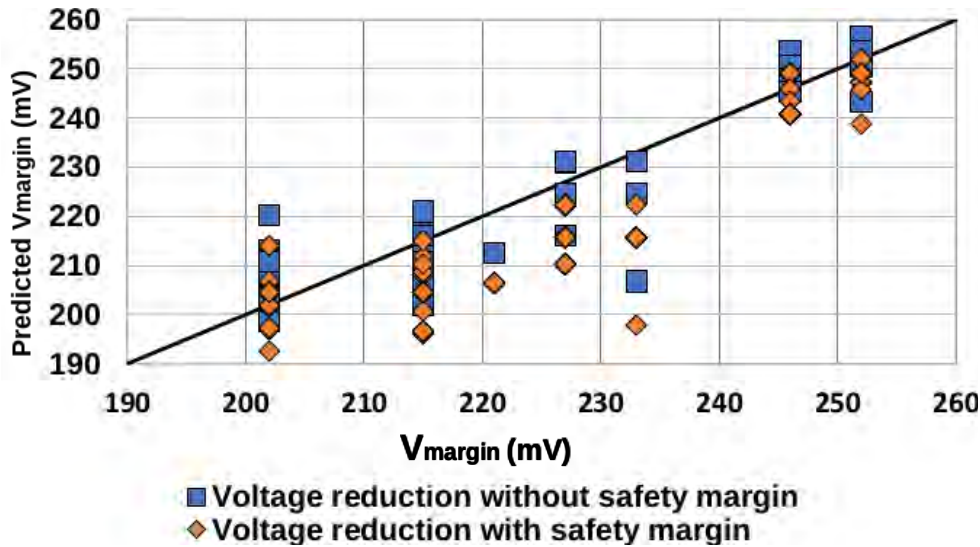
FIGURE 4.9: Prediction of our model with and without the safety margin, for samples in the validation data set.

conservative, yet pessimistic, safety margin is the maximum error between the predicted values and the validation data. Instead, we use RMSE as safety margin (7.01 and 5.45 mV for the Skylake and Haswell families, respectively) and trust the modeling procedure to correctly handle the outliers. Our approach is validated in the evaluation; no failures have been observed during application execution.

Figure 4.9 shows the predictions of our model for benchmarks in the validation set, running on the two Skylake processors, with and without the safety margin. The black line represents the $maxV_{margin}$ values that would be predicted by a perfect model (corresponding to the observed V_{min}). Predictions above the line correspond to application phases that can be executed in a V_{dd} lower than the conservative, application-wide V_{min} which was obtained in the offline characterization. Such cases are discussed in Section 4.6. Including the safety margin reduces power efficiency, but enables safe operation.

4.6 Evaluation

The model introduced in the previous section can be used online, to enable fine-grained undervolting at runtime, according to the resource pressure quantified by performance counters samples and the cores utilized by the current workload. To this end, we use an extended dynamic voltage scaling governor (xDVS), which is invoked periodically (every 100 msec in our experiments). Upon invocation it feeds the model with the performance counter measurements collected during the previous interval, and uses the suggested $maxV_{margin}$ to derive a less-than-nominal but still safe supply voltage V'_{dd}.

In this section we evaluate the ability of the governor to dynamically identify voltage margins based on our prediction model and drive the CPU to a more energy
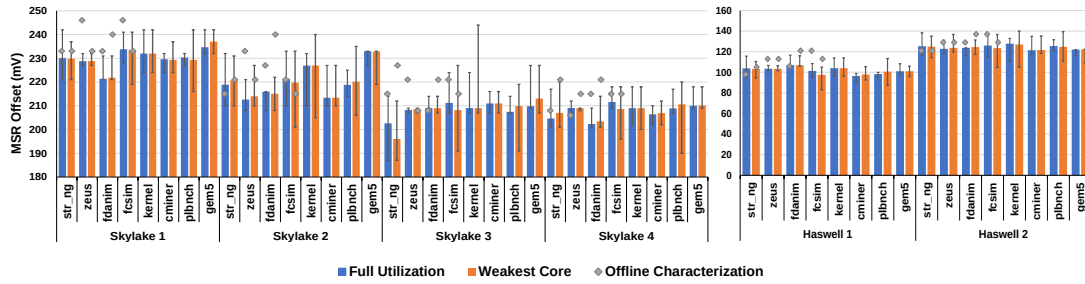
FIGURE 4.10: The bars show the average dynamic $maxV_{margin}$ applied by xDVS for Skylake (left) and Haswell (right) workstations. The min-max bars represent the minimum and the maximum $maxV_{margin}$ applied by xDVS. The gray diamond represents the $maxV_{margin}$ as identified by offline characterization at the granularity of the whole application.
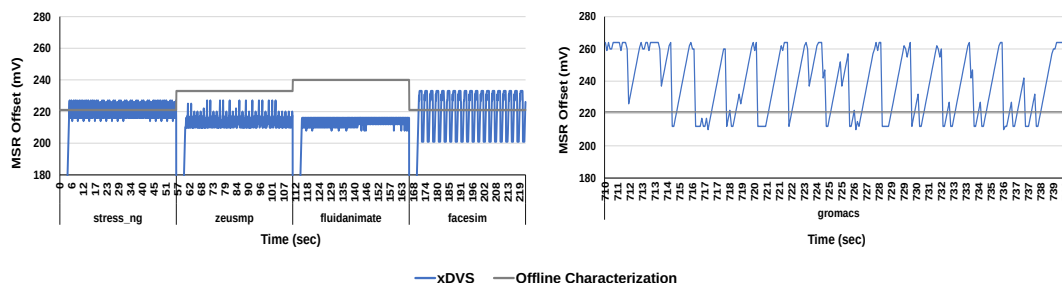
FIGURE 4.11: Timeline showing the applied $maxV_{margin}$ for consecutive single core executions of four applications on Skylake 2 (left), and a snapshot of full system utilization execution for *gromacs* application on Skylake 4 (right).

efficient state. We quantify the resulting energy gains using the benchmarks in the test set (*stress_ng, zeusmp, fluidanimate,facesim*), which have not been used during model training and validation and we compare the energy gains of xDVS against the Intel *P-state* DVFS governor.

In addition, we evaluate xDVS with 4 larger-scale applications, namely as Gem5 [10], a CPU miner [51], the compilation of the Linux Kernel [80] and Polybench [102]. More specifically, *Gem5* simulates an ARM processor using the system emulation mode. During the simulation we execute a variety of simple micro kernels such as Integer and Floating Point Matrix Multiplication, Sorting algorithms, and Combinatoric problem solving kernels. The *CPU miner* employs five different hashing algorithms (*Bitcore, Sha256d, Xevan, Timetravel and Cryptonight* [93]), all used to perform mining for different cryptocurrencies such as Litecoin and Bitcoin. Finally we use multiple solvers and stencils included in the *Polybench* suite such as *Alternating Direction Implicit (ADI), Jacobi, LU factorization, Gram–Schmidt process, Gauss–Seidel*. Each one of these 4 larger-scale applications is executed for approximately 1 hour.

Figure 4.10 shows the average $maxV_{margin}$ applied by xDVS when applications are executed on all cores and evaluates the *full utilization model*, or on the weakest core and it evaluates the *single core model*. For the large-scale applications, in which there is no offline characterization, we present the average dynamic $maxV_{margin}$ across all single core executions.
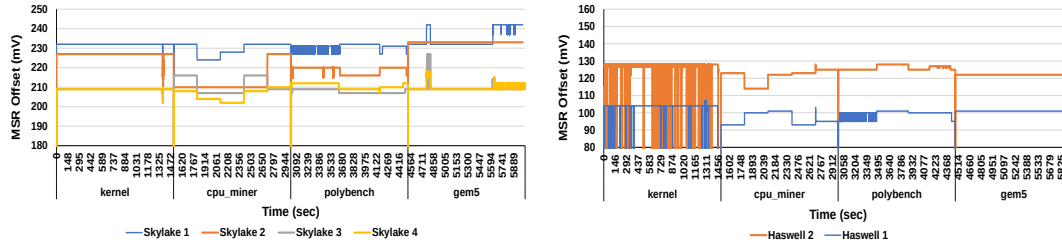
FIGURE 4.12: The timeline showing the applied $maxV_{margin}$ while executing the large applications in full system utilization for Skylake (left) and Haswell (right) workstations.
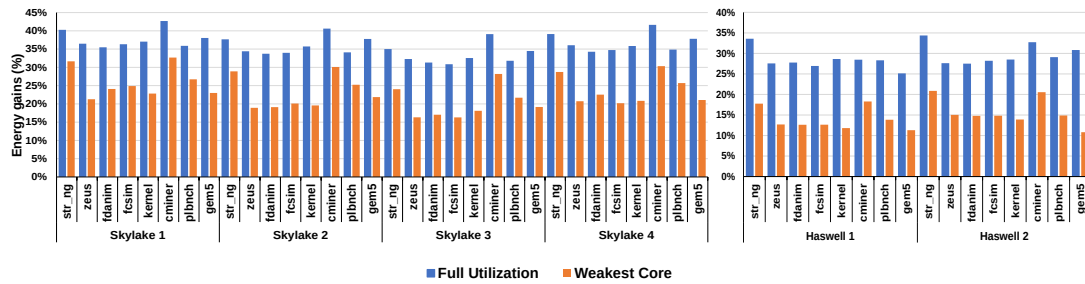
FIGURE 4.13: Energy gains of xDVS when compared with Intel P-state governor for Skylake (left) and Hawell (right) CPUs. The grey horizontal lines represent the $maxV_{margin}$ obtained by the offline characterization.

The $maxV_{margin}$ applied by xDVS includes the extra safety margin, thus we expect it to be on average more pessimistic than the $maxV_{margin}$ identified by offline characterization (as shown in Figure 4.10). The voltage applied by xDVS (including the safety margin) is on average $9.3mV$ and $8.2mV$ higher that the one identified by offline characterization for the Skylake and Haswell microarchitectures respectively.

Despite the safety margin, there are cases in which xDVS successfully identifies application phases and adjusts the supply voltage to lower values than those identified in the offline characterization, without compromising system reliability. Note in Figure 4.10 that xDVS identifies wider static and dynamic (phase-dependent) voltage margins for the Skylake family, which is compatible with the findings of the offline characterization presented in Figure 4.5.

The left side of Figure 4.11 shows the dynamically applied $maxV_{margin}$ when four benchmarks are scheduled for consecutive execution on the same core of Skylake 2. xDVS is able to capture the dynamic nature of the applications. In the *facesim* case (an iterative application that executes 3 separate kernels per iteration) the $maxV_{margin}$ varies between $201mV$ and $233mV$ while the model captures the phases of the application. A similar behavior is observed for *gromacs* (Figure 4.11 on the right), in which xDVS captures the periodic phase changes and drives undervolting even more aggressively than what static $maxV_{margin}$ dictates. For brevity, we do not include graphs of the remaining CPUs as they present similar trends.

Figure 4.12 shows the $maxV_{margin}$ timeline when the four larger-scale applications are scheduled for execution on each workstation. The graphs reveal relatively

large intra-family margin variations, as well as variations due to different workload characteristics. Note that the xDVS governor is able to capture the different algorithms consecutively used by the CPU miner application. Although the variation between different predictions is small, as observed in the offline characterization, the margin between correct execution and failure is small (less than $5mV$). Therefore, even minimal adjustments to the supply voltage can have an noticeable impact on system reliability.

Figure 4.13 shows that the xDVS governor achieves 29.59% and 21.93% average CPU energy gains and can reach up to 42.68% and 34.37% for the Skylake and Haswell processors respectively, compared with the Intel *P-state* governor. Higher gains are – as expected – obtained when cores of the CPU are highly utilized. For the energy consumption extraction we used Linux *Perf* tool.

4.6.1 Mixed Workload Long Run Evaluation

To stress the reliable execution of xDVS as well as the ability of the model to predict safe margins for mixed workloads we orchestrated a custom long run execution. We created a workload pool comprising the testing set benchmarks, the four larger applications as well as idle tasks. We randomly select a benchmark from this pool and randomly define the level of parallelism (either in the form of number of threads or instances). We keep spawning benchmarks this way until all cores are assigned one benchmark. Every time an application terminates another application is spawned. We left this experiment running for 72 hours. All workstations successfully terminated their executions. Our xDVS governor performed on average $2.6 * 10^6$ voltage regulations per workstation during that period.

Note that when xDVS is enabled, Intel's Turbo Boost technology is disabled and the operating frequency is pinned to the CPU maximum nominal frequency. This penalty, due to lower operating CPU frequency, translates to application execution slowdown and is on average 8.73% and 5.59% for the Skylake and Haswell, respectively. However, the actual performance overhead of the xDVS governor itself is minimal. When the governor is active and performs estimations every $100ms$ the prediction requires only $160ns$, while changing the new $maxV_{margin}$ requires $155us$. On average, the performance overhead is equal to merely 0.04% of execution time.

4.7 Voltage Emergencies

Our technique is designed to predict V_{min} voltage oscillations when executing conventional workloads in a multi-core CPU. However, aggressive voltage scaling comes at the cost of increased risk of CPU malfunctioning in pathological cases, when, for example a voltage droop virus is injected (potentially maliciously) for execution. Such viruses (also called stressmarks) are artificially generated to produce large voltage fluctuations and expose the susceptibilities of the power delivery network of

microprocessors [133, 66, 65]. Typically, stressmarks consist of a periodic sequence of high activity and low activity instruction regions at a frequency equal to the resonance frequency of the power delivery network ($50 - 200MHz$ in modern CPUs). This pattern simulates the invocation of high CPU activity workloads immediately after a period of very low activity, which causes large di/dt swings and large voltage droops. Such viruses can affect the reliable operation of CPUs, even when power supply is at nominal voltage levels.

Our methodology depends on the average behavior of the current workload as this is quantified by the monitored performance counters. The experimental evaluation demonstrates that for typical workloads the performance counters can be used as indicators to aggressively reduce the supply voltage. However, the sampling of the performance counters and the supply voltage adjustment is performed every 100 ms, which is too coarse to capture events caused by viruses which appear at the granularity of tens of nsecs. The problem of voltage droop detection and mitigation in modern CPU and GPUs is (and should be) addressed at the hardware level with specialized circuitry [127, 116]. A high speed droop detection mechanism continuously monitors the power grid causing a rapid charge shunt to the V_{dd} rail to correct a voltage emergency within a few clock cycles.

Our scheme makes the CPU more vulnerable to voltage emergencies and would probably require stricter thresholds and faster response times from such hardware mechanisms to counteract the effects of voltage droop viruses. Ideally, these mechanisms could also inform the software stack when voltage emergencies are detected, acting as a trigger towards more conservative undervolting. In general, stronger error protection and error recovery mechanisms to correct timing violations are very helpful for extra protection in aggressive voltage scaling. Note also that voltage droop viruses are difficult to generate and may be different across not only different microarchitectures but also across different CPU parts. Moreover, on top of a realistic software stack, the background operating system activity (jitter) smooths out large voltage swings caused by such viruses [106].

Chapter 5

Experimental Frameworks for Reliability Analysis

This dissertation seeks energy reduction by reducing the correctness requirements of the computer systems. Relaxing the hardware correctness requirements reduces the reliability of the system. A major challenge of this thesis was to understand under which circumstance do errors manifest on the hardware and how different applications mask the respective errors. To be more precise we faced the following challenges:

- Understand how and under what circumstances modern CPU microarchitectures fail when executing code at reduced margins

- Identify the voltage margins of the system and associate them with their respective energy gains.

- Evaluate the resilience of individual applications and the complete software stack.

5.1 Contributions

To overcome these challenges we developed two fault injection frameworks which where used by this thesis to conduct the experimentation. Below we present the contributions of this chapter:

GemFI A simulation based fault injection tool, the tool extends the popular Gem5 simulator. The primary objective of the tool is to enable fault injection based on different fault models and on systems with various configurations. A variety of different system configurations and architectures can be supported without affecting the implementation of fault injection in GemFI

XM²: A framework that monitors and manages the operation of systems when their CPU is either overclocked or undervolted. It can be effectively used to identify the margins of a system, or study the effect of real faults on the application and associate the undervolting/overclocking with the respective energy gains.

- We use **XM²** to analyze the effects of the source code and compiler optimizations to the frequency margins of the ARM Cortex A53 processor.

In the remainder of the chapter we describe and compare these two frameworks and the analysis on the ARM Cortex A53 processor.

5.2 GemFI:Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates

5.2.1 The Gem5 Simulator

Gem5 is a popular open-source system simulator. It provides a modular platform for computer system-level architecture research, encompassing system-level architecture as well as processor micro-architecture.

Object oriented design enhances the flexibility of Gem5. The ability to construct configurations from independent objects facilitates multicore and multi-system design. Moreover, Gem5 provides four different CPU models, each of them representing a different point in the speed vs simulation accuracy trade-off. *Atomic Simple* is a single IPC CPU model. *Timing Simple* is similar but also simulates the timing of memory references. *InOrder* is a pipelined in order CPU. Finally, *O3* is a pipelined out-of-order CPU model. Gem5 also supports two memory system models: *classic* and *ruby*. The classic is fast and easily configurable, while the ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherence memory systems.

Gem5 operates in two modes: *System Call Emulation (SE)* and *Full System (FS)*. In SE mode applications execute on simulated "bare metal". Whenever the program executes a system call, Gem5 traps and emulates the call usually by passing it to the host OS. Currently there is no thread scheduler in SE mode. Therefore, threads are statically mapped to a core, hindering its use with multi-threaded applications. FS mode offers an environment for running an operating system (OS) on top of the simulator. There is support for interrupts, exceptions and I/O devices. Applications are executed under the control of the OS.

Gem5 supports a number of ISAs, including Alpha, MIPS, ARM, Power, SPARC and x86. The simulator's modularity allows these different ISAs to be easily implemented on top of the generic CPU models and the memory system.

5.2.2 GemFI Design and Implementation

We extended Gem5 with fault injection capabilities, following the General Processor fault model described in [147]. The result, GemFI, is a configurable tool for studying the effect of faults in a processor.

GemFI was developed using C++ and Python. It fully supports the Alpha and Intel x86 ISAs. Supporting more instruction sets is rather straightforward, since the
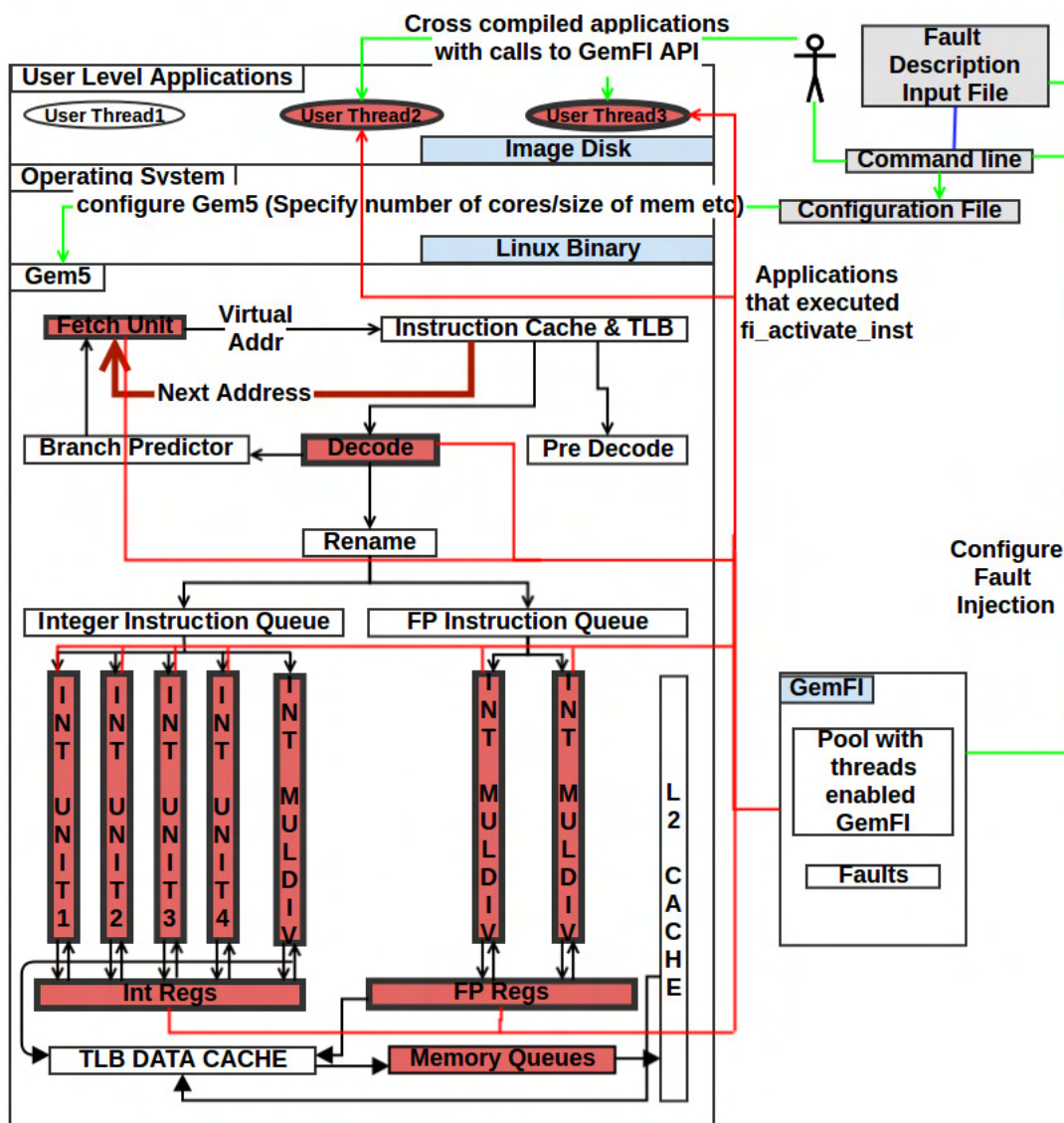
FIGURE 5.1: An architectural overview of GemFI. The red components of the architecture demonstrate the possible locations where faults can be injected, whereas the red ovals represent applications which use the extended ISA.

implementation of GemFI is fairly ISA-agnostic. GemFI supports full system simulation mode as well as the execution of multi-threaded applications. An architectural overview of GemFI is depicted in Fig. 5.1, whereas the following sections discuss its main features in more detail.

GemFI User Interface

GemFI provides an API consisted of two intrinsic functions.

- **void fi_activate_inst(int id)** is translated to a pseudo-assembly instruction. Its successive occurrences toggle (active/inactive) the manifestation of faults for the specific process/thread. The executing thread is assigned a numerical id which can be used as an identifier of the thread in fault injection configuration.

- **void fi_read_init_all()** checkpoints the simulation. Upon restoring from the checkpoint, it resets all the internal information of GemFI, allowing the same checkpoint to be used as a starting point for multiple experiments with potentially different fault injection configurations.

On GemFI invocation the user also provides – at command line – an input file specifying the faults to be injected in the upcoming simulation. Each line of the input file describes the attributes of a single fault. Faults are characterized by four attributes: *Location*, *Thread*, *Time* and *Behavior*.

Location: Fault location specifies the micro-architectural modules to be targeted for fault injection. The user specifies the core, the module within the core and finally the specific bit location to be corrupted. Supported locations include registers (integer, floating point, special purpose), the fetched instruction, the selection of read/write registers during the decoding stage, the result of an instruction at the execution stage, the PC address and finally memory transactions (load/stores).

Thread: The thread attribute allows to selectively inject faults to specific threads, using the id assigned to the thread upon execution of *fi_activate_inst(id)* as an identifier.

Time: Another important aspect of the fault injection configuration is its timing. Timing is relative to a simulation milestone, marked by the execution of the *fi_activate_inst*. Faults are scheduled relatively to the number of instructions already executed, or to the number of elapsed simulation ticks of the targeted thread.

Behavior: The values of the specified faulty location can be corrupted in following ways:

- by assigning an immediate value provided by the user to the location.
- by XORing the running value at this location with a user-specified constant.
- by flipping the running value at bit locations. Multiple bit flips are supported by injecting multiple faults on the same module.
- by setting all bits of the location to a value of 0 or 1.

To emulate the behavior of transient and permanent faults, the user can define how long the fault is active in terms of the number of simulation ticks or number of instructions. For example, a fault injected in the execution stage of the processor can be injected continuously for the next N instructions (or for the next N simulation cycles) if so instructed by the user.

```
1 "RegisterInjectedFault Inst:2457 Flip:21
2     Threadid:0 system.cpu1 occ:1 int 1"
```

LISTING 5.1: A sample input file to GemFI

```
1 #include <m5op.h>
2 int main(int argc, char *argv[]){
3   int id = 0;
4   initialize_input_data();
5   fi_read_init_all();
6   fi_activate_inst(id);
7   foo();
8   fi_activate_inst(id);
9 }
```

LISTING 5.2: Modified source code of an application for fault injection.

5.2.3 Simple Example

Listing 5.1 outlines a user-provided fault configuration example. The fault is injected in the 21st bit of register R_1 of the CPU (location), when the application fetches the 2457th instruction after the initiation of fault injection for this thread (*fi_activate_inst*). The fault is activated for a single instruction (occ:1) and only for the thread with id equal to 0.

The end user compiles (or cross-compiles) the application to be tested (Listing 5.2). Target applications must, at least, contain one call to initialize fault injection. Afterwards, the user moves the binaries into the disk image serving as the virtual disk of GemFI. Using the command line, the user provides a configuration file (Listing 5.1) describing all the faults to be injected in the simulation. After *fi_activate_inst(id)* is called, the thread identifier is stored in the internal data structures of GemFI. Simulation continues normally, until it is time for a fault to be injected. At that time, GemFI alters the state of the target hardware structure according to the fault specification in the configuration file.

5.2.4 GemFI Internals and Implementation

Fig. 5.2 demonstrates the steps executed by GemFI for each simulated instruction.

Threads that have enabled fault injection are internally represented as instances of a class (*ThreadEnabledFault*), containing all per thread information necessary for fault injection, such as the number of instructions the thread has executed on each core. Each simulated core has a pointer to a *ThreadEnabledFault* object. If the thread executing on the core has not activated fault injection, the pointer is NULL. When a thread executes **fi_activate_inst()**, GemFI looks in a hash table to identify whether the specific thread has already activated fault injection. Threads are identified at the hardware/simulator level by their unique Process Control Block (PCB) address. If the thread is not found in the hash table, a new *ThreadEnabledFault* object is created and the running core is set to point to that object. On the other hand, if there was already an entry in the hash table, the invocation of **fi_activate_inst()** deactivates
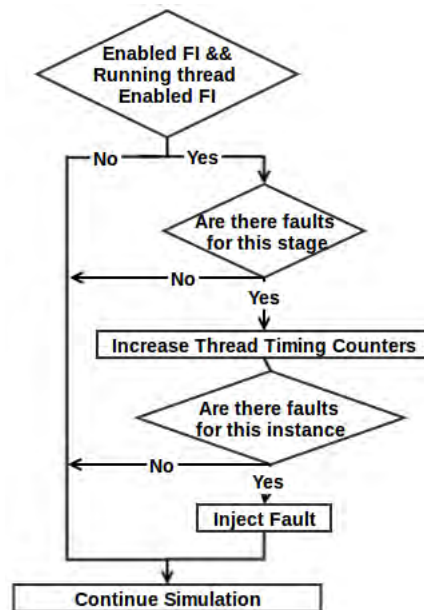
FIGURE 5.2: GemFI functionality on each simulated instruction.

fault injection for the specific thread. The thread is removed from the hash table, the corresponding *ThreadEnabledFault* object is destroyed and the core's pointer is set to NULL. During context switches, which are identified by the change of the PCB address, GemFI checks whether the switched-in thread has activated fault injection, in order to properly set the core's pointer to the thread's *ThreadEnabledFault* object. Monitoring context switches allows GemFI to eliminate the overhead of checking the fault injection status of the executing thread in the hash table on each simulated clock tick.

Faults are described in the input file provided by the user at GemFI command line. The file is parsed at startup and each fault is inserted to one of five internal queues. Each queue corresponds to a different pipeline stage.

On each simulation tick, GemFI checks if fault injection has been enabled for the running thread. In such a case, it prefetches the corresponding *ThreadEnabledFault* objects. Then and for each instruction served at a pipeline stage, GemFI updates the thread's data and scans the corresponding queue for faults targeting the executing thread at the specific simulation point. Queue entries are sorted according to the timing of each fault. If such a fault is found, the value of the targeted location is corrupted according to fault's behavior.

5.2.5 Simulation Checkpointing

Checkpointing allows saving the state of a process or a system at a specific time snapshot and reverting to that later, to restart the execution from that point if needed. Checkpointing is necessary in order to avoid losing simulations in case of unexpected failures. It is particularly useful when simulation campaigns are executed to non-dedicated networks of workstations, a feature supported by GemFI.
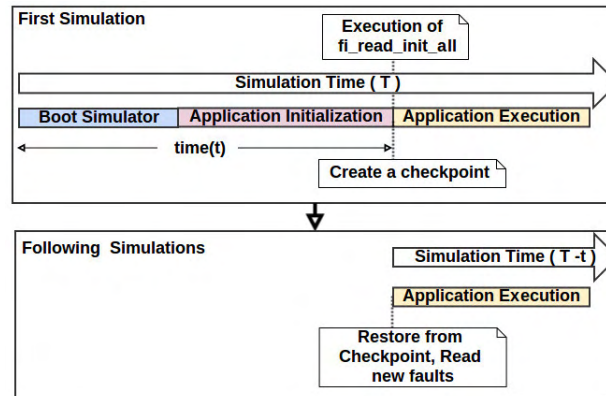
FIGURE 5.3: Simple checkpoint-restore mechanism to speedup simulation campaigns.

Gem5 provides checkpointing, however with limitations. One method is to switch the simulation from O3 to atomic simple mode, create the checkpoint, and revert back to O3 mode to continue the simulation. This requires a pipeline flush, presenting a potential realism loss hazard. The second method requires simulating the MOESI hammer cache coherency protocol, which however dramatically increases simulation time.

We used DMTCP (Distributed MultiThreaded Checkpointing) [2] to checkpoint the state of the Linux process running the simulator, instead of checkpointing the internal state of the simulator. A feature of DMTCP is its ability to take checkpoints either by programmatically invoking checkpointing from within the process to be checkpointed, or asynchronously, by setting environment variables. The ability to invoke DMTCP from within the simulator allows us to exploit the front-end checkpointing mechanism of Gem5, while altering the checkpointing back-end to use the DMTCP API.

Apart from protecting against unexpected problems in simulation campaigns, checkpointing can be used to speed-up simulations. Before starting simulation campaigns, the user executes one simulation up to the point when fault injection is activated (including booting of the operating system and application initialization). Using GemFI's API the user can checkpoint the simulation at this point. The saved state is then used as a starting point for all experiments in the campaign (Fig. 5.3). Upon restoring a checkpoint GemFI parses again the faults configuration file. Therefore, this strategy allows fast-forwarding of the execution to the checkpoint and spawning of multiple experiments, with different fault injection configurations from that point on. As a result, the cumulative execution time of the simulation campaign is significantly reduced, as we demonstrate in Sec. 5.2.8.

5.2.6 Simulation Campaigns on a Network Of Workstations

GemFI is accompanied by a set of shell scripts which facilitate launching simulation campaigns on a network of workstations (NoW). The workstations need to share a network file-system, in order to store the fault description files of the experiments,

the simulation checkpoints and the output of each simulation. The main steps for parallel execution of simulation campaigns on a NoW are the following:

1. The configuration files for all experiments are stored on a network share.

2. A simulation is executed up to the point fault injection is activated and the simulator process is checkpointed. The checkpoint is stored to the share.

3. Each workstation gets a local copy of the checkpoint.

4. Each workstation checks the share for experiments to be executed. It selects one of the remaining experiments and executes it locally, starting from the checkpointed state.

5. Simulation results are moved from the workstation back to the network share.

6. Steps 4-6 are repeated until there are no experiments left.

5.2.7 Validation

In order to validate the functional correctness of GemFI, we conducted an experimental study using a set of benchmark applications. Our simulator system was set to simulate a single core ALPHA CPU coupled with a tournament branch predictor, a L1 instruction cache and a L1 data cache and as a L2 cache we used a unified L2 cache.

DCT, is a kernel of JPEG image compression and decompression [129]. We applied each kernel on a gray-scale 512X512 image. *Jacobi* is applied on a diagonally dominant 64X64 matrix. *Monte Carlo PI* estimates the value of PI by randomly selecting 10^5 points within a unit square and evaluating whether they fall into the inscribed into a circle with radius one. *Knapsack* is a solution of the zero one knapsack combinational problem using a genetic algorithm. We use an input of 24 items and a weight limit of 500. The *Deblocking* filter is a kernel of the AVS video decoding process [30]. We apply it on a $720X240$ pixel image. *Canneal* is a benchmark of the PARSEC Benchmark Suite [9]. *Canneal* employs an annealing (SA) algorithm to minimize the routing cost of a chip design by randomly swapping netlist elements. It was applied on 100 nets, allowing up to 100 swaps in each step.

The number of executions of each application for every experiment varied from 2501 to 2504 and has been calculated using the method presented in [75], setting 99% as a target confidence level and 1% as the error margin.

The execution of each application was simulated both with our tool and the original Gem5 simulator. When simulating using GemFI we did not inject any faults. We then compared the application output from the two experiments, as well as the statistical results provided by the simulator. For all benchmarks the results were identical. This indicates that GemFI does not corrupt the simulation process.
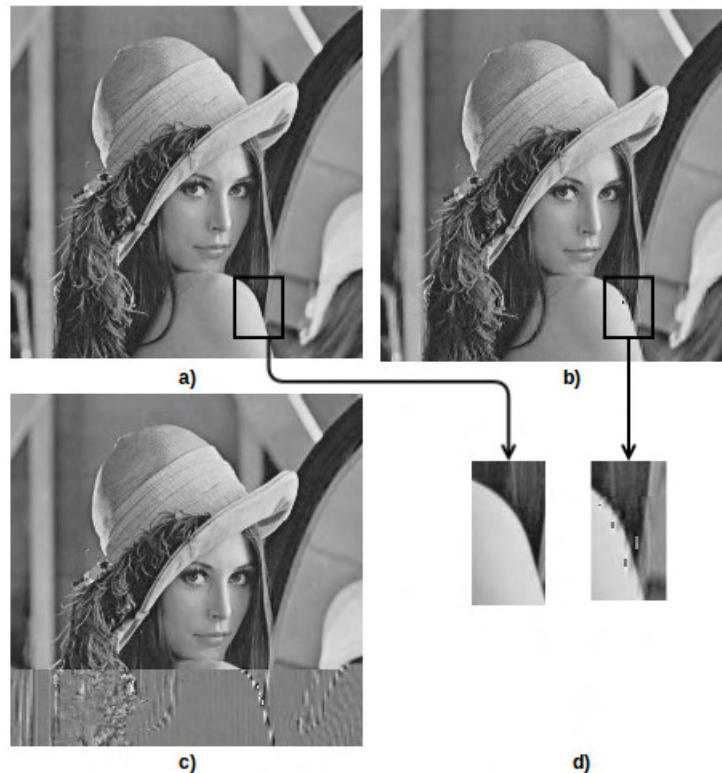
FIGURE 5.4: Different categories of results for the *DCT* benchmark.
a) A strict correct result b) Relaxed correct result c) SDC d) The difference between (a),(b)
(loss of quality)

Validation Methodology

We launched simulation campaigns in which applications are injected with faults. We use a single event upset fault model. Each experiment injects a flip-bit fault, using a uniform distribution for the *Location*, *Time* and *Behavior*. As mentioned earlier, GemFI can support any user-provided realistic fault model.

We initially checkpoint after the system boot-up and the initialization phase of the application under investigation. For each experiment in a campaign, we restore from the checkpoint, start simulating in O3 mode and inject the fault. The simulation continues until the affected instruction commits or squashes (for example, due to a branch misprediction). At that point we switch to atomic simulation and after application termination (normal or crash) we evaluate the quality of the end-result. When injecting a fault we print information on the affected assembly instruction. This information is used *postmortem* to correlate, either analytically or statistically, the fault with the simulation result.

The outcome of each experiment can be classified in the following categories: *crashed*, *non propagated*, *strictly correct* result, *correct* result and *SDC (Silent Data Corruption)*. *Crashed* are experiments which fail to successfully terminate. *Non propagated* are experiments in which faults did not manifest as errors (for example they were inserted in registers, however the corrupted register was either not used during the execution of the application or overwritten before the erroneous value was used).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type
Opcode						Ra					Rb					Unused			0	Function							Rc					Integer Operate
Opcode						Ra					Literal								1	Function							Rc					Integer Operate,Literal
Opcode						Ra					Rb					Function											Rc					Floating Point Operate
Opcode						Ra					Rb					Displacement																Memory Format
Opcode						Ra					Displacement																					Branch Format
Opcode						Function																										CALL_PAL Format

TABLE 5.1: Alpha instruction formats

Strictly correct experiments produce results which are bit-wise identical to those produced by the corresponding error-less execution. *Correct* experiments produce results that are within acceptable quality margins, although not bit-wise identical to those of the error-less execution. The degree of tolerance is application dependent. For *DCT* we compare the produced compressed image with the uncompressed one used as input. Images with PSNR higher than 30 are regarded as correct, since typical PSNR values in lossy image and video compression range between 30 and 50 dB [143]. For the deblocking filter, outputs with PSNR higher than 80 dB, when compared with the error-free execution, are characterized as correct [143]. For *PI* estimation we accept experiments that have computed the first two decimal points correctly, since this the accuracy expected by the error-free execution for the 10^5 test points. Since the tolerance on *Jacobi* is highly dependent on the application domain, we characterize as correct solutions that result to the same (bit-exact) output as the golden model, converging after a potentially different number of iterations. Correct *Canneal* executions are those that reduce the total cost of routing and produce a correct chip. Finally, *SDC*s are executions that terminate normally, yet they produce results outside the acceptable range compared to the results of the error-free execution. Fig. 5.4 depicts an example of the different classes of results.

Experimental Results

Fig. 5.5 depicts the results of the fault injection campaigns, correlating the *Location* of the fault with application behavior. The last column of each chart summarizes the results for the specific application.

All applications demonstrate their highest resiliency to faults targeting floating point registers. Most applications use a small subset of these registers, hence there is a low probability for a fault to affect a live register. Moreover, floating point registers are typically used to store data and not system state information or control flow information. *Deblocking*, a benchmark with no floating point operations, behaves exactly as expected, demonstrating 100% strict correctness.

On the other hand, faults on the integer register file result to higher crash rates. The compiler uses integer registers for storing important information (global pointer, stack pointer, frame pointer, return address register). Moreover compiler uses integer registers for control flow information (loop iterators, base addresses for memory translation). The integrity of these registers is crucial. Integer resisters tend to be live during large spans of the application life. Therefore, any fault affecting them has a
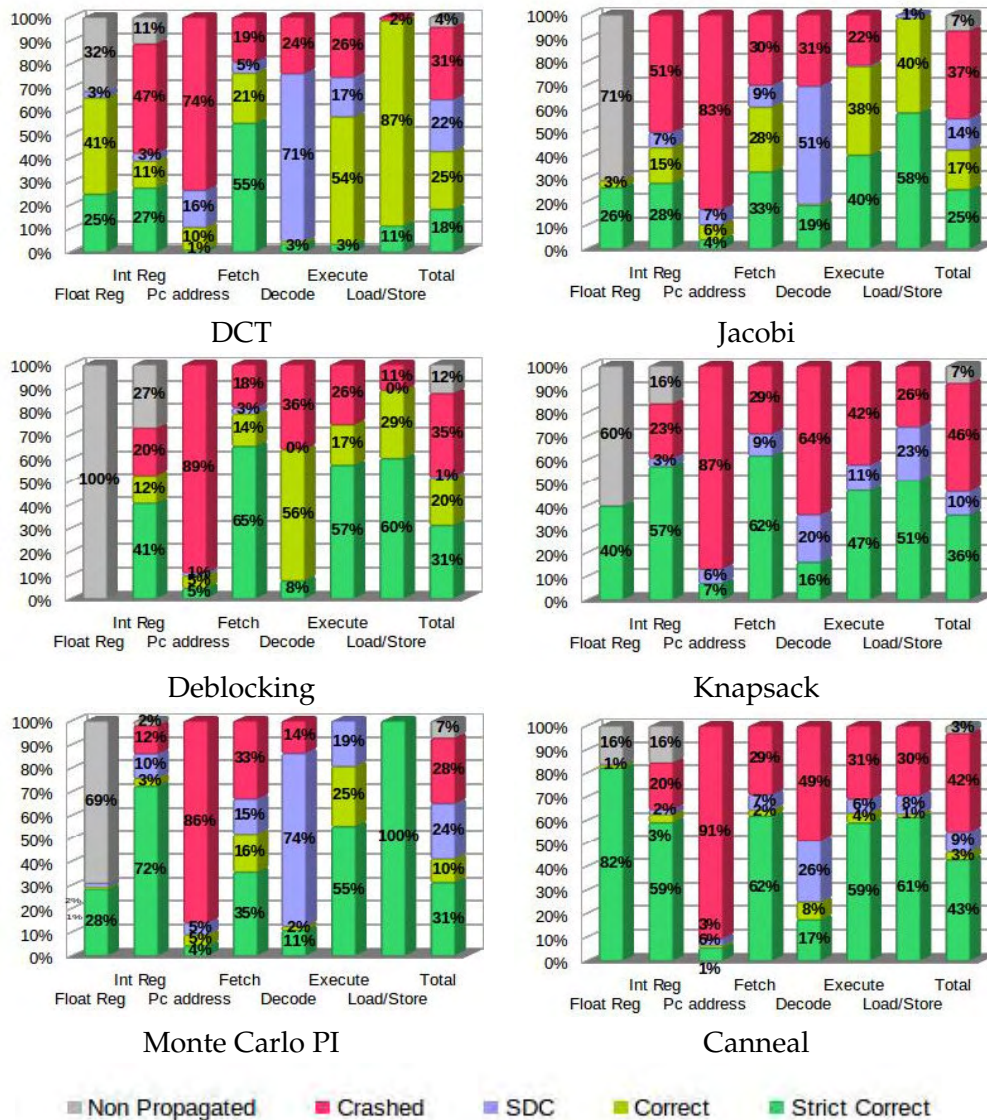
FIGURE 5.5: Application behavior when fault injecting different architectural components.

high probability to cause a crash. For example *DCT* and *Jacobi* which are characterized by many memory accesses and use multi-level loop nests exhibit almost twice the crash rate compared with other applications.

In order to validate fault injection at the fetch stage, we correlated the affected bit location and the instruction type with the end result of the application. The analysis is ISA dependent; Table 5.1 summarizes Alpha instruction format. Experiments affecting unused bits always resulted into strict correct results. Faults affecting branch instructions were validated by checking the simulation statistical information. For example when inserting a fault into the *displacement* bits of the instruction and the branch is not taken the simulation statistics were the same and the end-result was categorized as strict correct. Faults affecting the *Ra* field may cause no error, should the result of the branch remain the same. Whenever faults altered the *displacement*

field of memory instructions the application would crash with a high probability. The same was observed when the error altered the *Ra* value of a memory instruction, since the base address was read by another register. Finally we observed that, exactly as expected, when faults were injected into the *opcode* or the *function* and the resulting opcode/function is not implemented the benchmarks always terminated their execution due to illegal instruction.

A similar analysis was applied for faults inserted in the selection of registers during the decoding stage. Errors which affect the selection of the base of load/store instructions would usually cause a segmentation fault. An interesting observation is that faults inserted in the decoding stage of the *PI* algorithm result to crashes almost at half the probability compared with the remaining applications, because *PI* performs almost no data accesses from memory. Errors in the decoding stage usually lead to SDCs. This is expected, since operations are executed with different inputs. Correct results may be produced only by faults which alter a squashed instruction, or due to inherent, algorithmic application resiliency.

Faults introduced in the execution stage, which alter memory access instructions tended to result to crashes, because at this stage the virtual address of the memory transfer is being calculated. Faults altering the resulting address usually result to segmentation violations. The variation between the percentage of crashes among different applications is consistent with the variation of the percentage of memory operations in the instruction mix. In *Knapsack*, which makes heavy use of arrays and pointers 42% of faults in the execution stage result to crashes. On the other hand, *PI* evaluation, with almost no data accesses from memory, suffers almost no crashes. Correct and strictly correct results when fault injecting in the execution stage were found to be due to faults that have been masked during the remaining execution of the application, or faults that affected the less significant bits of data computations.

Faults altering the result of data loads/stores rarely resulted to crashes, and when they did it was because the error affected a store/load of an address. For example, altering the stored or loaded value of the return address usually led to crash. In total,

Finally faults altering the value of the PC address were almost always fatal for the affected applications. Correct results were obtained in the few cases when the corrupted PC address was close to the correct one. This, in practice, corresponds to a small forward or backward jump.

Another interesting aspect of the experimental validation is the correlation of the timing of fault injection to the effects on the application. Fig. 5.6 depicts the results from three fault injections campaigns with interesting trends. The horizontal axis corresponds to the timing of fault injection normalized to the application execution time and the vertical axis corresponds to the fraction of experiments that resulted to each of the classes of outcomes. *Acceptable* represents the union of correct and strictly correct results.

For *Monte Carlo PI* estimation the time when fault injection took place appears
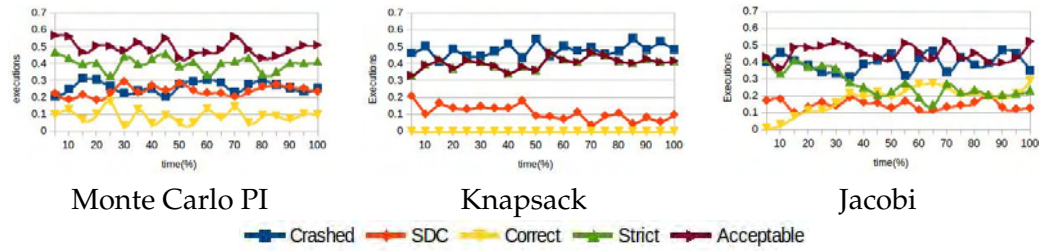
FIGURE 5.6: Correlation of the timing of fault injection with the effect on the application.

to be uncorrelated with application behavior. This is reasonable, since the application iteratively produces random numbers, which are used to compute the final result. All iterations affect the final result similarly, therefore we did not expect different behavior with respect to the timing of the faults. On the other hand, *Knapsack* demonstrates a different behavior. The later the faults are injected, the more likely the results are acceptable. Faults corrupting data in a manner that does not result to values which converge towards the solution will be discarded on the following iteration, after applying the fitness function. This effect becomes more intense on each consecutive iteration of the algorithm. In *Jacobi*, faults inserted at the beginning of the execution tend to result to strict correctness. The later the faults are injected, the more the correct results at the expense of strictly correct. Given that the input matrix is diagonally dominant, errors which do not alter important variables of the application (etc. iterators) but alter input or intermediate data, will have no significant effect to the results, since the algorithm is bound to converge. However, more iterations may be needed to achieve convergence.

5.2.8 GemFI Performance Evaluation

In order to evaluate the overhead of GemFI we compare the execution time for simulated runs of all the aforementioned benchmarks on both GemFI and the unmodified Gem5 simulator. We measure and compare simulation time for the part of the application for which fault injection is active (between *fi_activate_inst()* calls). Despite activating the fault injection functionality, in this set of experiments we do not actually inject any faults in the GemFI simulations. Should we inject faults, the behavior of applications would potentially change, thus making the comparison between the two tools infeasible. It should be noted that, despite the fact that no faults are injected, all GemFI functionality is activated — especially the modules of GemFI that are executed on each simulated cycle, thus resulting to most overhead — apart from the last step of the process described in Fig. 5.2, the fault injection itself. However, the actual fault injection step would, in any case, be activated only once, with negligible overhead. Moreover, since no faults are injected, there are no opportunities to switch to atomic simple mode after fault manifestation, therefore the simulation is performed in the high-overhead O3 CPU model.
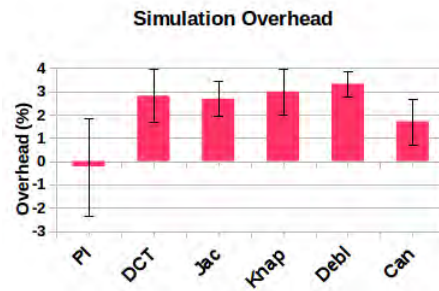
FIGURE 5.7: GemFI average overhead compared with unmodified Gem5. The chart also depicts the 95% confidence interval for each application.

Fig. 5.7 depicts the experimental results, which can be considered as a worst-case overhead scenario for GemFI. The overhead varies from -0.1% to 3.3%. It is mainly dependent on the number of instructions simulated with fault injection enabled. The overhead introduced by GemFI clearly is minimal. For PI estimation GemFI appears to perform better than Gem5, however this observation is not statistically significant.

Using the checkpointing methodology presented in Sec. 5.2.5, GemFI is able to significantly reduce the time for executing simulation campaigns. Fig. 5.8 summarizes the simulation time for the campaigns discussed in Sec. 5.2.7, with and without using the checkpointing capability to fast-forward the simulation to the point where fault injection is activated. The benefit from checkpointing is a 3x to 244x (64.5x on average) speedup with respect to the non fast-forwarded execution of the campaign. The speedup is mainly dependent on the ratio of the execution time spent for each application on the pre- and post-checkpoint code.
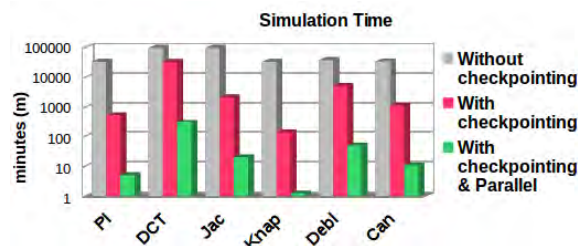
FIGURE 5.8: Effect of GemFI optimizations on the execution time of fault injection campaigns (y-axis in logarithmic scale).

The third set of bars in Fig. 5.8 depicts the execution time of the simulation campaigns on a network of 27 workstations, using the meta-simulation infrastructure discussed in Sec. 5.2.6. Each workstation is equipped with quad core Intel Xeon E5520 CPUs at 2.27 Ghz and 8 GB RAM each. On each workstation we execute simultaneously 4 experiments (simulations). The additional speedup, compared with execution on a simple system with checkpoint-based fast forwarding, is consistent with the number of simultaneously executed experiments (in all cases approximately 108x).

5.3 XM²: A Framework for Evaluating Software on Reduced Margins Hardware

XM² adopts a reusable, platform-neutral approach that can scale in simultaneous multi-board, multi-CPU and multi-process execution campaigns with or without operating system support. The framework supports user-defined methods for the collection and classification of the different execution outcomes, and can manage very large campaigns, thus relieving the user from manually initiating, controlling and monitoring the experiments.

5.3.1 Platform Requirements

XM² facilitates the design and implementation of experimental campaigns to characterize either the hardware itself, or the resilience of software operating on top of overclocked or undervolted hardware. XM² can be used on top of different hardware platforms and software stack configurations. It assumes the following support from the underlying hardware and software of the platform used for the experiments:

Hardware support: The hardware must provide support for controlling and scaling the system operating point (voltage, frequency) beyond the normal working envelope. Modern Intel x86-64 CPUs offer such capabilities, starting from the Haswell family, through the programmable Fully Integrated Voltage Regulator (FIVR) [14]. Several processors based on ARM architectures offer similar functionality. The AppliedMicro X-Gene 2 [95] chip does so through the SLIMpro management processor included in the chip. The ARM Cortex A53 processor in Raspberry PI 3b boards can be set to operate in non-nominal conditions via a configuration file.

Compiler support for common function attributes: Every application using XM² must be linked with a thin library, used to notify external systems about the execution status of the application. It also undertakes the management of the application and supports data exchange with external systems. The library exploits the common function attributes *constructor* and *destructor* provided by the *gcc* compiler.

Connectivity: We assume that the target (tested) system can support TCP connections to other systems. These are used to orchestrate the execution campaign, to supply input data, and to collect results of the computation from the target system. The TCP functionality is provided either by the OS, or directly by the aforementioned library when running on bare metal.

Remote reset support: When operating at extended configurations, errors leading to full system failure are likely. Therefore, the target system needs to offer a hardware interface for a full/clean reset.

5.3.2 Tool Design and Configuration

An experimental characterization campaign on top of unreliable hardware typically involves the execution of multiple experiments under the same configuration (in
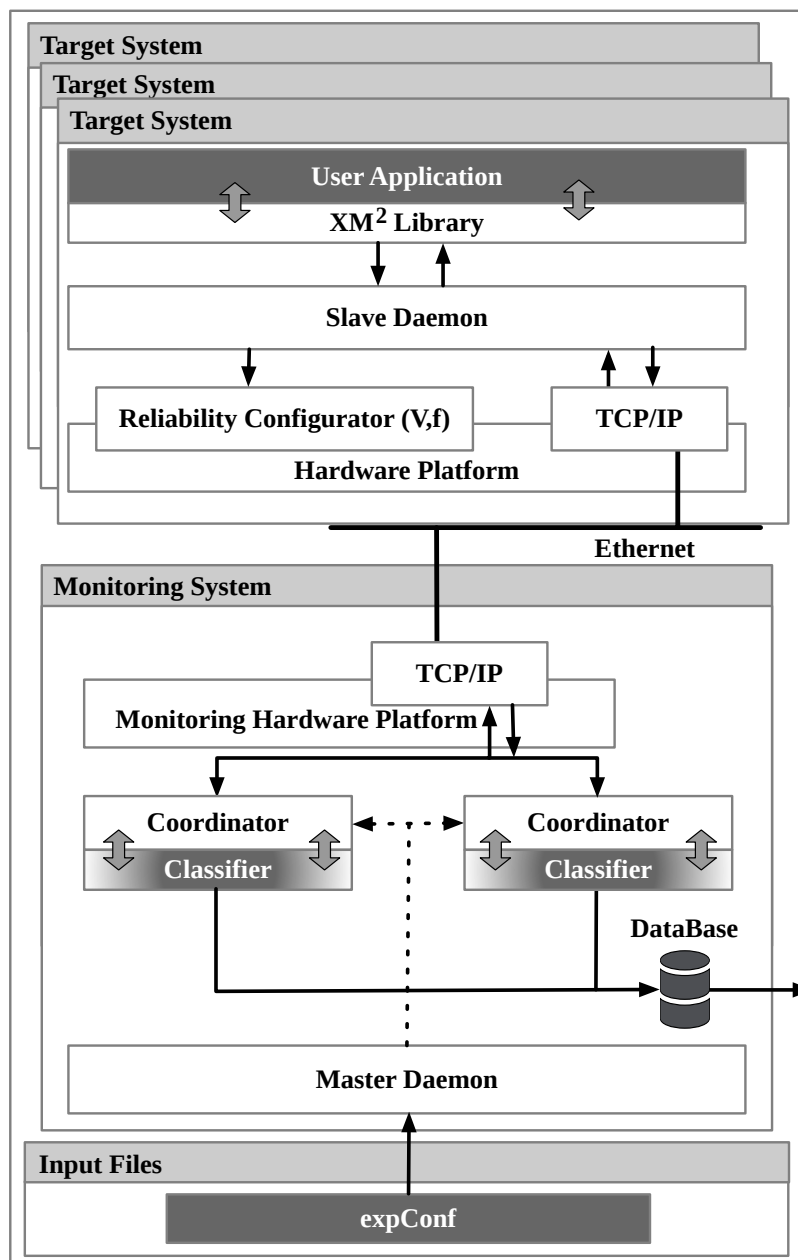
FIGURE 5.9: System architecture of XM^2. It comprises a single *monitoring* system and multiple *target* systems. The components corresponding to dark gray boxes are supplied by the user. XM^2 includes a built-in classifier of results, however the latter can be substituted by a user-provided one.

terms of the underlying hardware, its voltage/frequency configuration, the input set of the application etc). After each experiment terminates, its results are checked and classified, depending on potential effects of faults. This experimental procedure continues until the number of experiments is sufficient to provide statistically significant results.

Figure 5.9 presents a high level overview of XM^2. It is structured in a distributed way, comprising a single *monitoring* system, and one or more *target* systems of the same hardware architecture. The monitoring system deploys a *Master* daemon which

spawns a *Coordinator* thread for each target system. Every target system spawns a *Slave* daemon, which receives commands from the *Coordinator* and orchestrates the experimental campaign locally, through the library which manages/invokes the application.

5.3.3 Configuration File

We employ a configuration file that allows the user to define an experimental campaign by using a single file called *expConf*. The user defines the following parameters:

Target Application: An absolute path to the binary file which will be executed on the target system.

Target System: The Internet Protocol (IP) or Media Access Control (MAC) addresses of the target system(s).

Input File: Input of the application to be executed on the target system(s). XM^2 supports only a single input file per application. If the application requires multiple inputs, they need to be combined into one file by the user.

Operating Configuration: The voltage and frequency settings for the reliable ($Nominal$) and unreliable ($unRel$) configuration of the target system. Applying aggressive overclocking or undervolting settings increases the frequency of errors. Notably, the user can change simultaneously both the frequency and the supply voltage in the $unRel$ configuration.

Result Classification: XM^2 comes with a default classifier, which characterizes the outcome of each experiment as: (i) *Exact:* if the result is identical to that of a nominal execution; (ii) *SDC:* if the result differs from that of a nominal execution; (iii) *Data Abort:* if the CPU raised a data abort trap due to accessing a non existent physical memory address; (iv) *Illegal Instruction:* if the CPU raised a trap because it detected an non-existent opcode; (v) *CPU Crash:* if the execution time exceeds, by far, the time of a nominal execution.

Nominal outputs for the same target may differ, for example in multi-threaded applications which use floating point arithmetic. To be flexible, XM^2 allows users to provide their own classifiers that implement a customized comparison between the golden results and the application output. For example, one can use a deviation threshold to detect erroneous results.

Termination Criteria: The user may define a custom binary which is used by the XM^2 to determine when to terminate a campaign. The XM^2 invokes the user defined binary using as input the number of experiments classified in each category. The default termination checker terminates a campaign simply when reaching a predefined number of experiments, which can be set by the user via the *expConf* file.

Nominal Experiments: The user defines the number of experiments to be performed by XM^2 in *Nominal* setting. These experiments are used to profile the execution time of the application and to obtain the error-free (golden) output files.

void readInput(void *ptr, size_t sz,size_t nmemb):	Receives *nmemb * sz* bytes from the input file available at the *Coordinator* filesystem and stores them to the memory region pointed to by *ptr*.
void writeOutput(void *ptr,size_t sz,size_t nmemb)	Sends *nmemb * sz* bytes from the memory region pointed to by *ptr* to the *Coordinator*.
void switchToRel()	Switch to $Nominal$ state. The implementation of the function is architecture dependent. It is *blocking* – the *Slave* daemon needs to acknowledge the state switch to the *Coordinator*.
void switchToUnRel()	Switch to $unRel$ state. Similar semantics as *switchToRel*.

TABLE 5.2: API to the run-time library of XM^2.

5.3.4 Run-time Library API

The run-time library that accompanies XM^2 needs to be linked with the target application. It offers an API that enables the application to control data exchange with the *Coordinator* node, as well as hardware switching between the *Nominal* and *unRel* states. Table 5.2 lists the primitives of the API.

5.3.5 Example

Listing 5.3 provides an example of the *expConf* configuration file. The file initially assigns a name to the target system and specifies it using its *IP* and the *MAC* address. In this example there are two target systems (*PC_A, PC_B*). The specified *Nominal* operating point is used by XM^2 to compute error-free outputs (golden) as well as to determine the normal execution time of the application for each target system. The configuration file also includes a list of $unRel$ configurations. A separate experimental campaign will be executed for each of those configurations.

The configuration file also provides the paths to the application binary and the input file. The keyword *Monitor* indicates that these files reside in the monitoring filesystem and need to be fetched over the network. In this example, the user also specifies the *Classifier* binary, which will be used to classify the outcome of each experiment/run. The user specifies the number of experiments (10) to be performed by XM^2 in the *Nominal* configuration to obtain the golden output file. Finally, the user defines the maximum number (500) of experiments to be performed on each $unRel$ configurations.

Listing 5.4 outlines the modified source code of a mini-application implementing a *Sobel* filter. Lines 4,5,7 and 8 contain function calls to the run-time library of our tool. Finally, Listing 5.5 outlines a classifier which categorizes experiments as *Exact, Acceptable, SDC*. *Exact* experiments are those that produce a bit-wise exact

```
1  {
2    "TargetSystem:{
3    "idName":"PC_A"
4      "IP Address":"10....",
5      "MAC address": "AA:BB:CC:DD:EE:FF",
6      "Nominal": [1.2,1200],
7      "unRel": [ [1.2, 1320],[1.2, 1330],[1.0, 1320]]}
8    "TargetSystem_:{
9    "idName":"PC_B"
10     ....}
11   "Application":{
12     "Path":["pathToExecutable","Monitor"]
13     "InputFiles":[["pathToInput",262144, "Monitor"]]
14     "Classifier":"/path/to/psnr.exe"
15     "Termination" : { "default":{500}},
16     "NominalExp" : {10}}
17 }
```

LISTING 5.3: An example *expConf* file using the *json* format.

```c
#include <FIOrchestrator.h>
void sobel(unsigned char* in, unsigned char *out);
int main( int argc, char* argv[]){
  readInput( in, sizeof(char), SIZE);
  switchToUnRel();
  sobel(in,out);
  switchToRel();
  writeOutput( out, sizeof(char), SIZE);
  return 0
}
```

LISTING 5.4: Source code of the application extended with calls to the run-time API.

```c
#include <FIOrchestrator.h>
float PSNR(unsigned char* gld, unsigned char *tst);
int main( int argc, char* argv[]){
  float res = PSNR(gold,test);
  if ( isinf(res) & 1)
    printf("Exact inf\n");
  else if ( res > 50.0 )
    printf("Acceptable \%f\n",res);
  else
    printf("SDC \%f\n",res);
  return 0
}
```

LISTING 5.5: Source code of a custom classifier.

copy of the result of the error-free execution. *Acceptable* experiments produce outputs with a *PSNR* higher than 50.0 dB in comparison with a *golden* output. All other experiments are categorized as *SDC*. Note that the classifier is not invoked if the application terminates abruptly due to a runtime error or a crash. In this case, the framework automatically classifies the experiment as *CPU Crash, Illegal Instruction*, or *Data Abort*.

5.3.6 Flow of a Fault Injection Campaign

Figure 5.10 illustrates a simplified time-line of a fault injection campaign controlled by our tool. Initially, the user provides the *expConf* file to the *Master Daemon*. The
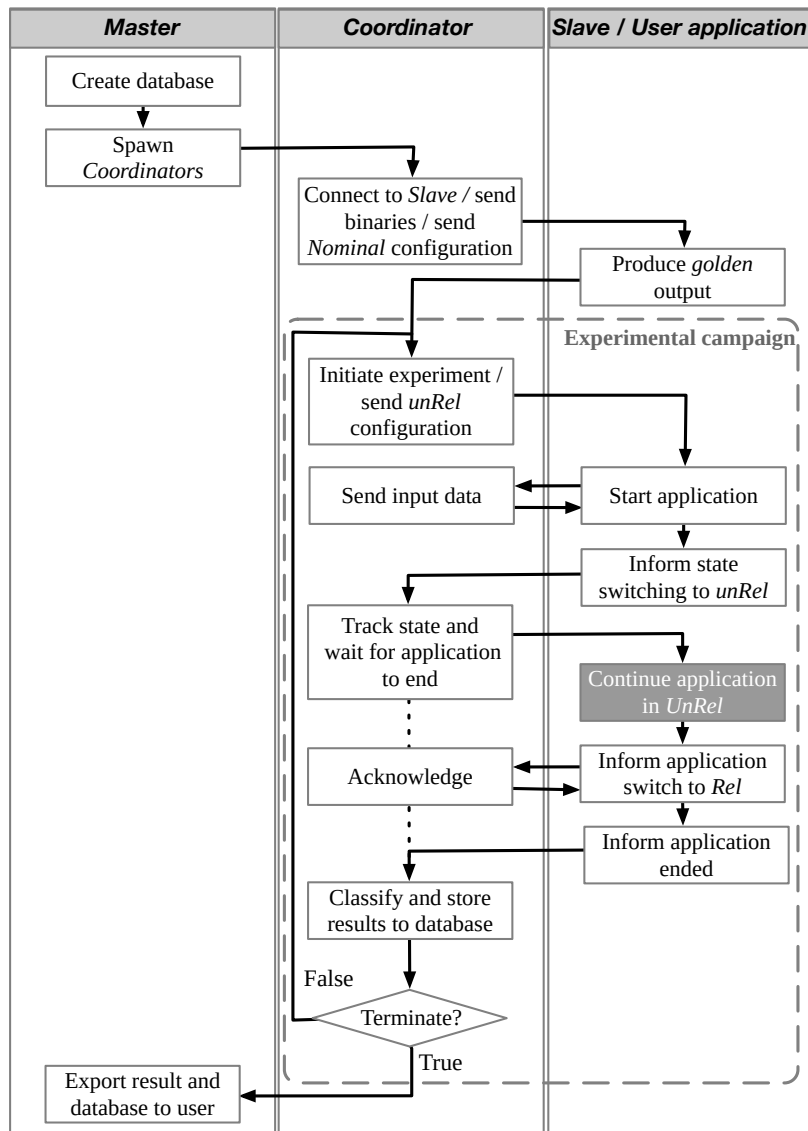
FIGURE 5.10: Flow chart for the main steps performed by XM² for the basic case of an experimental campaign that does not result to crashes. The dark box is the only state where the target system is configured at an unreliable state.

daemon creates a new database and spawns a *Coordinator* thread for each of the target systems. The *Coordinator* connects to the respective *Slave* daemon on the target system using the *TCP* protocol, and transfers the *Nominal* configuration, the application binary and inputs to the target system. The inputs will be used when the target application uses the *readInput* function.

The *Slave* daemon performs the *Nominal* number of experiments as specified in the *expConf* (without transitioning to the $unRel$ state). The purpose of this step is to produce error-free golden outputs as well as to profile the time required to execute the application under nominal conditions. The golden file is used by the classifier for comparison against the outputs of unreliably executed code.

At this point the actual experimental campaigns start. The *Coordinator* sends the configuration parameters of the $unRel$ state to the *Slave*. These parameters will be

used for all subsequent experiments. The *Slave* then spawns the application. Any requests to read input data by the application using the XM^2 API calls are handled transparently. Before a *Slave* transitions to the $unRel$ state it notifies the *Coordinator* first. At this point the *Coordinator* starts a watchdog which waits for the application to terminate. The maximum waiting time is equal to the profiled time when the code was executed reliably, increased by 10%. In case the $unRel$ frequency is lower than the frequency of the *Nominal* point, we proportionally increase the waiting time to match the maximum expected performance degradation due to frequency scaling. If the *Coordinator* does not receive any information about the application status within this period, the *Slave* is reset and the corresponding experiment is flagged as *CPU Crash*.

If the application terminates abruptly, e.g example due to executing an *Illegal Instruction*, the *Slave* informs the *Coordinator* and the experiment is classified accordingly. In case the application terminates normally, the *Slave* sends all output data, as defined by the *writeOutput* call, to the *Coordinator*.

Afterwards, the *Coordinator* invokes the classifier binary to characterize the experiment. If the experiment is not flagged as *Exact*, the *Coordinator* resets the *Slave* so that it is re-initialized to a valid state. The *Coordinator* then evaluates the termination criterion and either terminates the campaign or proceeds to deploy the next experiment. When a campaign terminates, XM^2 prints the statistics for the different experiment classifications, as well as the path to the output database. In the database, we store the classification of each experiment and a path to the raw output data of executions. If there are more $unRel$ configurations specified in the *expConf*, a new experimental campaign starts, otherwise the tool terminates.

5.3.7 Evaluation

To evaluate our framework we use three *Raspberry PI 3b boards* (Table 5.3) as target systems. We set *Nominal* configuration to $f = 600MHz$, $V_{dd} = 0.8V$. Even though XM^2 supports both undervolting and overclocking, for the evaluation we perform overclocking. We overclock the system by providing a list of $unRel$ states starting from $V = 1.2V$, $f_u = 1370MHz$ with intermediate steps increasing f_u by $10MHz$, up to the highest frequency state ($V = 1.2V$, $f_u = 1450MHz$). The termination criterion for the experimental campaign is a number of experiments equal to 2000, which provides a confidence level of 98% and an error margin of 2.5%. For the evaluation, we use the default classifier.

We use *Circle*, a C++ library supporting execution on bare metal, to evaluate the error resiliency of software under unreliable execution without any interference from the OS software stack, e.g. scheduler of Linux kernel or background OS services. *Circle* provides several C++ classes which selectively enable or use different hardware features (*MMU, Interrupt Support* etc.).

The target systems are reset whenever necessary using a small circuit per system, which employs a transistor operating as a switch to connect the respective reset pins

System On Chip	Broadcom BCM2837
Instruction Set	ARMV-8
CPU	4x ARM Cortex-A53, 1.2Ghz
	In Order
	Dual Instruction Decode and execute
	6KB Conditional Predictor
	256 Entry Indirect Predictor
	NEON advanced SIMD
	8 - 64K I-Cache With Parity
	8 - 64K D-Cache With Ecc
RAM	1 GB LPDDR2 (900MHz)

TABLE 5.3: Raspberry 3B Specifications
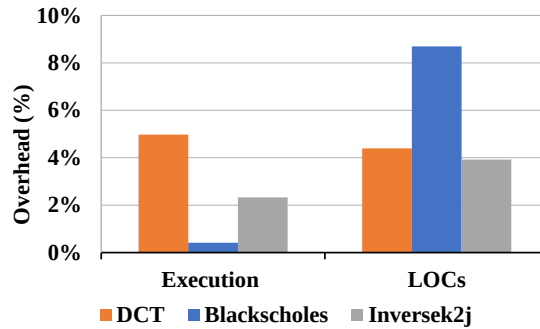
FIGURE 5.11: Overhead of XM^2 in terms of execution time and additional lines of code (LOC) when compared to a native execution and the original version of the code respectively.

on the *Raspberry PI 3b*. The circuits are controlled by the monitoring system through a serial interface.

We evaluate the programmers' effort to use our tools in terms of extra lines of code (LOCs) that are introduced to the source code of an application. Moreover, we quantify the communication overhead introduced by XM^2 between the *Coordinator* and the target system. We use three benchmarks: *Blackscholes [9], Inversek2j, DCT*. *Blackscholes* implements a mathematical model for a market of derivatives, *Inversek2j* calculates the angles of a 2-joint arm using the kinematic equation and *DCT* is a module of the JPEG compression and decompression algorithm.

Figure 5.11-left presents the execution time overhead (%) due to the communication protocol and data exchange between the *Coordinator* and the target. XM^2 adds, in the worst case (*DCT*), an extra 5% of execution time compared with a native execution on the target platform under the same configuration. The execution time to compute *DCT* is not negligible, compared to the time needed to transfer the data. Consequently, this benchmark results to the highest overhead. The remaining benchmarks are mainly compute-bound. On average XM^2 introduces an execution time overhead of 2.5%.

Figure 5.11-right illustrates programmers' effort to prepare an application for our framework. In *Blackscholes*, the developer needs to unpack and pack the input/output data prior to transferring them, thus the volume of the new code is equal to 8.7% of the existing one. The remaining two benchmarks are small in terms of LOCs, so

even small code additions produce a large overhead (%). In both cases we simply replace the *fread, fwrite* functions with *readInput, writeOutput* of the XM² API. Moreover, we add two extra function calls to switch between states. On average, preparing applications for XM² requires 5.6% extra LOCs.
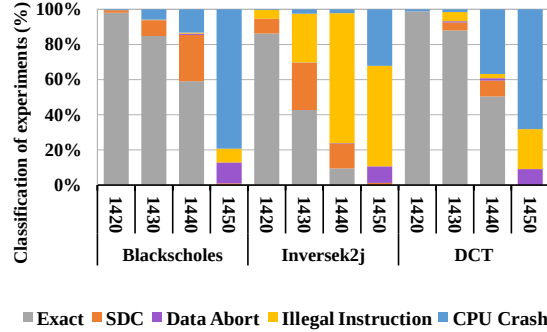
FIGURE 5.12: Experimental results for different applications and different overclocked configurations.

Figure 5.12 shows the experimental campaign results evaluating the reliability of the system under different overclocked configurations. *Blackscholes* uses double precision arithmetic. Due to the representation of such numbers, faults are unlikely to be masked. Therefore, this benchmark suffers the highest percentage of *SDCs* (up to 26%) when executed on $f_u = 1440MHz$. *Inversek2j* uses primarily trigonometric functions, which heavily rely on branches. The experiments indicate that faults corrupt the computation of the target address, resulting in decoding memory that does not contain instructions. Consequently, 74% of the experiments result to *Illegal Instructions* when executed at $f_u = 1440MHz$. Finally, 36% of *DCT* experiments result in *CPU Crash* when executed at the same frequency. This benchmark employs six nested loops to iterate through the image pixels and apply the coefficient transformation. Corruptions in the control flow of these loops often results to infinite loops. Therefore, execution is terminated by the watchdog and experiments are classified as *CPU crashes*.

5.4 Arm Cortex A53 Vulnerability Analysis

In this section we demonstrate the versatility of XM² using three case studies. The first two studies focus on recording and analyzing the behavior of small kernel programs running on the overclocked *Raspberry PI* platform, whereas the third study focuses to study the behavior of compiler and source code tranformations on the $max f_{margin}$ of the applications.
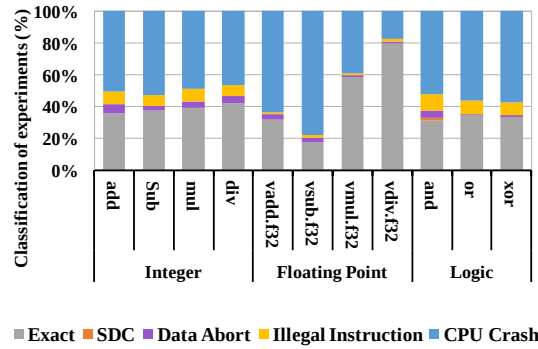
FIGURE 5.13: Experimental results of the instruction error resiliency characterization when $V_u = 1.2V$, $f_u = 1450MHz$. The X-axis shows the different microkernels and the Y-axis presents the classification of the experiments according to the effects of overclocking on execution.

5.4.1 Instruction Level Error Resiliency Analysis

Initially, we employ our tool to assess the resiliency of ARM instruction when executed individually in the overclocked Cortex A53 pipeline causing minimal disruptive events such as cache misses and branch mispredictions (Listing 5.6). We selected a subset of instructions that perform integer (*add, sub, mul, div,*), floating (*vadd.f32, vsub.f32, vmul.f32, vdiv.f32*), and boolean (*or, and, xor*) arithmetic. The microkernels use only four registers, two as input, one for temporary storage, and one to accumulate the final result which is propagated to the *Coordinator* after the end of the execution.

In our case, we evaluate the error resiliency of instructions using one input file which sets the register to the value of 1. However the user could define multiple input files and perform multiple experimental campaigns. Finally, the *Nominal* and $unRel$ configurations are the same as in the previous section.

We observe that microkernels which execute multiple times the same instruction, regardless of the instruction, produce exact results even when we increase the CPU frequency by 20% (from $f_u = 1200MHz$ to $f_u = 1440MHz$). When we overclock by an additional $10MHz$ the reliability of most kernels significantly drops as shown in Figure 5.13. This abrupt fall in reliability confirms previous findings that there are (V, f) settings called *Points of First Failure (PoFF)* at which circuits start to exhibit massive errors.

```
1 for (){
2 instruction r0, r1, r2
3 instruction r3, r3, r0
4 ...
5 }
```

LISTING 5.6: Template of microkernels used to stress the same execution path of the Pipeline.
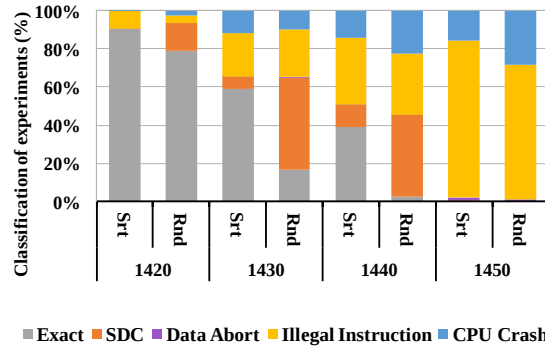
FIGURE 5.14: Experimental results stressing the branch predictor for the two microkernels for different overclocked frequencies (f_u).
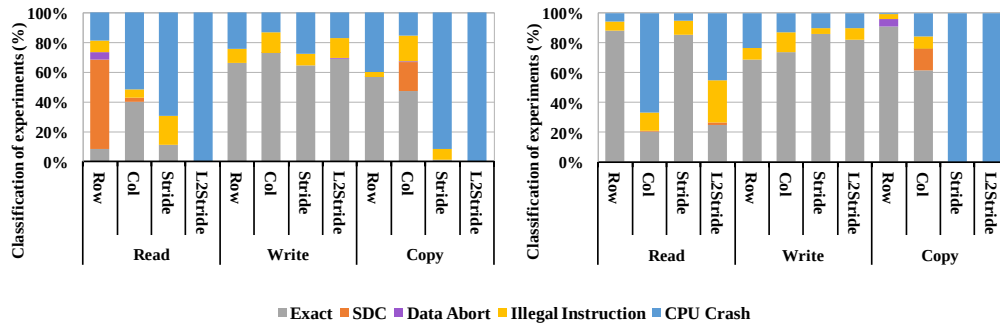
FIGURE 5.15: Experimental results of the *Cache* microkernels of Table 5.4 for $unRel = (1.2V, 1430MHz)$, when the hardware prefetcher is enabled (left) and disabled (right). The Y axis presents the classification of the experiments according to the effects of overclocked execution.

5.4.2 Error Resiliency of Source Code and Algorithm Transformations

In this case study, we demonstrate how XM² is used to identify source code transformations with a large effect on application resiliency. We focus on transformation which affect branch prediction mechanisms and the cache hierarchy.

To evaluate the vulnerability of the *branch prediction* mechanism we employ two simple kernels. The *Sorted* kernel traverses an array that contains sorted values and compares each value with the mean of the array. Depending on the outcome of each comparison a variable is increased or decreased. The branch predictor is able to predict correctly the behavior of the branches in the *Sorted* version. The second kernel, called *Random*, traverses the same array, however, as the name implies, the values are stored randomly within the array resulting to a very high misprediction rate (and subsequent pipeline flushes).

Figure 5.14 shows that the *Random* microkernel has higher percentages of *SDCs* across all frequencies. The two kernels have similar behavior only for extreme overclocking $f_h = 1450MHz$, leading to an increased percentage of *Illegal instructions*. We believe that this is due to the high branch misprediction rate which set the Program Counter to memory addresses without valid code or to a memory wrong segment.

For the cache we create kernels which perform *read, write* or *memcopy* using different memory access patterns (Table 5.4). Figure 5.15 presents results of the cache evaluation. Note that the *write* operation shows a higher degree of robustness in comparison with the rest of the operations. As the CPU automatically enables the read allocate mode during the execution of *write* microkernels, the cache is barely utilized. The *read* operations are slightly more robust than the *memcopy* operations. Actually, the *memcopy* operations usually result in *CPU Crash*, whereas the *read* operations result in *Illegal instructions*. When comparing the different access patterns, the more complex the pattern, the higher the number of experiments which terminate abnormally. In the case of the *L2 Stride*, all experiments result in *CPU Crash*.

We observe that the *Strided patterns* have high *CPU Crash* probabilities. We assume that the prefetcher may have a negative impact on reliability. To validate our assumption we programmatically disable the prefetcher and recompile the cache microkernels. Executing these binaries is trivial since the *expConf* files are the same and only the attribute describing the binary paths should change. The results of the fault injection campaign without the prefetcher are presented on the right side of Figure 5.15. In general, deactivating the prefetecher increases slightly the application resiliency, however in the strided patterns the results remain the same.

5.4.3 Compiler Optimizations VS Frequency Margins

We use the *gcc 4.9.3* compiler. While modern compilers provide users with specific options to optimize their code, individual optimizations are usually grouped in higher-level options, such as O0, O1, O2, O3, Os. Our study only considers these options.

Row	This pattern accesses bytes in the same order as they are stored in main memory. This is the optimal way to access the memory resulting in lowest *L1,L2* cache and *TLB* miss rate.
Col	This pattern accesses the first byte of each memory page, leading to the worst performance of the memory access. Each memory access causes a L1 and L2 cache miss.
Stride	This pattern iterates through the 2D array using a stride equal to the cache line (64 bytes). We disable the hardware prefetcher, so that every memory access leads to a L1 and L2 cache miss. The prefetcher would have been able to detect this strided pattern.
L2Stride	This pattern is similar to *Stride*. We disable the hardware prefetcher, so that every memory access leads to a L1 and an L2 cache hit. Similarly to *Stride*, the prefetcher would be have been able to detect this strided memory access pattern.

TABLE 5.4: Different memory access patterns used by the source code transformation case study.
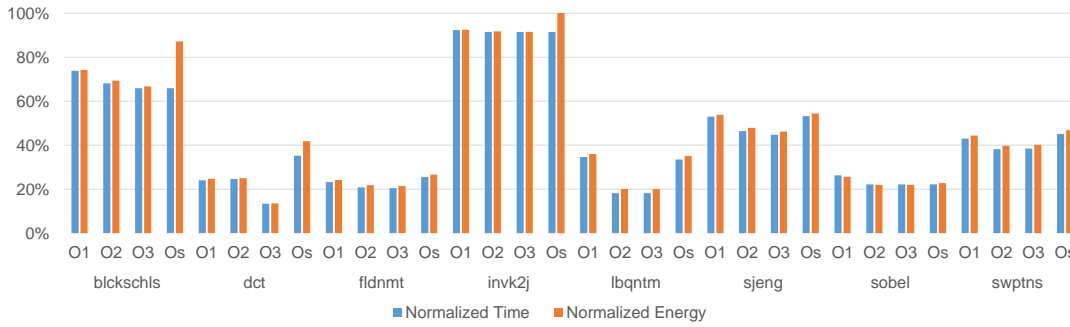
FIGURE 5.16: Execution time and energy consumption of the application benchmarks for the different compiler optimization levels, relative to O0.

We use several applications/kernels taken from various benchmark suites [40, 9, 146]. In this study, we analyze *Sobel, DCT, Inversek2j, Blackscholes, Swaptions, Fluidanimate, Sjeng* and *Libquantum. Swaptions* uses the Heath-Jarow-Morton framework to price a portofolio of swaptions. *Sjeng* is a chess-player application that finds the next move via a combination of alpha-beta and priority proof-number tree searches. Finally, *Libquantum* simulates a quantum computer.

Compiler optimizations aim at improving performance, we first analyze the effects of the different optimization levels (O0, O1, O2, O3, Os) on the execution time and energy consumption of our benchmark applications. Increasing the optimization level augments the previous set of optimizations with additional ones. In the case of Os, the compiler uses most, but not all, of the O2 optimizations, together with some extra optimizations that decrease the size of the executable.

Figure 5.16 shows the normalized energy consumption and execution time of the different compiler optimization levels with respect to O0. As expected, the higher the compiler effort the greater the performance and the energy gain. *DCT* presents the higher speedup when using the $O3$ optimizations. On the other hand, *Inversek2j* shows almost no speedup when compiled with increasing optimization levels. This is because it extensively uses trigonometric functions that are included in an already optimized version of the standard C library. According to our measurements, the different optimization levels do not impact CPU power consumption in a significant way, except of the case of Os level, which in some applications (*Blackscholes, DCT, Inversek2j*) increases the power consumption. This is due to the instruction selection performed on this optimization level as well as that alingment and function inlining is not performed. In any case typically the energy gains when using higher optimization levels are mainly due to the reduced execution time.

Figure 5.17 illustrates the experimentally identified $maxf_{margin}$ for the different optimizations levels, on the four raspberry PIs; The exploitable extra frequency ranges from 9% to 19% of the nominal CPU frequency ($1200MHz$). The highest frequency at which all applications can be executed reliably, is equal to 1309, 1356, 1346, 1356 Mhz for the four raspberry PIs, corresponding to a CPU part-specific *static*
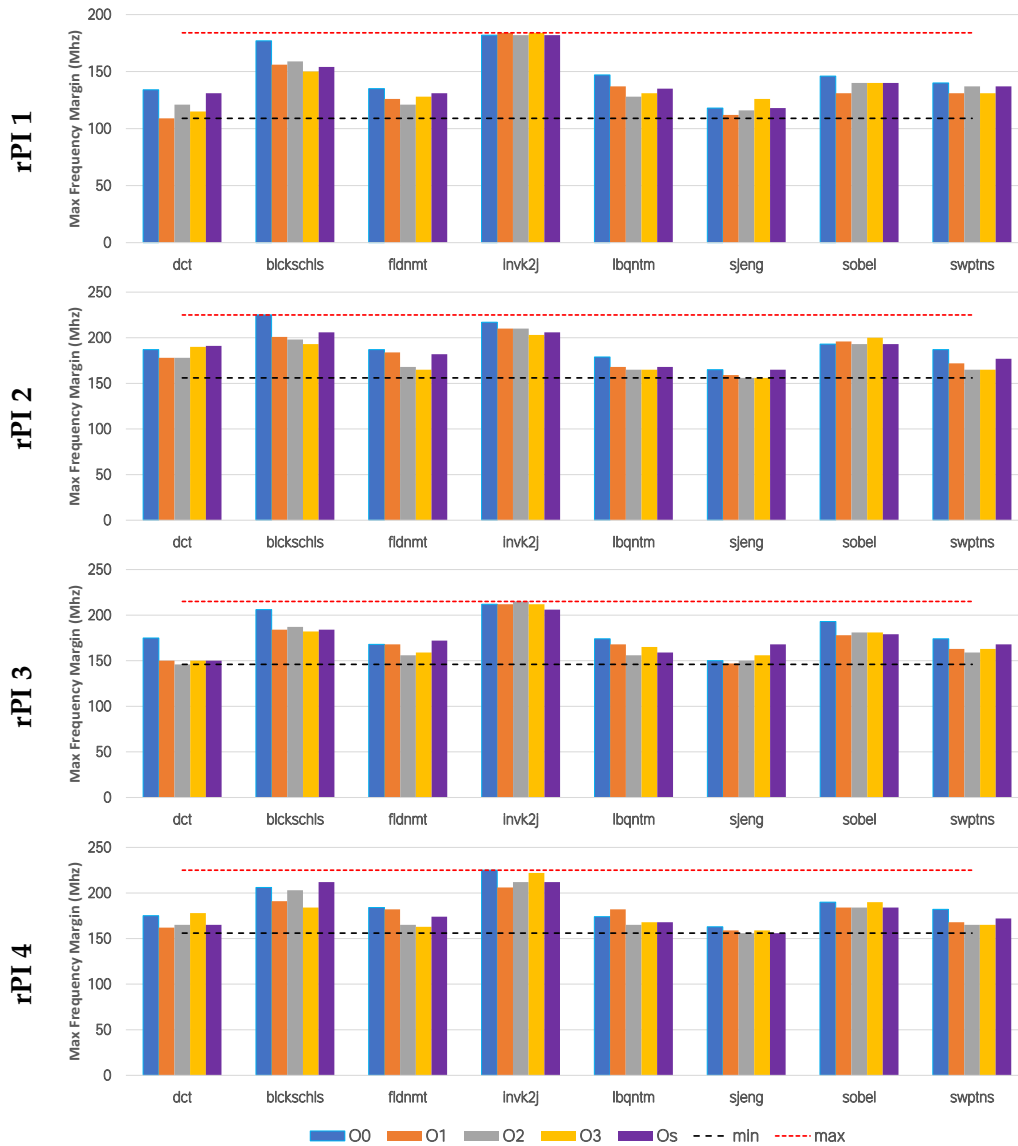
FIGURE 5.17: Evaluation of $maxf_{margin}$ settings for 8 benchmarks (1000 runs each) in each raspberry PI; the higher the bar, the wider the exploitable frequency margin. The horizontal dotted lines show the maximum (red) and minimum (black) values of $maxf_{margin}$.

frequency margin of $109, 156, 146, 156Mhz$, respectively. The workload-specific dynamic frequency margin for the four raspberry PIs is equal to $75, 69, 69, 69Mhz$ respectively

Different optimization levels impact the dynamic frequency margin and can increase or decrease $maxf_{margin}$ by up to $32Mhz$ for a given application. Interestingly, *O0* has a wider margin than higher optimization levels for the same application, in 62.5% of the configurations (combinations of different CPU parts and different applications). Despite the increased $maxf_{margin}$ of O0, the decrease in the execution time due to the extra frequency margin is relatively small, resulting in lower energy gains compared to higher optimization levels. Thus, using higher optimization levels is more beneficial not only in terms of performance but also in terms of energy
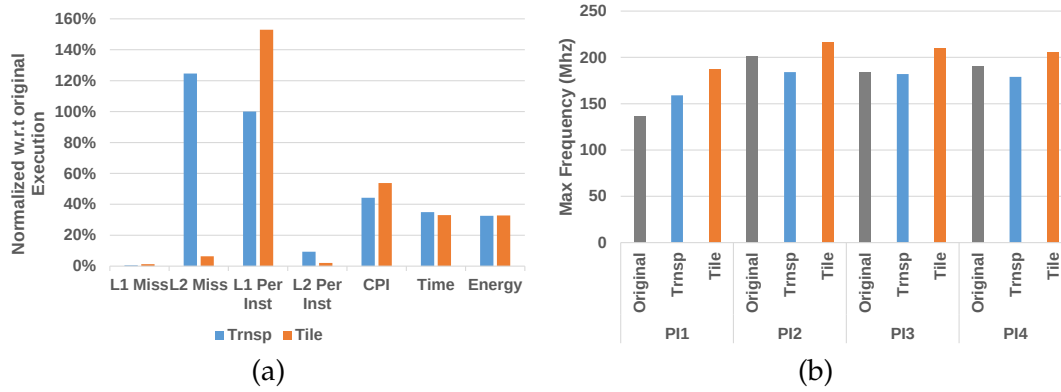
FIGURE 5.18: (a) Performance metrics and energy concumption of the transposed and tiled MM versions, with respect to the original implementation. (b) $maxf_{margin}$ for all raspberry PIs and all MM implementations.

gains, even though these have smaller frequency margins than O0. When comparing the remaining optimization levels (O1,O2,O3,Os) there is no dominant optimization level in terms of frequency margins. On the other hand, in 80% of the total cases $O3$ is the most energy efficient optimization level.

Source Code Transformations

Developers very often try to reduce the execution time of their applications by employing more efficient algorithms, optimizing memory accesses, reducing the number of instructions, or using special instructions for parallel processing and vectorization. In this section, we optimize a Matrix Multiplication (MM) kernel by using more efficient memory access patterns as well as Single Instructions Multiple Data (SIMD) instructions. In both cases we observe the effects of the optimizations on the energy efficiency, the execution time and the $maxf_{margin}$ of the different benchmark versions.

Memory Access Pattern Optimizations

The matrix multiplication (MM) kernel performs multiplication between two floating point matrices ($C = A * B$). We consider three different implementations/versions. The so-called original version accesses the first matrix (A) in a row-wise fashion and the second matrix (B) in a column wise fashion. The second implementation, performs a multiplication with the transposed B^T matrix, which is allocated on a new 2D-array. Finally, the third version uses a tiled version of the matrix multiplication. The size of the tile is equal to the cache line size (64 bytes).

Figure 5.18a presents the performance metrics and energy consumption of the transposed and tiled MM versions, normalized to the original implementation. As expected, both optimized versions have significantly lower L1-cache misses. They also demonstrate a significantly decreased CPI, which directly translates to performance and energy gains.

(a)   (b)
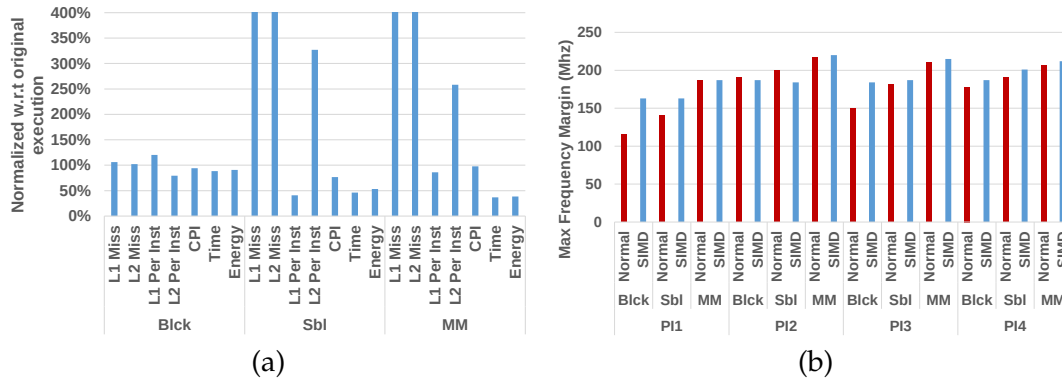
FIGURE 5.19: (a) Normalized performance metrics and energy consumption of the three benchmarks, with respect to the implementations without SIMD instructions. (b) $maxf_{margin}$ for all raspberry PIs and benchmarks.

Figure 5.18b presents the $maxf_{margin}$ of the different MM versions. In contrast to the compiler optimization analysis, where the non-optimized versions exhibit larger $maxf_{margin}$, the memory access optimizations present mixed results. On the one hand, in all raspberry PIs, the wider $maxf_{margin}$ is found for the tiled MM version, which in one of the raspberry PIs (PI1) yields an increase on the maximum frequency of up to $50Mhz$ compared to the original version. On the other hand, in three out of four PIs, the transposed MM version has lower frequency margins than the original. Also, on average the extra frequency margins results to an additional performance gain of 2.5%. We also observe margin variations across different parts, this difference can reach up to $63Mhz$ when comparing the original version of PI1 with the same version on PI2.

SIMD Optimizations

We use SIMD instructions to optimize the execution time and energy efficiency of *Blackscholes, Sobel* and *tiled MM*. Figure 5.19a presents the performance metrics and energy consumption, relative to the normal versions of the benchmarks without SIMD instructions. Figure 5.19b shows the frequency margins of the benchmarks for the four raspberry PIs.

As can be seen, when using SIMD instructions the execution time of *Sobel* and *MM* is decreased to 36% and 46% of the normal versions, respectively. This speedup is mirrored to energy gains since the power consumption does not increase significantly when using SIMD instructions. *Blackscholes* does not show any reduced execution time because many math functions used by that benchmark do not have a SIMD equivalent function. In PI1 the SIMD version of the blackscholes benchmark greatly increase the frequency margin by $44Mhz$. This increase in frequecny provides an extra performance gain of 3.5% on top of the performance gain obtained by the SIMD instructions. In general, the use of SIMD instructions on average increases the maximum frequency by *1%*.

	GemFI	XM^2
Fault Injection Accuracy	Depends on fault model	Realistic
Fault Injection Control	High	None
Deter-menism	True	None
Execution Speed	Simulation Speed	Native

TABLE 5.5: Brief Comparison between GemFI and XM^2

5.5 GemFI versus XM^2

Table 5.5 present a high level comparison between GemFI and XM^2. The fault injection results of XM^2 are always realistic and accurate as they represent the manifestation timing faults as errors on the software level, on the other hand GemFI, does not provide a fault model, this is to be performed from the user. The higher the accuracy of the fault model used the higher the accuracy of the fault injection of GemFI.

GemFI as it is a simulation based fault injection tool offers a high control over the fault injection procedure. For example, the user knows exactly the number and the characteristics of the faults. Using this information the user can replicate as many times as he needs the exact same fault injection scenario. On the other hand, XM^2 does not provide such features. The only control the user has over the fault injection procedure is to select the operating point of an experiment. The user is not aware whether a fault was manifested in a hardware component and masked by the application resiliency. Moreover, a specific corruption on the output of an application cannot be replicated. Finally, XM^2 is way faster than GemFI as fault injection takes place in native execution speed. However, GemFI offers mechanisms which allow silmutaneous fault injection in multiple computer nodes, which increase the fault injection procedure significantly.

Chapter 6

Related work

This chapter discusses related work and the differentiation between our research and previous efforts. As approximate and unreliable computing are very active research subjects , we organize literature discussion across three different axis: approximate computing, unreliable computing, Power & energy-aware optimization, voltage margin characterization and fault injection tools.

6.1 Approximate computing

The authors in [89] present a thorough survey of the approximation techniques used in computing systems. They discuss the opportunities and obstacles in use of approximate computing and present techniques for finding approximable program portions and monitoring output quality, along with the language support for expressing approximable variables/operations.

Quickstep [87], is a tool that approximately parallelizes sequential programs. The parallelized programs are subjected to statistical accuracy tests for correctness. Quickstep tolerates races that occur after removing synchronization operations that would otherwise be necessary to preserve the semantics of the sequential program. Quickstep thus exposes additional parallelization and optimization opportunities via approximating the data and control dependencies in a program. However, Quick-Step does not enable algorithmic and application-specific approximation and does not include energy-aware optimizations in the runtime system.

Variability-aware OpenMP [104] and variation tolerant OpenMP [105], are sets of OpenMP extensions that enable a programmer to specify blocks of code that can be computed approximately. The programmer may also specify error tolerance in terms of the number of most significant bits in a variable which are guaranteed to be correct. We follow a different scheme that allows approximate –in our context, not significant– tasks to be selectively dropped from execution and dynamic error checks to detect and recover from errors via selective task restarting. Variability-aware OpenMP applies approximation only to specific FPU operations, which execute on specialized FPUs with configurable accuracy. In contrast, we explore selective approximation at the granularity of tasks, using the significance abstraction. Our programming and execution model thus provides additional flexibility to drop

or approximate code, while preserving output quality. Furthermore, our framework for significance aware approximate computing does not require specialized hardware support and runs on commodity systems.

Several frameworks for approximate computing discard parts of code at runtime, while asserting that the quality of the result complies with quality criteria provided by the programmer. Green [6] is an API for loop-level and function approximation. Loops are approximated with a reduction of the loop trip count. Functions are approximated with multi-versioning. The API includes calibration functions that build application-specific QoS models for the outputs of the approximated blocks of code, as well as re-calibration functions for correcting unacceptable errors that may incur due to approximation. Sloan et al. [130] provide guidelines for manual control of approximate computation and error checking in software. These frameworks delegate the control of approximate code execution to the programmer. Emeuro [85] efficiently breaks down an application into subroutines of varying granularity and automatically generates approximate alternatives for said subroutines through the use of Artificial Neural Networks (ANNs). At execution time, an intelligent runtime system explores the high-dimension subroutine space and generates a graph of computations which comprises nodes that are either accurate versions of the subroutines approximate ones through the use of ANNs. Additionally, Emeuro employs a denoising autoencoder-based heuristic to detect ANNs which are incapable of producing outputs of acceptable quality for a given input. To this end, each ANN is coupled with a denoising autoencoder (DAE) whose aim is to reconstruct the input of the ANN. If the difference, between the actual input of the ANN and the one reconstructed by its DAE, is larger than some user specified constraint the ANN is considered to be a sub-optimal choice. In this scenario, a different subroutine graph is investigated. We explore an alternative approach where the programmer uses a higher level of abstraction for approximation, namely computational significance, while the system software translates this abstraction into energy- and performance-efficient approximate execution.

Loop perforation [125] is a compiler technique that classifies loop iterations into critical and non-critical ones. The latter can be dropped, as long as the results of the loop are acceptable from a quality standpoint. Input sampling and code versioning [150] also use the compiler to selectively discard inputs to functions and substitute accurate function implementations with approximate ones. Similarly to loop perforation and code versioning, our framework benefits from task dropping and the execution of approximate versions of tasks. However, we follow a different approach whereby these optimizations are driven from user input on the relative significance of code blocks and are used selectively in the runtime system to meet user-defined quality criteria energy savings and performance gain. While these approaches demonstrate aggressive performance optimization thanks to approximation, they do not consider parallelism in execution. Furthermore, these techniques

operate at a granularity different than parallel tasks or specific runtime energy optimization opportunities which are exposed through approximation.

Several software and hardware schemes for approximate computing follow a domain-specific approach. ApproxIt [149] is a framework for approximate iterative methods, based on a lightweight quality control mechanism. Unlike our task-based approach, ApproxIt uses coarse-grain approximation at a minimum granularity of one solver iteration.

Other tools automate the generation and execution of approximate computations. SAGE [120] is a compiler and runtime environment for automatic generation of approximate kernels in machine learning and image processing applications. Paraprox [119] implements transparent approximation for data-parallel programs by recognizing common algorithmic kernels and then replacing them with approximate equivalents. ASAC [114] provides sensitivity analysis for automatically generated code annotations that quantify significance. Contrary to our work on automatic significance analysis, ASAC systematically perturbates the variables of a program and observers the results. It then, applies a hypothesis tester to check against a correct output and subsequently score each variable to rank its contribution to the output of the program. In our work, we rely on interval analysis in conjunction with automatic algorithmic differentiation to qualitatively estimate the contribution of different parts of a code to the application output quality.

Hardware support for approximate computation has taken the form of programmable vector processors [142], neural networks that approximate the results of code regions in hardware [29], and low-voltage probabilistic storage [117]. These frameworks assume non-trivial, architecture-specific support from the system software stack, whereas our approximate computing work depends only on compiler and runtime support for task-parallel execution, which is already widely available on commodity multi-core systems.

6.2   Fault Tolerant computing

Gschwandtner et al. [35] use a similar iterative approach to execute error-tolerant solvers on processors that operate with near-threshold voltage (NTC) and reduce energy consumption by replacing cores operating at nominal voltage with NTC cores. Schmoll et al. [123] present algorithmic and static analysis techniques to detect variables that must be computed reliably and variables that can be computed approximately in an H.264 video decoder. Although we follow a domain-agnostic approach in our framework, we provide sufficient abstractions for implementing the aforementioned application-specific approximation methods.

Topaz [1] is a task-based framework which executes computations unreliably. An online outlier detection mechanism detects and then re-computes unacceptable task results reliably. Chisel [88] selects approximate kernel operations to minimize an application's energy consumption while satisfying its accuracy specification.

Rinard *et al.* [112], in one of the chronologically earlier efforts on task-based unreliable computing, propose a software mechanism that allows the programmer to identify task blocks and then creates a profile-driven probabilistic fault model for each task. This is accomplished by injecting faults at task execution and observing the resulting output distortion and output failure rates. Task Level Vulnerability (TLV [105]) captures dynamic circuit-level variability for each OpenMP task running in a specific processing core TLV meta-data are gathered during execution by circuit sensors and error detection units to provide characterization at the context of an OpenMP task. Based on TLV meta-data, the OpenMP runtime apportions tasks to cores aiming at minimizing the number of instructions that incur errors. TLV does not consider error recovery and user-specified approximate execution paths. Although, similar to our approach, this work does not consider error recovery and user-specified approximate execution paths.

Rehman *et al.* [109] present a framework for reliable code generation and execution using reliability driven compilation. A compiler generates multiple, functionally equivalent, versions of a given function which differ in terms of vulnerability and execution time. Upon profiling the versions, the runtime system selects one that both increases the reliability of the system and meets the application's real-time constraints. Their work enforces functional correctness but does not exploit the algorithmic characteristic of significance. [118] introduces a system that selects a reliability robustness mechanism (Triple or Double Modular Redundancy, DMR/TMR) as well as the CPU operating voltage and frequency. Its goal is to minimize power consumption while achieving the reliability and timing requirements of the system. In our work, we do not seek functional correctness, but we offer a mechanism to exploit the algorithmic significance to allow errors to manifest only on non-significant computations.

[110] introduces the instruction vulnerability index (IVI) for software reliability estimation. Vulnerability indexes at the granularity of the function (FVI) and the application (AVI) is computed based on IVI. Given a user specified tolerable performance overhead constraint they perform compiler transformation to enhance code reliability. In our work we do not take into account the instruction vulnerability. We consider the algorithmic property of significance to steer application execution on reliable and unreliable cores. Relax [70] is an architectural framework that lets programmers turn off recovery mechanism as well as annotate regions of code for which hardware errors can occur. The hardware supports error detection and a C/C++ language-level recovery mechanism provides error recovery from hardware faults at different levels of code granularity.

Hardware support for error-tolerant and approximate computing spans designs to novel architectures. Razor [26] is a processor design which is based on dynamic detection and correction of timing failures of the critical paths due to below-nominal supply voltage. The key idea is to tune supply voltage by monitoring the error rate

during operation using shadow latches controlled by delayed clocks. The observation that the sequence of instructions in an application binary can have a significant impact on timing error rate is studied in [41]. A number of simple, yet effective code transformations that reduce error rate are introduced.

In [48] a hardware module monitors the processor pipeline, and checks for possible control flow violations (infinite loops). This module is used by the OS/compiler/application to detect errors and take corrective action. ERSA is a multi-core architecture where cores are either fully reliable or have relaxed reliability [73]. ERSA uses an explicit and application-specific mapping of code to cores with different levels of reliability. Our work follows a different approach, the programmer uses significance to indicate code with relaxed correctness semantics and the framework implements error detection and recovery, potentially approximating the task output.

EnerJ [121] proposes a programming model which explicitly declares data structures that may be subject to unreliable computation in return for increased performance or fault tolerance. EnerJ allows operations to be computed in aggressively voltage-scaled processors and data structures to be stored in DRAM with low refresh rate and SRAM with low supply voltage. Exposing such computing to the programmer requires expanding the processor ISA with instructions that offer only an expectation, rather than a guarantee that a certain operation will be performed correctly [27]. Contrary to our framework EnerJ specifies significance in the granularity of data and does not consider task-parallel execution, whereas we use as vehicle the granularity of a task. Furthermore, EnerJ does not explore the idea of error detection and correction, whereas we provide a systematic approach to using Artificial Neural Networks to automate the process of error detection.

There has also been a large amount of work which aims to solve the problem of efficient error detection. Current state of the art approaches to online error detection rely on duplicating the instructions of selected application parts which are considered error-prone. Unsafe instructions are first identified via compiler-analysis and/or profiling. Subsequently, a compiler pass hardens the application by duplicating the unsafe instructions and inserting checks [31, 81, 23, 71]. The checks typically involve redundancy in the form of instruction duplication. When a check detected an error it proceeds to restore an earlier checkpoint. IPAS [71] expects the user to include a verification function that is used to check whether an injected fault has propagated to the output of the code which is targeted for software-hardening against soft-errors. This function is only used to train an Artificial Neural Network to drives the selection of instructions prior to their duplication. Other works [39, 34, 56] rely on manually implemented Light-Weight Checks (LWCs) to detect errors at the outputs of computations. [34] use manual LWCs to determine when an approximate alternative to a function computes outputs which severely differs from the exact implementation. [56] relies on manually implemented LWCs to detect errors on the output of unreliably executed code. [39] falls back to instruction duplication whenever light-weight error detectors result in low *Detection Coverage*.

Two offline debugging mechanisms and three online monitoring mechanisms for approximate programs are presented in [113]. Among the offline mechanisms, the first one identifies correlation between QoR and each approximate operation by tracking the execution and error frequencies of different code regions over multiple program executions with varying QoR values. The second mechanism tracks which approximate operations affect any approximate variable and memory location. The online mechanisms complement the offline ones and they detect and compensate for QoR loss while maintaining the energy gains of approximation. The first mechanism compares the QoR for precise and approximate variants of the program for a random subset of executions. This mechanism is useful for programs where QoR can be assessed by sampling a few outputs, but not for those that require bounding the worst-case errors. The second mechanism uses programmer-supplied "verification functions" that can check a result with much lower overhead than computing the result. The third mechanism stores past inputs and outputs of the checked code and estimates the output for current execution based on interpolation of the previous executions with similar inputs. They show that their offline mechanisms help in effectively identifying the root of a quality issue instead of merely confirming the existence of an issue and the online mechanisms help in controlling QoR while maintaining high energy gains. Our method could also be applied to detect errors due to approximation but we chose to evaluate our automatic error detectors to check for errors at the output of code which executes under unreliable conditions.

[62] presents an output-quality monitoring and management technique which can ensure meeting a given output quality. Based on the observation that simple prediction approaches, e.g. linear estimation, moving average, and decision trees can accurately predict approximation errors, they use a low-overhead error detection module which tracks predicted errors to find the elements which need correction. Using this information, the recovery module, which runs in parallel to the detection module, re-executes the iterations that lead to high-errors. This becomes possible since the approximable functions or codes are generally those that simply read inputs and produce outputs without modifying any other state, such as map and stencil patterns. Our approach differs in that we use an ANN to detect error whereas [62] uses hardware accelerated ANNs to approximate code whose output is subsequently error checked. Large errors on the approximated computations are corrected by means of executing the accurate code using the CPU.

6.2.1 Power and Energy-Aware Optimization

Dynamic quality control of non-functional application properties including power and energy has been explored in HeartBeats [43], a framework for user-directed execution steering; PowerDial [44], an environment for adapting applications to execute efficiently under power and load fluctuations; Metronome [128], an operating system substrate for dynamic performance and power management; and the

Angstrom processor [45], which provides hardware support for monitoring performance, power, energy and temperature with user-controlled settings.

Dynamic power and energy optimization in the runtime system has been explored in several parallel programming models including OpenMP [21], message passing and hybrid models [76, 79], new parallel programming languages that natively support transparent adaptation to dynamic execution conditions such as [131] as well as distributed programming frameworks [53].

Cohen in Energy types [18] used a type-based system for expressing phases of computation which were then executed in different energy states, to optimize overall energy-efficiency. However, this system did not consider approximation or dynamic parallel execution as techniques for saving energy.

6.3 Voltage Margin Characterization and Prediction

Many research approaches have emerged in the last few years that relax conservative guardbands to improve energy efficiency. Prior work focusing on commercially available chips include [95, 4, 5, 94, 74, 151].

In contrast to our work, all previous efforts in predicting voltage margins using performance counters only report theoretical energy gains, without deploying their model. They rely only on theoretical results which, if not validated in real hardware, could result in unrealiable operation and even system crashes. Our FSM based governor uses performance counter values to reduce the voltage margin of the system at run time on real hardware for unseen workloads which consist of several different applications executing concurrently.

In particular in [95] authors present an automated system-level analysis on multi-core CPUs based on the ARMv8 64-bit architecture when pushed to operate in scaled voltage conditions. Due to the manifestation of SDCs before system crashes, the authors propose a severity function that can predict safe, SDC-free undervolt levels for each core of the processor. Based on this function and the corresponding core V_{min} resulted from the offline characterization, they produce a linear regression model that predicts the V_{min} of a core for a single-threaded workload. Their model is trained and evaluated using only single core executions. In contrast to their work, we use multi-threaded workloads and demonstrate their importance as multi-threaded workloads usually result to more conservative V_{min}. Moreover, [95] states that V_{min} prediction is uncorrelated to performance counters ($R^2 = 0$) and in combination with limited dynamic variation of V_{min} in ARMv8 a naive prediction of using average values is sufficient. In the x86-64 architecture we observe higher dynamic variations and our model is able to track these variations of the current workload. Note also that we identify margin deviations between off-the-shelf, commercially available parts, and not parts from extreme corner nodes (TTT, TFF and TSS in [95]). In a more recent paper [59], the same authors present a comprehensive

statistical analysis for the same platform that improves on prediction on dynamic variations.

In [94] the authors characterize the voltage margins of two x86-64 microprocessors (Sandy-Bridge-E and Haswell) for a subset of the SPEC CPU2006 benchmark suite. Similar to [95], they do not consider the implications induced by multi-core executions. Finally, in our work we also quantify the voltage margins deviations between off-the-shelf parts that belong to the same CPU family.

The heuristics presented in [4] and [5] that dynamically reduce voltage margins while always preserving safe operation, are based on the error correction ECC hardware built on modern processors such as the server-class Intel Itanium 9560. A key observation there is that as V_{dd} is lowered, ECC correctable errors appear before uncorrectable errors (SDCs and CPU crashes). The rate of ECC correctable errors is used as an indicator on how to adjust the V_{dd} voltage. In our work, errors reported by the ECC mechanism appear very rarely, and they are always accompanied by immediate CPU crashes. We do not rely on ECC, but rather predict a safe supply voltage using a selected set of performance counters as estimators. Eventually, the methodology we propose is generic enough that can be applied to any processor, that provides the ability to manipulate the supply voltage, by measuring performance counters.

Authors in [151] exploit the large margins available when only one core in a server-class 8-core Power7+ processor is utilized, turning under-utilized margin into power and performance benefits. Similar to our work, this paper finds that as more cores are progressively utilized by a multi-threaded applications causing larger IR drops across the chip, the benefits of an adaptive margin scheme begin to diminish.

A study of the voltage margins on several Kepler and Fermi GPUs is presented in [74]. They first characterize the impact of process, temperature and voltage variation on V_{min}, and then predict safe values of V_{min} by deploying a linear regression and a neural network model. They show that high energy margins can be achieved by shaving conservative guardbands in modern GPUs. Our work targets CPU architectures which are significantly more complex than GPUs. Moreover, typically CPUs - contrary to GPUs - serve volatile workloads, with diverse characteristics, consisting of mixed user and OS jobs. Our model is able to provide accurate voltage margins predictions for workloads which consist of a mixture of different benchmarks.

Based on microarchitectural events (such as branch mispredictions and cache misses) that flush and stall the CPU pipeline and cause large voltage droops, in [107] they propose a voltage emergency predictor that learns the signatures of such voltage emergencies and triggers a CPU throttling mechanism (e.g. increase voltage or decrease frequency) to prevent their recurrence. This work is based on CPU modeling on the Simplescalar simulator. In a subsequent paper, the authors measure run time voltage emergencies on a Core 2 Duo processor and attribute them to pipeline stalls and flushes [106]. Based on these experimental observations, they propose that a mechanism that uses more aggressive margins and a recovery (check-point based)

scheme may be better than a fail-safe static margin. Moreover, they propose a program scheduling mechanism so that the combined voltage droop is canceled out as much as possible. In the same spirit, [46, 86] identify and mitigate voltage emergencies introduced by multithreaded workloads by correlating performance counters and by introducing other thread synchronization mechanisms, respectively. Unlike our work, the previous papers do not resort to undervolting to reduce voltage margins but instead they describe techniques and provide solid design guidelines that could be exploited by the supplementary mechanism we describe in Section 4.7.

A number of proposed techniques present design approaches at the circuit or micro-architectural level that trade reliability for lower voltage, by attempting to reduce voltage down to the point that produces maximum allowable errors without causing catastrophic failures [57]. Several approaches propose methods which ensure correct operation of caches under undervolted conditions at the microarchitectural level [25, 17, 144]. Architectural techniques are presented to eliminate data corruption, and by extension enable cache operation at scaled voltage settings. In our work we are only interested on the behavior of the whole CPU, and not of any specific component. EVAL is a framework for dynamic adaptation of supply voltage, processor frequency and body bias using a machine learning algorithm [122]. Similar ideas include dynamic pipeline adaptation transferring the time slack of faster pipeline stages to the slower ones (ReCycle)[134], and using variable latency techniques to mitigate the impact of variations on the register file and execution units in a microprocessor [77].

Furthermore, there are many approaches that employ simulations and modeling techniques to provide design guidelines for future hardware. Authors in [64] propose a multi-core processor which can scale its resources and the number of operating cores to lower than that of integrated to meet certain power constrains. Several approaches [55, 54, 72] employ regression analysis to map certain performance counters to micro-architectural events and power-delivery estimation. [63] explores ECC protection mechanisms to enable low-power caches through a detailed SRAM failure modeling. [16] introduces a workload dependent technique to identify the paths excited by individual applications on ultra-low-power microprocessors and reduce voltage to a level that meets timing of those paths (instead of all paths).

6.4 Fault Injection Tools

In CLKSCREW[133], the voltage and frequency scaling capabilities of modern processors are exploited in order to compromise system security, by injecting faults during code execution and extracting cryptographic keys from the ARM TrustZone. This work focuses on the security risks raised by modern energy management techniques. RIFLE [82] and MESSALINE [3] introduce a deterministic and reproducible fault injection technique at the pin-level of a processor. FIAT [7] and FERRARI [60] implement a software-level fault injection which models complex systems with great

accuracy however, ensuring that the simulated models are realistic and restraining simulation time are significant challenges. VERIFY [126], MEFISTO [52] and GemFI [96] are fault injection simulators that provide high accuracy in both the location and the timing of the fault, but they introduce significant overhead. Finally, REFINE [32] allows fault injection in the back-end of the LLVM compiler.

In [15] a multi-faceted microarchitecture-level toolset for reliability assessment of modern microprocessors is presented. The framework is built around the Gem5 simulator and provides several modes of operation which employ acceleration features for all stages of a fault-injection based reliability assessment campaign. The same authors in [58] examine the effectiveness of microarchitectural fault injection for x86 and ARM microprocessors in a differential way: by developing and comparing two fault injection frameworks on top of the most popular performance simulators, MARSS and Gem5.

In our work we provided two tools. GemFI a simulation based fault injection tool which is based on a broadly used reconfigurable simulator (Gem5). The purpose of GemFI is to support any arbitrary fault model, by allowing the user to describe the faults to an input file. Moreover to the best of our knowledge GemFI is the first infrastructure that can target specific applications areas, while minimizing the changes to the original source code of the application under test. On the other hand XM^2 performs fault injection natively on the targeted system, therefore it provides native execution time and does not rely on any fault models as they manifest due to real hardware errors.

Chapter 7

Conclusions

In the first chapters of our work we discussed the significance based computing paradigm. Our main objective was to formulate the notion of computational significance of an application while providing flexible mechanisms to trade off quality for energy efficiency.

We encapsulate these objectives in two parameters *significance* and *ratio*. The *significance* defines the relative importance of a computation in respect to the quality of the output. On the other hand, the *ratio* parameter defines the percentage of computations which should be judged as significant and which should be not. These conceptual idea of *significance-ratio* are used into two different programming pardigms, the *fault tolerant* and the *approximate*.

We introduce and evaluate a framework that targets *fault tolerant* computing that enables execution on platforms operating unreliably outside their normal voltage / frequency envelope in order to aggressively reduce energy consumption. We present a programming model for the development of error resilient programs in a disciplined manner. Our model exploits programmer wisdom to characterize task significance, to provide checks and repair mechanisms to the outputs of tasks that are executed unreliably. We evaluate the effectiveness of different protection mechanisms. We show that traditional system software protection mechanisms are not adequate, however their combination with programmer wisdom provides effective protection against crashing and silent data corruptions, while enabling considerable energy gains. Interestingly, modern processors with the assistance of our framework can produce acceptable results until they reach the Point of First Failure. Decreasing the supply voltage below that point, either additional energy gains are too low, or massive failure rates defeat any software-based realistic protection mechanism.

A similar programming model that exposes the *significance-ratio* parameters is introduced for *approximate* computing. This information is used to achieve energy-constrained execution with graceful quality degradation. An offline, profile-based, training process produces a model which predicts the energy footprint of a given application as a function of its input size, the number of cores used, the processor frequency, and the ratio of accurate to total number of tasks. This model is exploited by the runtime system of an energy-constrained multi-core platform to steer execution towards a configuration that maximizes quality of output while complying with

energy constraints.

Significance based computing presents significant energy gains, however, it is not always applicable as it requires source code modifications and error handling mechanisms. Therefore we introduce an application agnostic modeling methodology, based on the monitoring of selected CPU performance metrics, to estimate and eliminate voltage margins without lowering they reliability of the system. Our offline characterization on CPUs from Intel Skylake and Haswell microarchitectures reveals wide voltage margins in both CPU families, and considerable margin variation across CPU parts, CPU cores, benchmarks and workload configurations, thus motivating the need for customized and dynamic voltage adjustment to significantly improve energy efficiency. We evaluate our model effectiveness by using a voltage governor. The governor relies on our prediction model to continuously adjust the supply voltage and drive the processor to a more energy efficient, yet safe zone of operation. The experimental evaluation across a wide range of benchmarks and larger-scale applications shows large energy savings up to 42.68% and 34.37% over the standard Intel P-state DVFS governor for parts of the Skylake and Haswell families, respectively.

Finally, we present two experimental frameworks for reliability analysis to study the effects of sub-nominal CPU operation. *GemFI* is simulation based fault injection and enables fault injection of transient, intermittent and permanent faults. It simulates unreliable environments in full system, cycle accurate mode. It is not limited to specific fault models, but is easily extensible and facilitates support of future fault models. Moreover, GemFI features such as checkpointing allow the execution of large-scale fault injection campaigns. XM^2 is a software framework which facilitates the experimental evaluation of the effects of voltage/frequency margins on the operation of CPU platforms. XM^2 can be used to study the behavior of software running on platforms operating – potentially unreliably – outside their nominal operational envelope.

7.1 Future Work

There are several open research opportunities based on our work. An interesting research direction is the relationship of significance, task granularity, quality of results, and overhead of error detection & correction. This is a problem similar to choosing the appropriate granularity for parallel code to optimize the resulting performance. It is important that application developers can make educated decisions regarding the granularity of their application tasks. Otherwise, the resulting implementations might under-utilize the hardware or even produce output of lesser quality.

Another interesting research direction is to observe the interactions between the voltage margins of modern processors and the operating frequency of the system. Using a similar modeling methodology one can express the margins of the system as a function of the performance metrics and the operating frequency. Using such
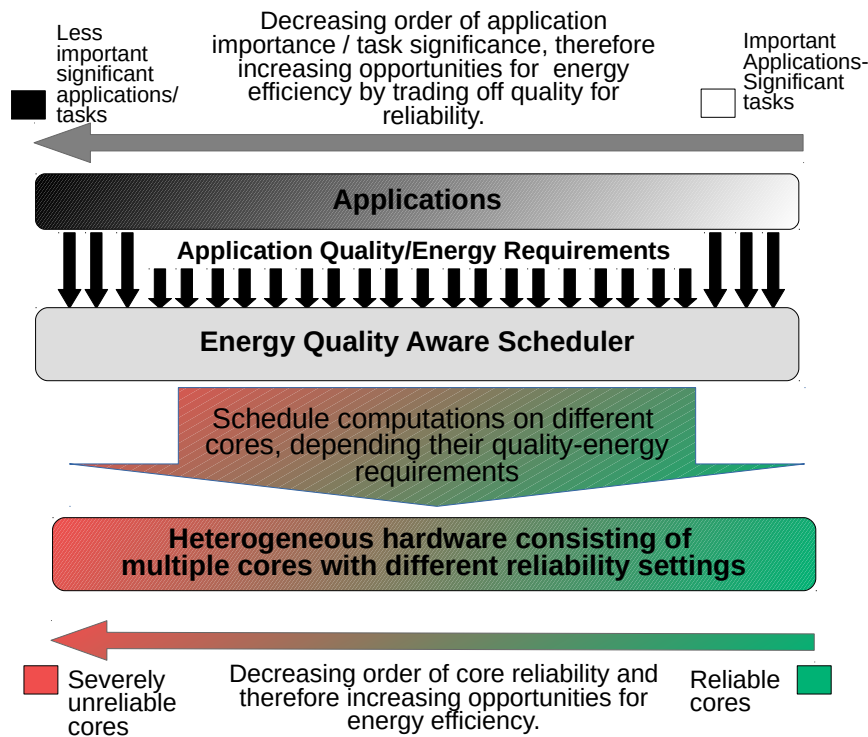
FIGURE 7.1: Vision of our approach. The applications and the computations should provide information to the software stack about their quality/energy requirements. Using this information the computations can be scheduled in hardware with different energy-reliability settings.

a model one can create a power capping governor which will select the voltage-frequency setting of a processor depending on a user defined power budget.

Moreover, we are also plan to study the effects of computation approximations and the dynamic margins of the application. We have shown that the voltage margins of both X86 and ARM processors are workload dependent. For example the approximate versions of the code could present wider voltage or frequency margins, hence allowing our governor to set the operating point of the system to lower values. Such a method would increase the energy efficiency of the system even more, as the approximate version by design consume less energy and these computation can be executed in a more energy efficient processor operating point.

This thesis concludes with the following suggestion towards the realization of energy efficient computing on next generation hardware depicted in figure 7.1. The transition to subnominal operation requires effort from the side of the software stack, to design systems which present clear quality requirements. Of course not all parts of an application are amenable to *errors*. There are critical regions of software which would significantly deter the reliability of the system if an erroneous event corrupted their execution flow. Such sensitive software inclure for example such software Operating Systems, runtime systems etc. Fortunately, there are classes of applications, as presented in this thesis, in which the majority of computations present error resiliency characteristics, such applications can be executed in deep unreliable hardware setting. The remaining of the software can benefit from execution on hardware

that operates in the *safe* region.

Related publications

[1] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, Nikolaos Bellas GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2014

[2] Konstantinos Parasyris, Nikolaos Bellas, Christos D. Antonopoulos, and Spyros Lalis Exploring the Effects of Code Optimizations on CPU Frequency Margins, ATCET 2018 In *Proceedings of the 1st Workshop on Approximate and Transprecision Computing on Emerging Technologies*, ATCET 2018

[3] Konstantinos Parasyris, Nikolaos Bellas, Christos D. Antonopoulos, and Spyros Lalis A Framework for Evaluating Software on Reduced Margins Hardware In*Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2018

[4] Konstantinos Parasyris, Vassilis Vassiliadis, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas. Significance-Aware Program Execution on Unreliable Hardware. *ACM Transactions on Architecture and Code Optimization*, TACO 2017, 14(2):12:1–12:25

[5] Panos Koutsovasilis, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas and Spyros Lalis Dynamic Undervolting to Improve Energy Efficiency on Multicore X86 CPUs. *(Under review)*

[6] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. Towards automatic significance analysis for approximate computing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016

[7] Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Exploiting significance of computations and profile-driven regression for energy-constrained approximate computing. *International Journal of Parallel Programming*, IJPP 2016, 44(5):1078–1098

[8] Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. A Significance-driven Programming Framework for Energy-constrained Approximate Computing In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF 2015

[9] Vassilis Vassiliadis, Konstantinos Parasyris, Charalampos Chalios, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. A programming model and runtime system for significance-aware energy-efficient computing. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 50, pages 275–276. PPOPP 2015 (poster abstract)

Contribution to Joint Publications

The results presented in this Thesis have been partially published in [68, 97, 99, 141, 140, 139, 138, 96]. This appendix discusses my contribution to each of the aforementioned publications.

In [99] I contributed to the design of the significance aware fault tolerant programming model. I contributed the runtime system, the power/time/energy/fault model, and the hybrid software-simulation based fault injection methodology. Additionally, I contributed to the process of benchmarking, as well as the analysis of the results. These extensions facilitate the development of applications using the significance aware programming model as well as the implementation of the result check function.

For [96] I contributed to the design of the fault injection tool, the checkpointing methodology as well as the deployment of the tool on network of workstations. Additionally I contributed to the validation and the experimentation of the tool.

For [97] I contributed to the design monitoring and deployment of the fault injection tool. Additionally, I integrated the functionality of the tool with Circle [115]. Finally, I contributed the case studies regarding the *ARM Cortex A53* processor.

In [98] I contributed the study about the correlation of compiler and source code optimization to the extend of the frequency margins of the *ARM Cortex A53* processor.

For [141, 140, 139, 138], I performed benchmarking, and analysed the results of the experimental campaigns. I also contributed to the design of the approximation extensions for our significance aware computing programming model.

Finally, for [68] I contributed the entire modeling methodology. Additionally, I contributed the synchronization library used by the applications in the offline characterization. The process of benchmarking of the *offline* characterization and the *online* evaluation was jointly performed by me and Panos Koutsovasilis.

Bibliography

[1] Sara Achour and Martin C. Rinard. "Approximate Computation with Outlier Detection in Topaz". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 711–730. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814314. URL: http://doi.acm.org/10.1145/2814270.2814314.

[2] J Ansel, K Arya, and G Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop Proc". In: *Proc. of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2009.

[3] Jean Arlat et al. "Fault Injection for Dependability Validation: A Methodology and Some Applications". In: *IEEE Trans. on Software Engineering* (1990).

[4] A. Bacha and R. Teodorescu. "Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014. DOI: 10.1109/MICRO.2014.54.

[5] Anys Bacha and Radu Teodorescu. "Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors". In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 297–307. ISSN: 0163-5964. DOI: 10.1145/2508148.2485948. URL: http://doi.acm.org/10.1145/2508148.2485948.

[6] Woongki Baek and Trishul M. Chilimbi. "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. New York, NY, USA: ACM, 2010, pp. 198–209. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806620. URL: http://doi.acm.org/10.1145/1806596.1806620.

[7] J. H. Barton et al. "Fault Injection Experiments Using FIAT". In: *IEEE Trans. on Computers* (1990).

[8] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.

[9] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*. ACM. 2008, pp. 72–81.

[10] Nathan Binkert et al. "The Gem5 Simulator". In: *ACM SIGARCH Computer Architecture News* (2011).

[11] David Blaauw et al. "Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance". In: *IEEE Int. Solid-State Circuits Conference, ISSCC , Digest of Technical Papers*. 2008.

[12] OpenMP Architecture Review Board. *OpenMP Application Program Interface (version 4.0)*. Tech. rep. 2013.

[13] Keith A Bowman et al. "A 45 nm resilient microprocessor core for dynamic variation tolerance". In: *IEEE Journal of Solid-State Circuits* 46.1 (2011), pp. 194–208.

[14] Edward A Burton et al. "FIVR—Fully Integrated Voltage Regulators on 4th Generation Intel® Core ™ SoCs". In: *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*. IEEE. 2014, pp. 432–439.

[15] A. Chatzidimitriou and D. Gizopoulos. "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 69–78. DOI: 10.1109/ISPASS.2016.7482075.

[16] H. Cherupalli, R. Kumar, and J. Sartori. "Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 671–681. DOI: 10.1109/ISCA.2016.64.

[17] Zeshan Chishti et al. "Improving Cache Lifetime Reliability at Ultra-low Voltages". In: *In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 89–99. ISBN: 978-1-60558-798-1.

[18] Michael Cohen et al. "Energy Types". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 831–850. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384676. URL: http://doi.acm.org/10.1145/2384616.2384676.

[19] Jeremy Constantin et al. "Exploiting Dynamic Timing Margins in Microprocessors for Frequency-over-scaling with Instruction-based Clock Adjustment". In: *Proc. of the Design, Automation & Test in Europe Conference & Exhibition*. 2015.

[20] C. Constantinescu. "Trends and challenges in VLSI circuit reliability". In: *IEEE Micro* 23.4 (2003), pp. 14–19. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1225959.

[21] Matthew Curtis-Maury et al. "Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. New York, NY, USA: ACM, 2008, pp. 250–259. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454151. URL: http://doi.acm.org/10.1145/1454115.1454151.

[22] Shidhartha Das et al. "A self-tuning DVS processor using delay-error detection and correction". In: *Solid-State Circuits, IEEE Journal of* 41.4 (2006).

[23] Moslem Didehban and Aviral Shrivastava. "nZDC: A Compiler technique for near Zero Silent data Corruption". In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM. 2016, p. 48.

[24] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering". In: *Statistics and computing* 10.3 (2000).

[25] H. Duwe et al. "Rescuing Uncorrectable Fault Patterns in On-Chip Memories through Error Pattern Transformation". In: *In Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 634–644. DOI: 10.1109/ISCA.2016.61.

[26] Dan Ernst et al. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation". In: *Proc. of the 36th Annual IEEE/ACM Int. Symposium on Microarchitecture*. 2003.

[27] Hadi Esmaeilzadeh et al. "Architecture Support for Disciplined Approximate Programming". In: *Proc. of the Seventeenth Int. Conference on Architectural Support for Programming Languages and Operating Systems*. 2012.

[28] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376.

[29] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 449–460. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.48. URL: http://dx.doi.org/10.1109/MICRO.2012.48.

[30] Liang Fan, Siwei Ma, and Feng Wu. "Overview of AVS video standard". In: *Proc. of the IEEE International Conference on Multimedia and Expo (ICME)*. 2004.

[31] Shuguang Feng et al. "Shoestring: probabilistic soft error reliability on the cheap". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 1. ACM. 2010, pp. 385–396.

[32] Giorgis Georgakoudis et al. "REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed". In: *Proc. of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*. 2017.

[33] Íñigo Goiri et al. "ApproxHadoop: Bringing Approximations to MapReduce Frameworks". In: *Proc. of the 22th Int. Conference on Architectural Support for Programming Languages and Operating Systems*. 2015.

[34] Beayna Grigorian and Glenn Reinman. "Dynamically adaptive and reliable approximate computing using light-weight error analysis". In: *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*. IEEE. 2014, pp. 248–255.

[35] Philipp Gschwandtner et al. "On the potential of significance-driven execution for energy-aware HPC". English. In: *Computer Science - Research and Development* (2014), pp. 1–10. ISSN: 1865-2034. DOI: 10.1007/s00450-014-0265-9. URL: http://dx.doi.org/10.1007/s00450-014-0265-9.

[36] Meeta S. Gupta et al. "Tribeca: Design for PVT Variations with Local Recovery and Fine-grained Adaptation". In: *Proc. of the 42Nd Annual IEEE/ACM Int. Symposium on Microarchitecture*. 2009.

[37] D. Hackenberg et al. "Introducing FIRESTARTER: A Processor Stress Test Utility". In: *In Proceedings in the 4th Conference on International Green Computing (IGC)*. 2013, pp. 1–9. DOI: 10.1109/IGCC.2013.6604507.

[38] Nikos Hardavellas et al. "Toward dark silicon in servers". In: *IEEE Micro* 31.4 (2011), pp. 6–15.

[39] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. "Low-cost program-level detectors for reducing silent data corruptions". In: *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE. 2012, pp. 1–12.

[40] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[41] Giang Hoang, Robby Bruce Findler, and Russ Joseph. "Exploring Circuit Timing-aware Language and Compilation". In: *Proc. of the 16th Int. Conference on Architectural Support for Programming Languages and Operating Systems*. 2011.

[42] Arthur E. Hoerl and Robert W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 55–67. DOI: 10.1080/00401706.1970.10488634. eprint: http://amstat.tandfonline.com/doi/pdf/10.1080/00401706.1970.10488634. URL: http://amstat.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634.

[43] Henry Hoffmann et al. "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments". In: *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88. ISBN: 978-1-4503-0074-2. DOI: 10.1145/1809049.1809065. URL: http://doi.acm.org/10.1145/1809049.1809065.

[44] Henry Hoffmann et al. "Dynamic Knobs for Responsive Power-aware Computing". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 199–212. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950390. URL: http://doi.acm.org/10.1145/1950365.1950390.

[45] Henry Hoffmann et al. "Self-aware Computing in the Angstrom Processor". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, pp. 259–264. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228409. URL: http://doi.acm.org/10.1145/2228360.2228409.

[46] X. Hu et al. "Orchestrator: A low-cost Solution to Reduce Voltage Emergencies for Multi-threaded Applications". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 208–213. DOI: 10.7873/DATE.2013.056.

[47] SI ITRS. "International technology roadmap for semiconductors: Executive summary". In: *Semiconductor Industry Association, Tech. Rep* (2013).

[48] Ravishankar K Iyer et al. "Recent Advances and New Avenues in Hardware-Level Reliability Support". In: *IEEE Micro* 25.6 (2005).

[49] Antoine Petitet Jack J. Dongarra Piotr Luszczek. "The LINPACK Benchmark: Past, Present and Future". In: *Concurrency and Computation Practice and Experience* 10 (9 2003).

[50] Norman James et al. "Comparison of split-versus connected-core supplies in the POWER6 microprocessor". In: *2007 IEEE Int. Solid-State Circuits Conference. Digest of Technical Papers*. 2007.

[51] ArtForz. Jeff Garzik. *CPU-Miner*. URL: https://github.com/tpruvot/cpuminer-multi.

[52] Eric Jenn et al. "Fault Injection into VHDL Models: The MEFISTO Tool". In: *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*. 1994.

[53] Myeongjae Jeon et al. "Adaptive Parallelism for Web Search". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 155–168. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465367. URL: http://doi.acm.org/10.1145/2465351.2465367.

[54] W. Jia, K. A. Shaw, and M. Martonosi. "Stargazer: Automated Regression-Based GPU Design Space Exploration". In: *In Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*. 2012, pp. 2–13. DOI: 10.1109/ISPASS.2012.6189201.

[55] P. J. Joseph, Kapil Vaswani, and M. J. Thazhuthaveetil. "Construction and Use of Linear Regression Models for Processor Performance Analysis". In: *In Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*. 2006, pp. 99–108. DOI: 10.1109/HPCA.2006.1598116.

[56] Edin Kadric, Kunal Mahajan, and André DeHon. "Energy reduction through differential reliability and lightweight checking". In: *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE. 2014, pp. 243–250.

[57] Andrew B. Kahng et al. "Designing a Processor from the Ground up to Allow Voltage/Reliability Tradeoffs". In: *In Proceedings of the 16th International Conference on High-Performance Computer Architecture (HPCA)*. 2010, pp. 1–11.

[58] M. Kaliorakis et al. "Differential Fault Injection on Microarchitectural Simulators". In: *2015 IEEE International Symposium on Workload Characterization*. 2015, pp. 172–182. DOI: 10.1109/IISWC.2015.28.

[59] M. Kaliorakis et al. "Statistical Analysis of Multicore CPUs Operation in Scaled Voltage Conditions". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 109–112. ISSN: 1556-6056. DOI: 10.1109/LCA.2018.2798604.

[60] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A Flexible Software-Based Fault and Error Injection System". In: *IEEE Trans. Comput.* 44 (1995).

[61] Georgios Karakonstantis et al. "Significance driven computation on next-generation unreliable platforms". In: *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*. IEEE. 2011, pp. 290–291.

[62] Daya S. Khudia et al. "Rumba: An Online Quality Management System for Approximate Computing". In: *SIGARCH Comput. Archit. News* 43.3 (2015), pp. 554–566.

[63] Jangwoo Kim et al. "Modeling SRAM Failure Rates to enable Fast, Dense, Low-Power Caches". In: *SELSE'09* (2009).

[64] N.S. Kim. *Resource and Core Scaling for Improving Performance of Power-Constrained Multicore Processors*. US Patent 9,606,842. 2017. URL: https://www.google.com/patents/US9606842.

[65] Y. Kim et al. "AUDIT: Stress Testing the Automatic Way". In: *In Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2012, pp. 212–223.

[66] Youngtaek Kim and Lizy Kurian John. "Automated Di/Dt Stressmark Generation for Microprocessor Power Delivery Networks". In: *In Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED)*. 2011, pp. 253–258. ISBN: 978-1-61284-660-6.

[67] Jonathan Koomey et al. "Implications of historical trends in the electrical efficiency of computing". In: *IEEE Annals of the History of Computing* 33.3 (2011), pp. 46–54.

[68] Panos Koutsovasilis et al. "Dynamic Undervolting to Improve Energy Efficiency on Multicore X86 CPUs". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2018).

[69] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. "Estimating Mutual Information". In: *Phys. Rev. E* 69 (6 2004), p. 066138. DOI: 10.1103/PhysRevE.69.066138.

[70] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. "Relax: An Architectural Framework for Software Recovery of Hardware Faults". In: *Proc. of the 37th Int. Symposium on Computer Architecture*. 2010.

[71] Ignacio Laguna et al. "Ipas: Intelligent protection against silent output corruption in scientific applications". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM. 2016, pp. 227–238.

[72] Benjamin C. Lee and David M. Brooks. "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction". In: *SIGPLAN Not.* 41.11 (Oct. 2006), pp. 185–194. ISSN: 0362-1340. DOI: 10.1145/1168918.1168881. URL: http://doi.acm.org/10.1145/1168918.1168881.

[73] Larkhoon Leem et al. "ERSA: Error Resilient System Architecture for Probabilistic Applications". In: *Proc. of the Conference on Design, Automation and Test in Europe*. 2010.

[74] J. Leng et al. "Safe limits on Voltage Reduction Efficiency in GPUs: A Direct Measurement Approach". In: *2015 48th Annual International Symposium on Microarchitecture (MICRO)*. 2015. DOI: 10.1145/2830772.2830811.

[75] Régis Leveugle et al. "Statistical fault injection: quantified error and confidence". In: *Design, Automation & Test in Europe Conference & Exhibition, 2009*. 2009.

[76] Dong Li et al. "Strategies for Energy-Efficient Resource Management of Hybrid Programming Models". In: *IEEE Trans. Parallel Distrib. Syst.* 24.1 (Jan. 2013), pp. 144–157. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.95. URL: http://dx.doi.org/10.1109/TPDS.2012.95.

[77] Xiaoyao Liang and David M. Brooks. "Mitigating the Impact of Process Variations on Processor Register Files and Execution Units". In: *In Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*. 2006.

[78] A. Liaw and M. Wiener. "Classification and Regression by RandomForest". In: *R news* 2.3 (2002), pp. 18–22.

[79] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. "Adaptive, Transparent CPU Scaling Algorithms Leveraging Inter-node MPI Communication Regions". In: *Parallel Comput.* 37.10-11 (Oct. 2011), pp. 667–683. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.07.001. URL: http://dx.doi.org/10.1016/j.parco.2011.07.001.

[80] *Linux Kernel.* 2017. URL: https://www.kernel.org/.

[81] Qining Lu et al. "SDCTune: a model for predicting the SDC proneness of an application for configurable protection". In: *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on.* IEEE. 2014, pp. 1–10.

[82] Henrique Madeira et al. "RIFLE: A general purpose pin-level fault injector". In: *Proc. of the European Dependable Computing Conference (EDCC).* 1994.

[83] Jackson Marusarz, Shannon Cepeda, and Ahmad Yasin. *How to Tune Applications Using a Top-Down Characterization of Microarchitectural Issues.* Tech. rep. Technical report, Intel, 2013.

[84] Abdelhafid Mazouz et al. "Evaluation of CPU Frequency Transition Latency". In: *Comput. Sci.* 29.3-4 (2014).

[85] Lawrence McAfee and Kunle Olukotun. "EMEURO: A Framework for Generating Multi-purpose Accelerators via Deep Learning". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 125–135. ISBN: 978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738616.

[86] T. N. Miller et al. "VRSync: Characterizing and Eliminating Synchronization-induced Voltage Emergencies in Many-core Processors". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA).* 2012, pp. 249–260. DOI: 10.1109/ISCA.2012.6237022.

[87] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. "Parallelizing Sequential Programs with Statistical Accuracy Tests". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (2013), 88:1–88:26. ISSN: 1539-9087. DOI: 10.1145/2465787.2465790. URL: http://doi.acm.org/10.1145/2465787.2465790.

[88] Sasa Misailovic et al. "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels". In: *SIGPLAN Not.* 49.10 (2014), pp. 309–328. ISSN: 0362-1340. DOI: 10.1145/2714064.2660231. URL: http://doi.acm.org/10.1145/2714064.2660231.

[89] Sparsh Mittal. "A survey of techniques for approximate computing". In: *ACM Computing Surveys (CSUR)* 48.4 (2016), p. 62.

[90] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. "Significance Driven Computation: A Voltage-scalable, Variation-aware, Quality-tuning Motion Estimator". In: *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '09. San Fancisco, CA, USA: ACM, 2009, pp. 195–200. ISBN: 978-1-60558-684-7. DOI: 10.1145/1594233.1594282. URL: http://doi.acm.org/10.1145/1594233.1594282.

[91] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[92] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. 1st ed. Society for Industrial and Applied Mathematics, 2009. ISBN: 9780898716696. URL: http://amazon.com/o/ASIN/0898716691.

[93] U. Mukhopadhyay et al. "A Brief Survey of Cryptocurrency Systems". In: *In Proceedings of 14th Annual Conference on Privacy, Security and Trust (PST)*. 2016, pp. 745–752. DOI: 10.1109/PST.2016.7906988.

[94] G. Papadimitriou et al. "Voltage Margins Identification on Commercial x86-64 Multicore Microprocessors". In: *2017 IEEE 23rd Int, Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2017.

[95] George Papadimitriou et al. "Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs". In: (2017).

[96] K. Parasyris et al. "GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates". In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP Int. Conference on*. 2014.

[97] Konstantinos Parasyris et al. "A Framework for Evaluating Software on Reduced Margins Hardware". In: *Proceedings of the 2018 International Conference on Dependable Systems and Networks (DSN2018)*.

[98] Konstantinos Parasyris et al. "Exploring the Effects of Code Optimizations on CPU Frequency Margins". In: *Proceedings of the 1st Workshop on Approximate and Transprecision Computing on Emerging Technologies, ATCET*. 2018.

[99] Konstantinos Parasyris et al. "Significance-Aware Program Execution on Unreliable Hardware". In: *ACM Trans. Archit. Code Optim.* 14.2 (2017), 12:1–12:25. ISSN: 1544-3566. DOI: 10.1145/3058980. URL: http://doi.acm.org/10.1145/3058980.

[100] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2078195.

[101] *Perf: Linux Profiling with Performance Counters*. 2017. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[102] L-N Pouchet. *PolyBench/C 3.2*. URL: https://sourceforge.linuxKernelnet/projects/polybench/.

[103] Abbas Rahimi, Luca Benini, and Rajesh K Gupta. "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. 2012.

[104] Abbas Rahimi et al. "A Variability-aware OpenMP Environment for Efficient Execution of Accuracy-configurable Computation on shared-FPU Processor Clusters". In: *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '13. Piscataway, NJ, USA: IEEE Press, 2013, 35:1–35:10. ISBN: 978-1-4799-1417-3. URL: http://dl.acm.org/citation.cfm?id=2555692.2555727.

[105] Abbas Rahimi et al. "Variation-tolerant OpenMP Tasking on Tightly-coupled Processor Clusters". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 541–546. ISBN: 978-1-4503-2153-2. URL: http://dl.acm.org/citation.cfm?id=2485288.2485422.

[106] V. J. Reddi et al. "Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling". In: *In Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2010, pp. 77–88. DOI: 10.1109/MICRO.2010.35.

[107] Vijay Janapa Reddi et al. "Voltage Emergency Prediction: Using Signatures to Reduce Operating Margins". In: *In Proceedings of the 15th International Symposium on High Performance Computer Architecture, (HPCA)*. 2009, pp. 18–29.

[108] Vijay Janapa Reddi et al. "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling". In: *2010 43rd Annual IEEE/ACM Int. Symposium on Microarchitecture*. 2010.

[109] Semeen Rehman et al. "Cross-layer software dependability on unreliable hardware". In: *IEEE Trans. on Computers* (2016).

[110] Semeen Rehman et al. "Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability". In: *Proc. of the 7th IEEE/ACM/IFIP Int. Conference on Hardware/Software Codesign and System Synthesis*. 2011.

[111] George A. Reis et al. "SWIFT: Software Implemented Fault Tolerance". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–254. ISBN: 0-7695-2298-X. DOI: 10.1109/CGO.2005.34. URL: http://dx.doi.org/10.1109/CGO.2005.34.

[112] Martin Rinard. "Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks". In: *ICS '06*. ACM, 2006, pp. 324–334.

[113] Michael Ringenburg et al. "Monitoring and Debugging the Quality of Results in Approximate Programs". In: *ASPLOS '15*. ACM, 2015, pp. 399–411.

[114] Pooja Roy et al. "ASAC: Automatic Sensitivity Analysis for Approximate Computing". In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '14. New York, NY, USA: ACM, 2014, pp. 95–104. ISBN: 978-1-4503-2877-7. DOI: 10.1145/2597809.2597812. URL: http://doi.acm.org/10.1145/2597809.2597812.

[115] rsta2. *Circle: A C++ bare metal programming environment for the Raspberry PI*. URL: https://github.com/rsta2/circle.

[116] S. T. Kim and Y. C. Shih and K. Mazumdar and R. Jain and J. F. Ryan and C. Tokunaga and C. Augustine and J. P. Kulkarni and K. Ravichandran and J. W. Tschanz and M. M. Khellah and V. De. "Enabling Wide Autonomous DVFS in a 22 nm Graphics Execution Core Using a Digitally Controlled Fully Integrated Voltage Regulator". In: *IEEE Journal of Solid-State Circuits* 51.1 (2016), pp. 18–30. ISSN: 0018-9200. DOI: 10.1109/JSSC.2015.2457920.

[117] Mastooreh Salajegheh et al. "Half-Wits: Software Techniques for Low-Voltage Probabilistic Storage on Microcontrollers with NOR Flash Memory". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013), 91:1–91:25. ISSN: 1539-9087. DOI: 10.1145/2465787.2465793. URL: http://doi.acm.org/10.1145/2465787.2465793.

[118] Mohammad Salehi et al. "DRVS: Power-efficient reliability management through Dynamic Redundancy and Voltage Scaling under variations". In: *Proc. of the IEEE/ACM Int. Symposium on Low Power Electronics and Design*. 2015.

[119] Mehrzad Samadi et al. "Paraprox: Pattern-based Approximation for Data Parallel Applications". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 35–50. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541948. URL: http://doi.acm.org/10.1145/2541940.2541948.

[120] Mehrzad Samadi et al. "SAGE: Self-tuning Approximation for Graphics Engines". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540711. URL: http://doi.acm.org/10.1145/2540708.2540711.

[121] Adrian Sampson et al. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: ACM, 2011, pp. 164–174. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993518. URL: http://doi.acm.org/10.1145/1993498.1993518.

[122] Smruti R. Sarangi et al. "EVAL: Utilizing Processors with Variation-induced Timing Errors". In: *In Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*. 2008, pp. 423–434.

[123] Florian Schmoll et al. "Improving the Fault Resilience of an H.264 Decoder Using Static Analysis Methods". In: *ACM Trans. Embed. Comput. Syst.* 13.1s (2013), 31:1–31:27. ISSN: 1539-9087. DOI: 10.1145/2536747.2536753. URL: http://doi.acm.org/10.1145/2536747.2536753.

[124] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[125] Stelios Sidiroglou-Douskos et al. "Managing Performance vs. Accuracy Trade-offs with Loop Perforation". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. ACM, 2011, pp. 124–134. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025133. URL: http://doi.acm.org/10.1145/2025113.2025133.

[126] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions". In: *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*. 1997.

[127] Teja Singh et al. "Zen: An Energy-Efficient High-Performance \×86 Core". In: *IEEE Journal of Solid-State Circuits* 53.1 (2018), pp. 102–114.

[128] Filippo Sironi et al. "Metronome: Operating System Level Performance Management via Self-adaptive Computing". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, pp. 856–865. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228514. URL: http://doi.acm.org/10.1145/2228360.2228514.

[129] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. "The JPEG 2000 still Image Compression Standard". In: *Signal Processing Magazine, IEEE* 18.5 (Sept. 2001), pp. 36–58. DOI: 10.1109/79.952804. URL: http://doi.org/10.1109/79.952804.

[130] Joseph Sloan, John Sartori, and Rakesh Kumar. "On Software Design for Stochastic Processors". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, pp. 918–923. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228524. URL: http://doi.acm.org/10.1145/2228360.2228524.

[131] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. "Adaptive, Efficient, Parallel Execution of Parallel Programs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. New York, NY, USA: ACM, 2014, pp. 169–180. ISBN: 978-1-4503-2784-8.

DOI: `10.1145/2594291.2594292`. URL: `http://doi.acm.org/10.1145/2594291.2594292`.

[132] *Stress-NG*. URL: `http://kernel.ubuntu.com/~cking/stress-ng/`.

[133] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *In Proceedings of 26th Security Symposium (USENIX Security)*. 2017.

[134] Abhishek Tiwari, Smruti R. Sarangi, and Josep Torrellas. "ReCycle: Pipeline Adaptation to Tolerate Process Variation". In: *In Proceedings of the 34th International Symposium on Computer Architecture, (ISCA)*. 2007, pp. 323–334.

[135] Jan Treibig, Georg Hager, and Gerhard Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *Proc. of the 2010 39th Int. Conference on Parallel Processing Workshops*. 2010.

[136] George Tzenakis et al. "BDDT:: Block-level Dynamic Dependence Analysis for Deterministic Task-based Parallelism". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 301–302. ISBN: 978-1-4503-1160-1. DOI: `10.1145/2145816.2145864`. URL: `http://doi.acm.org/10.1145/2145816.2145864`.

[137] G. Tziantzioulis et al. "b-HiVE: A Bit-level History-based Error Model with Value Correlation for Voltage-scaled Integer and Floating Point Units". In: *Proc. of the 52Nd Annual Design Automation Conference*. 2015.

[138] Vassilis Vassiliadis et al. "A programming model and runtime system for significance-aware energy-efficient computing". In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM. 2015, pp. 275–276.

[139] Vassilis Vassiliadis et al. "A significance-driven programming framework for energy-constrained approximate computing". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM. 2015, p. 9.

[140] Vassilis Vassiliadis et al. "Exploiting significance of computations and profile-driven regression for energy-constrained approximate computing". In: *International Journal of Parallel Programming* 44.5 (2016), pp. 1078–1098.

[141] Vassilis Vassiliadis et al. "Towards Automatic Significance Analysis for Approximate Computing". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO 2016. New York, NY, USA: ACM, 2016, pp. 182–193. ISBN: 978-1-4503-3778-6. DOI: `10.1145/2854038.2854058`. URL: `http://doi.acm.org/10.1145/2854038.2854058`.

[142] Swagath Venkataramani et al. "Quality Programmable Vector Processors for Approximate Computing". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-2638-4. DOI: `10.1145/2540708.2540710`. URL: `http://doi.acm.org/10.1145/2540708.2540710`.

[143]   Stephen T Welstead. *Fractal and wavelet image compression techniques*. SPIE Optical Engineering Press, 1999. ISBN: 9780819435033.

[144]   C. Wilkerson et al. "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation". In: *In Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*. 2008, pp. 203–214. DOI: 10.1109/ISCA.2008. 22.

[145]   G. Woltman and S Kurowski. *GIMPS, The Great Internet Mersenne Prime Search*. 2008. URL: https://www.mersenne.org/.

[146]   Amir Yazdanbakhsh et al. "AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing". In: *IEEE Design & Test* (2016).

[147]   Charles R Yount and Daniel P Siewiorek. "A methodology for the rapid injection of transient hardware errors". In: *IEEE Trans. on Computers* (1996).

[148]   Foivos S. Zakkak et al. *Inference and Declaration of Independence: Impact on Deterministic Task Parallelism*. New York, NY, USA, 2012. DOI: 10.1145/ 2370816.2370892. URL: http://doi.acm.org/10.1145/2370816. 2370892.

[149]   Qian Zhang et al. "ApproxIt: An Approximate Computing Framework for Iterative Methods". In: *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*. DAC '14. New York, NY, USA: ACM, 2014, 97:1–97:6. ISBN: 978-1-4503-2730-5. DOI: 10.1145/2593069. 2593092. URL: http://doi.acm.org/10.1145/2593069.2593092.

[150]   Zeyuan Allen Zhu et al. "Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: ACM, 2012, pp. 441–454. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103710. URL: http://doi.acm. org/10.1145/2103656.2103710.

[151]   Y. Zu et al. "Adaptive Guardband Scheduling to Improve System-Level Efficiency of the POWER7". In: *In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 308–321.