UNIVERSITY OF THESSALY

PHD THESIS

Optimization of program execution using computational significance

Author:
Vassilis VASSILIADIS

Supervisor:
Christos D. ANTONOPOULOS

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

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

in the

Department of Electrical and Computer Engineering

November 30, 2017

"What drives magic is, at the end of the day, sheer will."

"Harry Dresden" in the novel "Cold days" by Jim Butcher

Abstract

Vassilis VASSILIADIS

Optimization of program execution using computational significance

To this day, advancements in terms of performance and power efficiency for computing systems are primarily the result of transistor shrinking, frequency scaling, and parallelism exploitation. However, according to the 2013 report of the International Technology Roadmap for Semiconductors, this trend is predicted to come to a halt beyond 2020. Past practices are bound to hit the so-called power-wall of CMOS technology. Power consumption is a limiting factor and a key concern for the future of computing systems.

Heterogeneous architectures are a step towards realizing power-, performance-, and energy- efficient computing platforms. The advantage of using different compute units in the same system is that each one may be very efficient for specific computation patterns. A modern example is General Purpose programming on Graphics Processing Units (GPGPU), which exploits GPUs to perform embarrassingly parallel computations efficiently. Exotic heterogeneous architectures involve the use of approximate hardware, such as Google's Tensor Processing Units. Approximate hardware enables developers to increase the performance and energy efficiency of their applications by relaxing their expectations on the accuracy of the results.

Heterogeneity is not applicable to just hardware. It is inherently present at the level of software as well. After all, not all parts or execution phases of a program affect the quality of its output equally. In this Thesis we explore ways to optimize applications based on this aspect of software heterogeneity, through the algorithmic property of *significance*. The significance of computations is a metric that qualitatively defines the impact of the computations to the application's output quality. We introduce methodologies and frameworks which aim to identify and exploit opportunities to aggressively optimize the execution of the least significant parts of a program. These parts are costly to execute but do not heavily impact the output quality of the application.

We show that it is possible to optimize applications through the principles of significance-aware approximate computing. In addition to approximate computing, we explore a more aggressive approach, significance-aware fault tolerant computing realized through the use of unreliable hardware. Our motivation is the observation that hardware designers sacrifice performance, energy, and power efficiency

to maintain the illusion of hardware reliability. Consequently, it is possible to relax the reliability constraints of hardware to reduce the associated costs. Unfortunately, unreliable hardware may adversely affect the execution of applications. To this end, beyond approximate computing, we introduce a collection of frameworks, and methodologies which make it possible for a developer to execute parts of her application on extremely efficient, albeit unreliable hardware.

First, we introduce the basic framework for significance aware computing. It consists of a versatile programming model and accompanying runtime-systems. It allows developers to exploit the algorithmic property of significance and gracefully trade-off output accuracy with optimized code execution. Developers may characterize the different parts of their code with respect to their significance, in other words their impact to the final application output. Following the significance characterization of the code, developers opt for either approximating the least significant parts of their code, or executing them using hardware configured at potentially unreliable operating points. During execution time, the respective runtime system orchestrates the execution of different parts of the application.

Application developers provide hints to the runtime systems in order to guide the accuracy/optimization trade-off through the use of a single knob, the so called "ratio". $Ratio$ serves as a single knob to enforce a minimum quality in the performance / quality / energy optimization space. Smaller ratios give the runtimes more opportunities for optimization, however at a potential quality penalty. For applications implemented using the fault tolerant flavor of the programming model, developers may implement result checking functions and instruct the runtime system to execute them at appropriate times. In the case of the approximate computing programming model, application developers may supply an alternative, approximate implementation of tasks.

Beyond discussing the basic framework for significance-aware approximate and fault-tolerant computing we also provide methodologies to automate the process of porting applications to these two computing paradigms. Firstly, we introduce a hybrid, mathematically rigorous, profile-driven methodology which combines automatic algorithmic differentiation and interval analysis to compute the impact of selected parts of a program to its output quality. Beyond significance characterization, the same methodology may be used to provide hints to a developer to design light-weight approximate implementations of tasks.

Secondly, we propose an analytical model to predict the energy consumption of an application under different input sizes and execution configurations, in terms of number of cores used, processor frequency, and the mix of accurately and approximately executed tasks. The model is used during execution-time by the runtime system to pick the best configuration for a user-specified energy budget. Even though we discuss the model in the context of approximate computing, the same methodology with minimal modifications can be used in the context of fault-tolerant computing.

Finally, we present a methodology for automatic error detection in the output of tasks, based on Artificial Neural Networks (ANNs). We show that ANNs can be quite effective for error detection purposes, offering a good trade-off between accuracy and execution overhead. At the same time, they can be generated in a highly automated manner, with limited developer effort.

Περίληψη

Βασίλειος Βασιλειάδης

Βελτιστοποίηση της εκτέλεσης προγραμμάτων αξιοποιώντας
τη σημαντικότητα των υπολογισμών

Η κατανάλωση ενέργειας αποτελεί έναν περιοριστικό παράγοντα και βασικό μέλημα για το μέλλον των υπολογιστικών συστημάτων. Οι ετερογενείς αρχιτεκτονικές αποτελούν ένα βήμα προς την υλοποίηση πλατφορμών υπολογιστών με μεγάλη απόδοση στους τομείς της ισχύος, επίδοσης, και κατανάλωσης ενέργειας. Φυσικά, η ετερογένεια δεν ισχύει μόνο για το υλικό. Είναι εγγενώς παρούσα και στο επίπεδο του λογισμικού. Όλα τα τμήματα ή οι φάσεις εκτέλεσης ενός προγράμματος δεν επηρεάζουν την ποιότητα των παραγόμενων αποτελεσμάτων στον ίδιο βαθμό.

Στην παρούσα διδακτορική διατριβή διερευνώνται τρόποι βελτιστοποίησης των εφαρμογών με βάση αυτή την πτυχή της ετερογένειας του λογισμικού, μέσω της αλγοριθμικής ιδιότητας της σημαντικότητας (significance). Η σημαντικότητα των υπολογισμών είναι μια μετρική που καθορίζει ποιοτικά τον αντίκτυπο συγκεκριμένων υπολογισμών στην ποιότητα εξόδου της εφαρμογής. Εισάγουμε μεθοδολογίες και υλοποιούμε εργαλεία λογισμικού που στοχεύουν στον εντοπισμό και την αξιοποίηση των ευκαιριών για φιλόδοξη βελτιστοποίηση της εκτέλεσης των λιγότερο σημαντικών τμημάτων ενός προγράμματος. Αυτά τα τμήματα μπορεί να έχουν αντίστοιχο κόστος εκτέλεσης με σημαντικότερα τμήματα αλλά δεν επηρεάζουν εξίσου έντονα την ποιότητα εξόδου της εφαρμογής.

Δείχνουμε ότι είναι δυνατή η βελτιστοποίηση των εφαρμογών μέσω των αρχών της προσεγγιστικής υπολογιστικής, που κάνει χρήση της αλγοριθμικής ιδιότητας της σημαντικότητας. Εκτός από την προσεγγιστική υπολογιστική, διερευνάμε και μια πιο φιλόδοξη προσέγγιση η οποία βασίζεται στη χρήση αναξιόπιστου υλικού και τις αρχές της ανοχής εφαρμογών σε σφάλματα. Το κίνητρό μας είναι η παρατήρηση ότι οι σχεδιαστές υλικού θυσιάζουν τις επιδόσεις, την ενέργεια και την αποδοτικότητα ισχύος για να διατηρήσουν την ψευδαίσθηση της αξιοπιστίας του υλικού. Κατά συνέπεια, μπορεί κανείς να χαλαρώσει τους περιορισμούς αξιοπιστίας του υλικού για να μειώσει τα σχετικά κόστη του σε ισχύ, απόδοση, καθώς και ενέργεια. Δυστυχώς, το μη αξιόπιστο υλικό μπορεί να επηρεάσει δυσμενώς την ορθότητα της εκτέλεσης των εφαρμογών. Για το σκοπό αυτό, σχεδιάζουμε μεθοδολογίες και υλοποιούμε υποστήριξη λογισμικού που δίνουν την επιλογή σε έναν χρήστη να εκτελέσει τμήματα της εφαρμογής του σε ενεργειακά αποδοτικό, αν και αναξιόπιστο υλικό.

Αρχικά, εισάγουμε το βασικό πλαίσιο για τον υπολογισμό της σημαντικότητας. Αυτό συμπεριλαμβάνει ένα ευέλικτο μοντέλο προγραμματισμού και συνοδευτικά συστήματα

χρόνου εκτέλεσης. Επιτρέπει στους προγραμματιστές να εκμεταλλευτούν την αλγοριθμική ιδιότητα της σημαντικότητας και την ακρίβεια της απόδοσης εξόδου με βελτιστοποιημένη εκτέλεση κώδικα. Οι προγραμματιστές μπορούν να χαρακτηρίζουν τα διάφορα μέρη του κώδικά τους σε σχέση με τον αντίκτυπό τους στην τελική έξοδο της εφαρμογής. Μετά τον χαρακτηρισμό του κώδικα ως προς τη σημαντικότητα του, οι προγραμματιστές επιλέγουν είτε να εκτελέσουν προσεγγιστικά τα λιγότερο σημαντικά τμήματα του κώδικα τους είτε να τα εκτελέσουν χρησιμοποιώντας εξοπλισμό διαμορφωμένο σε δυνητικά αναξιόπιστα σημεία λειτουργίας. Κατά τη διάρκεια του χρόνου εκτέλεσης, το αντίστοιχο σύστημα χρόνου εκτέλεσης ενορχηστρώνει την εκτέλεση διαφορετικών τμημάτων της εφαρμογής.

Οι προγραμματιστές εφαρμογών παρέχουν συμβουλές στα συστήματα χρόνου εκτέλεσης για να καθοδηγήσουν την αντιστοίχιση ακρίβειας / βελτιστοποίησης μέσω της χρήσης μιας μόνο ρύθμισης, του λεγόμενου "ratio". Το ratio χρησιμεύει ως ενιαίο μοχλός για την επιβολή ελάχιστης ποιότητας στο χώρο απόδοσης / ποιότητας / ενεργειακής βελτιστοποίησης. Μικρότερες αναλογίες δίνουν στους χρόνους εκτέλεσης περισσότερες ευκαιρίες για βελτιστοποίηση, αλλά δυστυχώς και μεγαλύτερη πιθανή ποινή στην ποιότητα. Για εφαρμογές που υλοποιούνται υπό τις αρχές της ανοχής σε σφάλματα, οι προγραμματιστές μπορούν να υλοποιήσουν μεθοδολογίες για τον έλεγχο των ενδιάμεσων αποτελεσμάτων και να δώσουν εντολή στο σύστημα χρόνου εκτέλεσης να τις εκτελέσει σε κατάλληλες στιγμές. Στην περίπτωση του μοντέλου προγραμματισμού προσέγγισης, οι προγραμματιστές εφαρμογών μπορούν να παρέχουν στο σύστημα χρόνου εκτέλεσης μια εναλλακτική, κατά προσέγγιση υλοποίηση των εργασιών (approximate tasks) η οποία απαιτεί μειωμένο κόστος εκτέλεσης.

Πέρα από τη συζήτηση του βασικού πλαισίου για προσεγγιστικούς υπολογισμούς αλλά και υπολογισμούς ανθεκτικούς σε σφάλματα, παρέχουμε επίσης μεθοδολογίες για την αυτοματοποίηση της διαδικασίας μεταφοράς εφαρμογών σε αυτά τα δύο υποδείγματα υπολογισμών. Βασιζόμαστε σε μια υβριδική, μαθηματικά αυστηρή μεθοδολογία η οποία όμως είναι καθοδηγούμενη από profiling εκτελέσεις και συνδυάζει την αυτόματη αλγοριθμική διαφοροποίηση (algorithmic differentiation) αλλά και την ανάλυση διαστημάτων (interval analysis) για να υπολογίσει την επίδραση επιλεγμένων τμημάτων ενός προγράμματος στην ποιότητα εξόδου του. Εισάγουμε μεθοδολογίες ώστε να εντοπίζονται σημαντικές διαφοροποιήσεις της σημαντικότητας (significance) σε μονοπάτια του κώδικα ώστε να χαρακτηρίζεται η σημαντικότητα υπολογισμών σε αδρότερο βαθμό καταμερισμού (tasks), αλλά και να παρέχεται καθοδήγηση στους προγραμματιστές για την υλοποίηση προσεγγιστικών εναλλακτικών των tasks.

Στη συνέχεια, προτείνουμε ένα αναλυτικό μοντέλο για την πρόβλεψη της κατανάλωσης ενέργειας μιας εφαρμογής με διαφορετικά μεγέθη εισόδου και διαμορφώσεις εκτέλεσης. Μία τέτοια διαμόρφωση συμπεριλαμβάνει τον αριθμό των χρησιμοποιούμενων πυρήνων, την συχνότητα του επεξεργαστή καθώς και τον λόγο (ratio) των εργασιών που εκτελούνται με ακρίβεια και κατά προσέγγιση. Το μοντέλο χρησιμοποιείται κατά το χρόνο εκτέλεσης από το σύστημα χρόνου εκτέλεσης για να επιλέξει την καλύτερη διαμόρφωση για έναν προϋπολογισμό ενέργειας που καθορίζεται από τον χρήστη. Μολονότι

συζητούμε το μοντέλο στο πλαίσιο της προσεγγιστικής υπολογιστικής, η ίδια μεθοδολογία με ελάχιστες τροποποιήσεις μπορεί να χρησιμοποιηθεί στο πλαίσιο των υπολογισμών ανθεκτικότητας σε σφάλματα.

Τέλος, παρουσιάζουμε μια μεθοδολογία για την αυτόματη ανίχνευση σφαλμάτων στις εξόδους των εργασιών, με βάση τα Τεχνητά Νευρωνικά Δίκτυα (ΤΝΔ). Δείχνουμε ότι τα ΤΝΔ μπορεί να είναι αρκετά αποτελεσματικά για τους σκοπούς ανίχνευσης σφαλμάτων. Προσφέρουν έτσι ένα καλό αντιστάθμισμα μεταξύ ακρίβειας στην αναγνώριση σφαλμάτων και κόστους για την εκτέλεσή τους. Ταυτόχρονα, μπορούν να δημιουργηθούν με πολύ αυτοματοποιημένο τρόπο, καθώς απαιτούν περιορισμένη προσπάθεια από τη μεριά του προγραμματιστή εφαρμογών.

Acknowledgements

This thesis is the result of research work conducted while I was pursuing my PHD degree in the Department of Electrical and Computing Engineering of University of Thessaly in Greece. I acknowledge the funding agencies which made this research possible through financial means. These include the European Commission through the SCoRPiO EU project as well as the Center for Research & Technology Hellas (CERTH).

First and foremost I would like to thank my mentors, professors Christos D. Antonopoulos, Nikolaos Bellas, and Spyros Lalis 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 Konstantinos Parasyris 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, Anna and Yannis, 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

Recent reports of the International Technology Roadmap for Semiconductors (ITRS) are disconcerting to scientists and consumers alike. The 2013 ITRS report warns the public that CMOS, due to their constant horizontal scaling, will reach their atomistic and quantum mechanical physics boundaries within the next 3-12 years [37]. The report goes on to suggest that the scientific community should investigate novel designs which integrate heterogeneous technologies and new information-processing paradigms. In the past, technology was driven by the desire for more performance. However, the primary goal of Integrated Circuit (IC) design has now shifted to minimizing the power consumption of the hardware or optimizing the power/performance balance. As such, ITRS reports that beyond the year 2020 devices and systems will eventually evolve to:

- be completely new devices
- operate on completely new principles
- support completely new architectures

1.1 Problem

Hardware designers go to great lengths to improve hardware reliability. Typically, they safeguard their designs against adverse combinations of factors that affect hardware reliability. This conservative design methodology essentially results in area, performance, power and energy overheads. However, such design choices are not unreasonable. 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 fact, in the event that an error arises it is treated as a particularly rare scenario with developers actively spending effort to mask the error from the user space, regardless of the magnitude of its effects. In this Thesis we find that this course of action is not necessarily the best choice in terms of performance, power, and energy efficiency.

Possible sources of hardware unreliability are transistor variability, aging, temperature, hardware/software interaction, or even alpha particles temporarily affecting hardware functionality. Traditional design techniques addressing this problem involve extra logic circuitry provisions known as guardbands. The guardband approach deals with expected performance degradation of transistors over time. Guardbanding typically involves increased voltage margins, layout rules, and circuit redundancy. On the one hand, guardbands have successfully ensured reliable operation up to date. On the other hand, their effectiveness in detecting and correcting all possible errors is being questioned by researchers, as geometries and supply voltages are being scaled down and circuits become more vulnerable to failures [89, 13, 78]. 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 guard bands are pessimistic, as they have to compensate for the worst case scenarios and combinations of nondeterminism, switching patterns, temperature and aging effects. According to [15] 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, [20] 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 [30].

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, much like [37] warned, unless we manage to come up with novel heterogeneous architectures, technology is bound to hit again the same power wall as single core architectures did ten years ago.

1.2 Motivation

Many algorithms and application domains are amenable to approximation due to their intrinsic fault tolerance. Inherently fault-tolerant applications typically share a common property: they have relaxed accuracy constraints. In other words, they can accept a range of possible values as "correct". Figure 1.1 lists a few examples

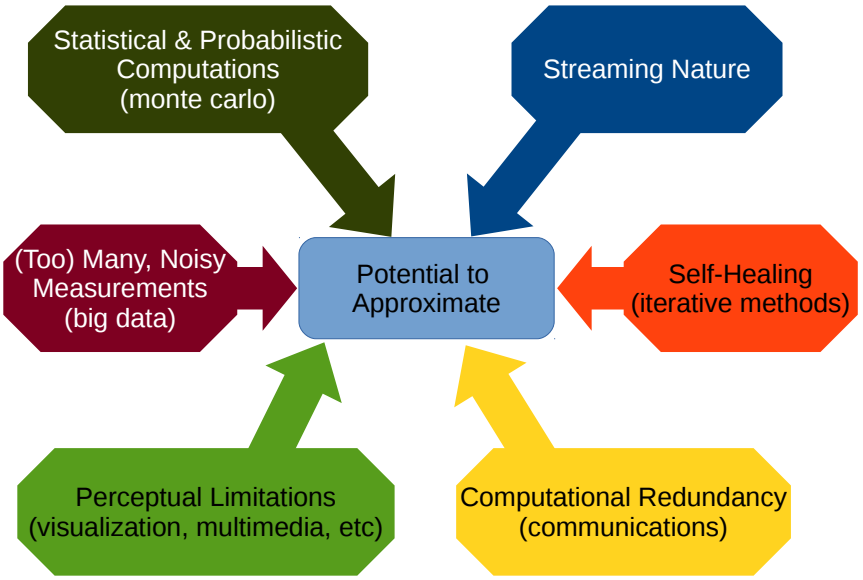


FIGURE 1.1: Application domains with intrinsic error resiliency.

of application domains which have intrinsic 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.

Iterative methods tend to be self healing. 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, such as Monte Carlo simulations also tend to suppress outliers and are thus amenable to approximation.

Approximate computing is an emerging paradigm that allows trading-off performance and energy efficiency with accuracy [4, 86, 109]. Typically, approximate computing is used as a blanket term to describe both:

a) the disciplined aggressive optimization at the algorithmic level to gracefully trade-off computation accuracy with performance/energy efficiency, and

b) the execution of instructions on hardware which 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 hardware artifacts, unreliable but 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 while executing code using unreliable hardware requires the use of fault tolerance techniques.

We make a distinction between approximate and fault tolerant computing. Approximate computing is controlled and predictable. A developer explicitly and consciously designs an approximate version of a code to be less accurate but more energy/power/performance efficient. Execution under unreliable conditions is less predictable and typically requires the application of fault tolerant computing techniques. For example, the developer must consider detecting as well as correcting errors 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 and reliability. However, as shown by previous work [64], 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 [45, 65] exploits the algorithmic property of computational significance to create optimization opportunities in terms of performance and power-efficiency of applications. This thesis explores two significance-driven approaches, which aim to gracefully trade-off application output quality with improved performance: a) significance-aware approximate computing, and b) significance-aware fault-tolerant computing. The next section discusses the key challenges of these two computing paradigms and presents our contributions.

1.3 Contributions

1.3.1 Significance aware approximate computing

We investigate significance aware approximate computing, based on the premise that specific phases of a computation may incur a high performance and energy toll without a 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 identify regions of an application which are amenable to aggressive optimizations which do not severely impact the final application outcome but lead to improvements in energy/power/performance.

Effectively applying significance aware approximate computing to gracefully trade-off application output quality with energy/performance gains requires a systematic approach to executing programs using the principles of significance aware approximate computing, as well as answering three key questions: What, how, and when to approximate?

How to implement and execute?

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 proposed paradigm. The programming model should enable application developers to both implement applications from scratch, as well as to gradually modify existing code bases to make use of the significance aware approximate computing advantages. The goal of the programming model is to give the means to application developers to use the principles of significance aware computing. The information exposed using the programming model needs to be efficiently utilized by an intelligent runtime system. More specifically, the runtime system should take advantage of the opportunities that the programming model creates for program optimization, and gracefully trade-off application quality with improved performance and or energy/power efficiency.

- We introduce a programming model which extends OpenMP, one of the most popular parallel programming models. The programming model offers the necessary expressiveness to enable the development of significance aware approximate and fault tolerant applications.
- We introduce a runtime system for significance-aware approximate computing.

What to approximate?

One could exploit the algorithmic property of significance to pinpoint parts of a program which are costly to execute but do not significantly impact the application output quality. Intuitively, a piece of code can be considered of low significance if it produces similar output values regardless of how much its input data are perturbed. On the contrary, a highly significant part of the application would produce highly varying outputs with even small perturbations to its input.

- We introduce a methodology to estimate the importance (significance) of computations for the quality of the end result. This information is used to decide which parts of a program to approximate or execute using unreliable hardware for the case of fault tolerant computing.

However, simply figuring out which parts of the program are safe to approximate is not sufficient for efficient significance aware approximate computing.

How to approximate?

Selected parts of an application which are deemed to be safe for approximations require an approximate implementation as well. By definition, this alternative algorithm is less precise than the original code but is cheaper in terms of computational cost. Typically, approximate alternatives for code are implemented by hand.

- We introduce an automatic significance analysis methodology which provides hints to application developers so that they can implement lightweight approximate alternatives for selected parts of their programs.

When to approximate?
Finally, when is it appropriate to resort to approximation? One the one hand, an application should be robust, in other words, it should always deliver user-acceptable results. 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 an application to be amenable to adaption with respect to different execution scenarios.

- We introduce a methodology which, based on application profiling and performance / energy modeling, can in turn automatically select the appropriate approximation level of computations in order to meet user specified energy budget constraints.

1.3.2 Significance aware fault tolerant computing

There is a crucial difference between significance aware fault tolerant computing and significance aware approximate computing. Executing code without reliability guarantees can lead to arbitrary errors, which are not easily controllable and may cause significant disruptions to the application output or even crash the program. 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 the answers to four questions: What/Why/When to execute unreliably? and How to detect errors? Furthermore, it also 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.

Interestingly, the answers to the questions "What/When to execute unreliably?" are identical to their sibling questions for significance aware approximate computing.

How to implement and execute?
Our programming model, through its fault-tolerance extensions, facilitates the development of applications for the significance aware fault tolerant computing paradigm

as well. A runtime system orchestrates the execution of mixed-significance tasks on unreliable hardware to exploit opportunities for graceful trade-off of program output quality with performance/energy/power improvements.

- We introduce a runtime system for significance-aware fault tolerant computing. It schedules tasks on reliable and unreliable hardware.and offers mechanisms for containment of severe errors such as crashes, elastic synchronization, and task output-error checking via user-supplied result-check functions.

Why execute unreliably?
In current and future large-scale systems unreliability is and will remain an inherent, first-class design concern. Moreover, in the quest of power and energy efficiency one may opt to purposely eliminate hardware guardbands by operating at non-nominal voltage/frequency configurations, thus increasing the chances to accidentally enter regions of operation where faults may occur. It should be noted that those regions are not static and may vary for different applications, different parts [69], different environmental conditions etc. In a sense, significance aware approximate computing exposes and takes into account the heterogeneity of importance of computations at the level of software, whereas significance aware fault tolerant computing additionally takes into account the heterogeneity of mixed-reliability hardware.

How to detect errors?
Arguably, the most important issue of significance aware fault tolerant computing is the detection of errors. Unreliable hardware is much less predictable than approximate implementations of algorithms. To this end, the need for error detection arises. Unfortunately, manually implementing detectors with high error coverage that are also also light-weight is a labor intensive process often requiring deep algorithmic insight.

- We introduce a methodology to generate lightweight error-detectors which are based on Artificial Neural Networks. These detectors can be utilized in the context of fault tolerant computing to identify errors and initiate correction actions before those errors manage to propagate to the end result of the application.

1.4 Outline

Figure 1.2 provides a high level illustration of the different aspects of significance aware approximate and fault tolerant computing that we focus on in this Thesis.

In more detail, Chapter 2 introduces the necessary background to assist the reader in understanding the methodologies presented in this document. Chapter 3 introduces the programming model which offers the necessary functionality to facilitate

the development of applications for the two computing paradigms through its extensions. Additionally, we present the runtime systems which orchestrate the execution of applications implemented using our programming model. Essentially, this chapter presents the supporting computing framework that we use throughout our research. The remaining chapters present a set of methodologies which aim to automate the process of addressing design challenges for both significance aware approximate and fault tolerant computing.

Chapter 4 discusses the algorithmic property of significance and introduces our automatic significance analysis methodology. The methodology can be used to automatically characterize parts of an application with respect to their effect to the quality of the end result. Additionally, it can be used to provide hints to a developer and assist her in implementing a light-weight, approximate implementation of the original code.

Afterwards, in Chapter 5 we demonstrate the use of a profiling- and model-based approach to 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 6 we introduce our methodology to automatically produce efficient

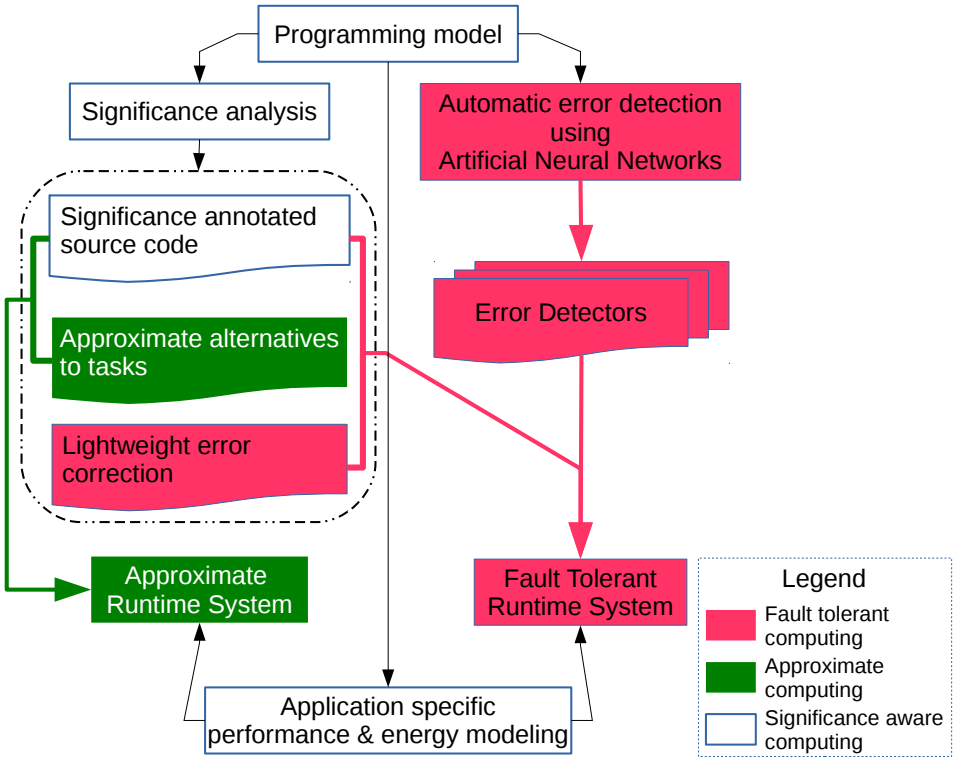


FIGURE 1.2: Overview of the interaction of all techniques introduced in this Thesis to facilitate the efficient implementation of applications using the significance aware approximate and fault-tolerant computing paradigms.

error detectors for partially unreliably executed applications. The goal of the approximate error-detectors is to identify invalid intermediate computation outputs which significantly differ from the expected ones while incurring a low execution cost overhead.

Finally, we discuss previous related work in Chapter 7 and present our concluding remarks in Chapter 8.

Chapter 2

Background

This Chapter outlines the necessary background to assist the reader in following the discussion in the subsequent Chapters, which introduce the contributions of this Thesis. Section 2.1 describes the benchmarks we used to evaluate our methodologies. In Section 2.2, we provide a detailed discussion regarding the mathematical definition of the algorithmic property of significance. We use this information in Chapter 4 to design a methodology for automatic significance analysis of application codes.

Section 2.3 provides background information regarding fault, energy, and execution time modeling for the purpose of simulating execution of code on unreliable environments through the use of software fault injection.

2.1 Benchmarks

This section describes the various benchmarks used throughout this document. The benchmarks are implemented using our OpenMP-like programming model which is introduced in Chapter 3.

2.1.1 DCT

Discrete Cosine Transformation (DCT) is used in image and video compression to transform a block of 8x8 image pixels to a block of 8x8 frequency coefficients. Low frequency coefficients are closer to the upper left corner of the 8x8 block, whereas high frequency coefficients reside in the lower right corner. A single task computes a 2x4 block of frequency coefficients. In order to produce an output and evaluate its quality, we drive the output of DCT to a quantization kernel, the result of which is then fed into a dequantization step. The dequantized DCT frequencies are then processed by an Inverse DCT (iDCT) step which produces an image. This pipeline effectively simulates the process of compressing and decompressing images. We measure the output quality by computing the Peak Signal to Noise Ratio (PSNR) between the input image and the decompressed image that was produced by iDCT.

2.1.2 Sobel

Sobel filtering is used in image processing and computer vision to produce an image emphasizing edges. It convolves the image with a $3x3$ block filter, once in the horizontal (t_x) and once in the vertical (t_y) direction. Afterwards, it combines the results (t_x and t_y) of the two convolutions to compute an intermediate value $t = \sqrt{t_x^2 + t_y^2}$. The output pixels are then produced by clipping t to the range of $[0, 255]$.

2.1.3 K-means

K-means is an iterative algorithm for grouping data points from a multi-dimensional space into *k* clusters. Each iteration consists of two phases: Chunks of data points are first assigned to different tasks, which independently determine the nearest cluster for each data point. Then, another task group is used to update the cluster centers by taking into account the position of the points that have moved.

2.1.4 Jacobi

Jacobi is an iterative solver for diagonally dominant systems of linear equations described by the equation $Ax = b$.

2.1.5 Blackscholes

Blackscholes is a benchmark of the PARSEC suite [8]. It implements a mathematical model for a market of derivatives, which calculates the buying and selling price of assets so as to reduce the financial risk. A task uses the Black-Scholes mathematical model to produce the price of N assets.

2.1.6 Fisheye

Fisheye [7] is an image processing application that transforms images distorted by a fisheye lens back to the natural-looking perspective space. The exact algorithm initially associates pixels of the output, perspective space image, to points in the distorted image. Then, interpolation on a 4×4 window is applied to calculate each pixel value of the output, based on the values of neighboring pixels of the corresponding point in the distorted image.

2.1.7 N-Body

The N-Body (molecular dynamics) application simulates the kinematic behaviour (position and velocity) of liquid Argon atoms within a bounded space, under the effects of a force produced by a Lennard-Jones pair potential [42]. The potential is defined as a function of distance (r) and two material specific constants (σ and ϵ):

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.1}$$

The significance of the interaction between atoms is strongly correlated with the distance between them. The greater the distance between atom A and atom B, the less the kinematic properties of A affect those of B (and vice versa). In the task-based version of N-Body, the 3D container of the particles is partitioned into regions which are updated every few time-steps to populate a list of the particles that reside inside them. For each given atom, one task per region is instantiated to calculate the forces that operate on the atom due to the particles contained in that specific region.

2.1.8 Lulesh

Lulesh [46] implements a solution of the Sedov blast problem for a material in three dimensions. It defines a discrete mesh that covers the region of interest and partitions the problem into a collection of elements where hydrodynamic equations are applied. A single task performs the computations required to compute the hydrodynamic equations for eight elements.

2.1.9 Bonds

Bonds [25] is a computational finance benchmark of the QuantLib library. In finance, a bond is an indication of indebtedness of the bond issuer to the holders. The issuer is obligated to pay the holders a debt which increases by a specified interest and/or pay the face amount at a pre-determined date which is referred to as the maturity date. Interests payments are deposited in intervals. A single task processes N bonds.

2.1.10 MC

MC [103] applies a Monte Carlo approach to estimate the boundary of a sub-domain within a larger partial differential equation (PDE) domain, by performing random walks from points of the sub-domain boundary to the boundary of the initial domain. An MC task performs N random walks and to estimate a solution for the walks' point of origin.

2.1.11 Bodytrack

Bodytrack is a benchmark from the PARSEC suite [8]. It uses an annealed particle filter to track the pose of a human subject in a series of frames which are captured from multiple angles using multiple cameras. A single task processes a subset of the particles associated with a particular frame.

2.1.12 Inversek2j

Inversek2j is a physics benchmark from the AxBench suite [107]. Inversek2j calculates the angles of a two joint arm using the kinematic equation. A task computes the pair of angles for a single two joint arm.

2.1.13 Barnes

Barnes [5] is a physics benchmark from the SPLASH2 suite [106]. It is an N-body simulation. A single step of the simulation calculates the forces acting on each body and subsequently updates the position, velocity and acceleration of each body. The total workload is divided into N smaller workloads and each one is handled by a single task.

2.1.14 Canneal

Canneal, a code from the Parsec benchmark suite [8], applies an annealing methodology to optimize the routing cost of a chip design. This optimization method pseudo-randomly swaps net-list elements. If the swap results in better routing cost it is accepted immediately. Local minima are avoided by rarely accepting swaps that increase the routing cost of the net-list. A single task processes a sub-set of the net-list.

2.2 Mathematical definition of the algorithmic property of significance

In this section we present the mathematical definition of algorithmic significance [102]. We use this mathematical definition as the basis for our hybrid (profile driven, yet mathematically rigorous) analysis which we discuss in Chapter 4.

2.2.1 Significance as an Algorithmic Property

The algorithmic property of computational significance denotes the contribution of a computation to the final output result. Consider the following example. For an input vector $\vec{x} \in I\!R^n$, the arithmetic evaluation of $y = f(\vec{x})$ can be written as a three part evaluation procedure [26] with internal variables u_j:

$$u_{k-n+1} = x_k, \qquad k = 0, \ldots, n-1, \tag{2.2}$$
$$u_j = \phi_j(u_i)_{i \prec j}, \qquad j = 1, \ldots, p, \tag{2.3}$$
$$y = u_p. \tag{2.4}$$

The input values $\vec{x} = (x_0, \ldots, x_{n-1})^T$ are copied in Eq. 2.2 to internal variables $u_{1-n}, \ldots, u_0$, and Eq. 2.4 stores the final result in y. In Eq. 2.3 the elementary function $\phi_j(u_i)$ represents an arithmetic operation (+,−,⋆,/), or an intrinsic function (sin, cos, exp, ...) of C++. Internal variables u_j are used to store the result of each elementary function evaluations, where $i \prec j$ denotes a direct dependence of u_j on u_i, $i < j$.

The evaluation of the compiled program for a specific input $\vec{x} \in I\!R^n$ leads to a unique sequence of elementary operations since all control flow decisions (branches, number of loop iterations) are determined uniquely. This sequence of elementary operations performed during such an evaluation is the basis of the three part evaluation procedure.

```
1 double f( double x0 ) {
2    return cos(exp(sin(x0) + x0) - x0);
3 }
```

LISTING 2.1: Implementation of example function.

```
1   u0 = x0
2   u1 = sin(u0)        u2 = u1 + u0        u3 = exp(u2)
3   u4 = u3 − u0        u5 = cos(u4)
4   y  = u5
```

LISTING 2.2: Elementary functions of `Listing 2.1`.

Listing 2.1 shows a concrete example for `C++` function $y = f(x_0)$ with scalar input x_0 and scalar output y. The respective code, broken down to single elementary functions, is given in *Listing* 2.2, where the first line corresponds to Eq. 2.2, the sequence of elementary functions in Eq. 2.3 is represented by lines 2–3 and line 4 corresponds to Eq. 2.4.

To evaluate the significance of variables $u_j, j = 1-n, \ldots, p$, for the final result $y = u_p$, we need to answer two questions: (a) What is the influence of inputs $\vec{x}$ on u_j, and (b) what is the influence of u_j on output $y = u_p$. Answering (a) requires analysing the code which computes the variable u_j from its inputs $u_1, \ldots, u_{j-1}$ (Eq. 2.3). The answer to (b) requires analyzing the use of u_j in obtaining the final result y, through the computation of $u_{j+1}, \ldots, u_p$. We denote this second part with $y = f(\vec{x}; u_j)$ to represent the dependency of y on u_j explicitly.

Interval arithmetic (IA) [66] is an appropriate tool to answer the first question: Given the range of possible input values as the input interval vector $[\vec{x}] = [\underline{x}, \overline{x}] = \{x \in I\!R^n | \underline{x} \leq x \leq \overline{x}\}$ with lower bound $\underline{x} \in I\!R^n$ and upper bound $\overline{x} \in I\!R^n$, an evaluation of f in interval arithmetic is obtained by replacing all variables and elementary functions ϕ_j with their interval version in Eq. 2.2-2.4:

$$[u_{k-n+1}] = [x_k], \qquad k = 0, \ldots, n-1, \tag{2.5}$$

$$[u_j] = \phi_j[u_i]_{i \prec j}, \qquad j = 1, \ldots, p, \tag{2.6}$$

$$[y] = [u_p]. \tag{2.7}$$

This will compute an interval enclosure $f[\vec{x}]$ of all possible values of $f(\vec{x})$ for $\vec{x} \in [\vec{x}]$, namely $f[\vec{x}] \supseteq \{f(\vec{x}) | \vec{x} \in [\vec{x}]\}$.

With IA, value ranges are propagated *forward* through the computation. For every variable $u_j, j = 1, \ldots, p$, we calculate an enclosure $[u_j]$ of all possible values for the given input range $\vec{x}$. The impact of all inputs $[x_k]$, $k = 0, \ldots, n-1$, on a variable u_j is combined in $[u_j]$, and can be quantified by the width $w([u_j]) = \overline{u_j} - \underline{u_j}$ of interval $[u_j]$: if $w([u_j])$ is narrow, variation of the input within the given range causes only little variance in $[u_j]$ (small influence). On the other hand, if $w([u_j])$ is wide, the exact value of the input has large influence on variable u_j.

```
1  u_(1)5 = y_(1)
2  u_(1)4 = − sin(u_4) · u_(1)5
3  u_(1)3 = 1 · u_(1)4
4  u_(1)2 = exp(u_2) · u_(1)3
5  u_(1)1 = 1 · u_(1)2
6  u_(1)0 = (−1) · u_(1)4 + 1 · u_(1)4 + cos(u_0) · u_(1)1
7  x_(1)0 = u_(1)0
```

LISTING 2.3: Adjoint code for *Listing* 2.2.

But the significance of u_j for the output y cannot be judged from this information alone. Subsequent operations during the evaluation of $y = f(\vec{x}; u_j)$ by computing u_k, $k = j+1, \ldots, p$, may amplify or reduce the contribution of u_j to y. Therefore, we also need to evaluate how much y will change for different values of u_j, or more formally, to quantify the impact of $[u_j]$ on y (question (b) above). Pure IA evaluation of $f(\vec{x}; u_j)$ does not suffice, since in the final interval value $[y]$ the individual impact of variables $[u_j]$ cannot be obtained separately. The answer to question (b) is inspired by the fact that the first order derivative of a differentiable function at a given point describes the function behavior in a neighborhood of that point. For a given point $\hat{x} \in I\!R^n$ and a function $f : I\!R^n \longrightarrow I\!R$ differentiable at point $\hat{x}$, $\nabla_{\vec{x}} f(\hat{x}) = (\frac{\partial f(\hat{x})}{\partial x_0}, \ldots, \frac{\partial f(\hat{x})}{\partial x_{n-1}})^T$ is the gradient of f at point $\hat{x}$. The elements of gradient $\nabla_{\vec{x}} f(\hat{x})$ quantify the rate of change in the function value near $\hat{x}$: if the absolute value of the partial derivative $\frac{\partial f(\vec{x})}{\partial x_i}$ is small, a disturbance in x_i will cause a small change in the function value $f(\vec{x})$.

Consider a computer program implementing a differentiable function $y = f(\vec{x})$. The gradient $\nabla_{\vec{x}} f$ of f at the evaluation point $\vec{x}$ can be obtained by *Algorithmic Differentiation (AD)* [26, 67] in adjoint mode. Based on the three part evaluation procedure Eq. 2.2-2.4, adjoint evaluation propagates the first order adjoint (denoted by subscript $_{(1)}$) $y_{(1)}$ of output y *backwards* through the computation towards first order adjoints $x_{(1)k}$, $k = 0, \ldots, n-1$, of the inputs $\vec{x}$:

$$u_{(1)p} = y_{(1)}, \tag{2.8}$$

$$u_{(1)i} = \sum_{j:i \prec j} \frac{\partial \phi_j(u_i)_{i \prec j}}{\partial u_i} \cdot u_{(1)j}, \quad i = p-1, \ldots, 1-n, \tag{2.9}$$

$$x_{(1)k} = u_{(1)k-n+1}, \quad k = 0, \ldots, n-1, \tag{2.10}$$

where $\frac{\partial \phi_j(u_i)_{i \prec j}}{\partial u_i}$ denotes the partial derivative of elementary function ϕ_j with respect to its argument u_i. After a single adjoint propagation with $y_{(1)} = 1$ the gradient $\nabla_{\vec{x}} y = \nabla_{\vec{x}} f = (x_{(1)0}, \ldots, x_{(1)n-1})^T$ can be harvested from the adjoints $u_{(1)k-n+1}$, $k = 0, \ldots, n-1$, of input $\vec{x}$. Moreover, derivatives $\nabla_{u_j} y = \frac{\partial f(\vec{x}; u_j)}{\partial u_j} = u_{(1)j}$ of y with respect to all internal variables u_j, $j = 1, \ldots, p$, are accumulated during the so-called reverse sweep.

Note that adjoint propagation expects that the original code (*Listing* 2.2 for the example in *Listing* 2.1) has been evaluated beforehand. Thus intermediate variables u_j, $j = 1-n, \ldots, p$, hold the actual values. The first line in *Listing* 2.3 corresponds to

Eq. 2.8, while the gradient harvesting of Eq. 2.10 is represented by the last line. The actual adjoint propagation (Eq. 2.9) is done in lines 2–4.

AD can be applied to interval functions [87] by replacing all variables and partial derivatives of elementary functions in Eq. 2.8-2.10 with their interval version. Therefore, we can compute an interval enclosure of the first order derivative $\nabla_{[u_j]}[y] = \frac{\partial f[\vec{x};u_j]}{\partial [u_j]}$, namely the derivative of the function result $[y]$ with respect to the internal variable $[u_j]$:

$$\nabla_{[u_j]}[y] \supseteq \left\{ \frac{\partial f(\vec{x};\hat{u}_j)}{\partial u_j} \middle| \hat{u}_j \in [u_j], x \in [x] \right\}. \tag{2.11}$$

In other words, the bounds of interval derivative $\nabla_{[u_j]}[y]$ are the steepest downward and upward slopes, respectively, of $y = f(\vec{x};u_j)$ in the interval $[u_j]$, which quantify the impact of all possible values from $[u_j]$ on the final result y.

We can now define the significance $S_y(u_j)$ of variables u_j, $j = 1-n,\ldots,p$, for the final result $y = f(\vec{x})$ over the input interval $[\vec{x}]$ as follows:

$$S_y(u_j) = w\left([u_j] \cdot \nabla_{[u_j]}[y]\right), \quad j = 1-n,\ldots,p. \tag{2.12}$$

Note that the interval product of $[u_j]$ and the interval derivative $\nabla_{[u_j]}[y]$ is a worst case scenario, that might introduce a considerable overestimation of the significance of u_j.

2.2.2 Limitations

This approach comes with some limitations. A simple transformation of code with real variables into an interval version might fail for various reasons (overestimation due to wrapping effect, special interval algorithms required, relational operators). Moreover, AD computes derivatives for a given evaluation point. The elementary function sequence in the implementation of function f is fixed and can be represented by a control flow free code. With IA, comparisons between values is no longer unique: for $c < [x]$ with $c \in [x]$, the answer is neither true nor false, since a part of interval $[x]$ is less than c whereas the remaining part is not. Since a fixed control flow is not guaranteed, in such scenarios the analysis is terminated and the relevant condition statement is reported to the user. Circumventing this issue by an automatic interval splitting approach is part of ongoing research.

This method allows the developer to utilize all language tools including arrays, dynamically allocated memory, pointers, and nested loops. However, the analysed code must be differentiable, which might not apply to codes universally. Currently we consider that it is the responsibility of the developer to check the differentiability of the code/function to be analyzed.

2.2.3 dco/scorpio Framework

The significance analysis of Section 2.2.1 is implemented in the profile-driven tool `dco/scorpio` which is based on the template class library `dco/c++` (Derivative

```
1 dco::ia1s::type f( dco::ia1s::type x0 ) {
2    return cos(exp(sin(x0) + x0) - x0);
3 }
```

LISTING 2.4: The example of Listing 2.1 with *double* being replaced with *dco::ia1s::type*.

Code by Overloading in C++) [57, 68, 93] implementing tangent-linear and adjoint Algorithmic Differentiation. For any C++ code implementing $y = f(\vec{x})$, dco/c++ exploits overloading of operators and intrinsic functions to compute derivatives $\nabla_{\vec{x}} y = \nabla_{\vec{x}} f(x)$ of outputs y with respect to input $\vec{x}$.

For the purpose of significance analysis, dco/c++ templates were specialized with an interval base type [53] to obtain *dco::ia1s::type*, which enables AD on interval functions. An interval enclosure of $[y] = f[\vec{x}]$ can be obtained for a C++ implementation of $f(\vec{x})$ by defining all variables, required to compute the output y (including y), as *dco::ia1s::type* instances (*Listing 2.4*, compare to *Listing 2.1*) and performing a profile run. To compute the interval valued first order derivative $\nabla_{[x]}[y] = \nabla_{[x]} f[x]$, the dco/scorpio internal recording mechanism needs to be activated. During the evaluation of the code $f[\vec{x}]$ (with variables of type *dco::ia1s::type*), an internal representation of the computation sequence is stored within a Dynamic DFG (DynDFG). A DynDFG is a directed acyclic graph $G = (V, E)$, where each vertex u_j corresponds to a dynamically executed elementary function $u_j = \phi_j(u_i)_{i \prec j}$, and an edge $e_{i,j} \in E$ between vertex u_i and u_j means that u_i provided an input operand to operation ϕ_j during execution. Moreover, the edges are annotated with interval valued partial derivatives $\frac{\partial \phi_j[u_i]}{\partial [u_i]}$ which are computed during forward sweep.

Figure 2.1a shows the annotated DynDFG of an interval evaluation of the example function given in Listing 2.1 by the implementation given in *Listing 2.4*.

The scalar input $[x_0]$ of the user code will be associated with the internal variable $[u_0]$. Five internal variables $[u_j] = \phi_j[u_i]_{i \prec j}$, $j = 1, \ldots, 5$, are computed before the final value is stored in the output value $[y]$ of the user code. Edges towards a vertex $[u_j] = \phi_j[u_i]_{i \prec j}$ are annotated with local partial derivatives of the operation represented by $[u_j]$ with respect to its arguments $[u_i]$, $i \prec j$. Note that the interval operations, DynDFG recording, and adjoint propagation are hidden within the data type *dco::ia1s::type* of dco/scorpio.

With first order adjoint mode, AD derivatives are computed by propagating an initial adjoint $y_{(1)} = 1$ backwards through the DynDFG using the internally stored local partial derivatives according to Eq. 2.9. After the adjoint interval propagation (reverse sweep), the interval derivative $\nabla_{[x_0]}[y] = \nabla_{[x]} f[x_0] = \nabla_{[u_0]}[u_5]$ can be retrieved from dco/scorpio's internal representation along with the interval derivatives $\nabla_{[u_j]}[u_5] = \nabla_{[u_j]}[y]$ of the final result with respect to *all* internal variables $[u_j]$, $j = 1, \ldots, 5$ (*Figure 2.1b*). Using this information and Eq. 2.12 we can compute significances $S_{[y]}[u_j]$ of all variables $[u_j]$, $j = 0, \ldots, 5$.

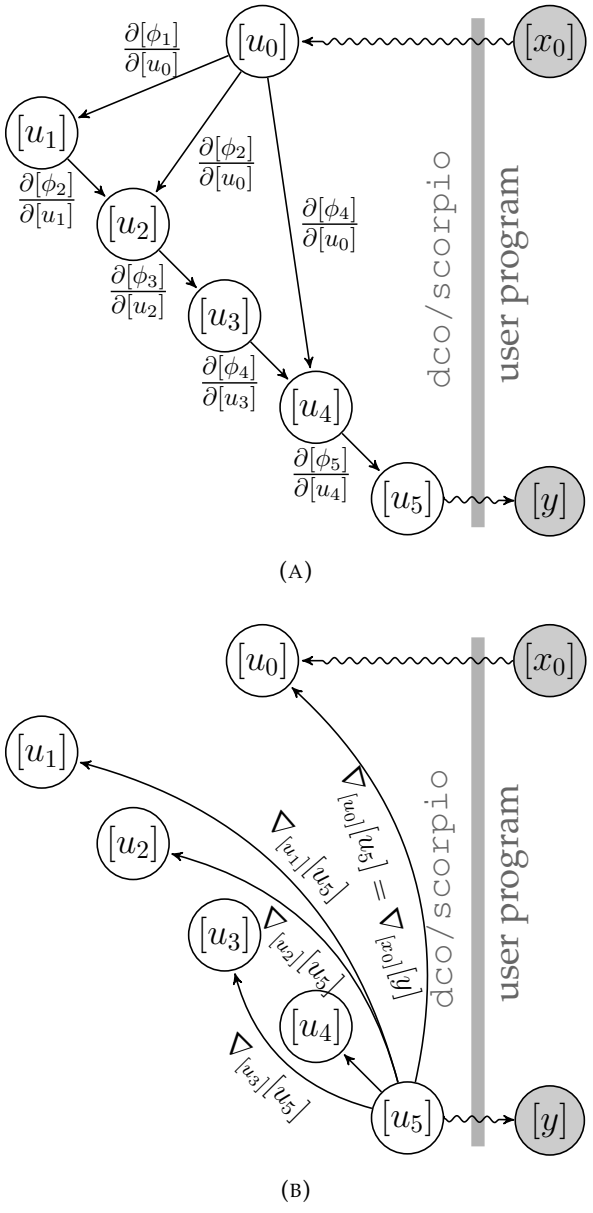


FIGURE 2.1: Annotated DynDFG and adjoint propagation : (a) DynDFG of $f(x)$ with local partial derivatives. (b) Derivatives available after evaluating $\nabla_{[x]}[y]$.

`dco/scorpio` offers a set of macros (*Table 2.1*) to annotate source code for significance analysis. They establish a link between variables in the code and their internal representation in the tool and hide all implementation details from the user. All inputs need to be registered before the first intermediate user variable, intermediate user variables need to be registered straight after their computation, and output variables last. Moreover, for a vector valued function $\vec{y} = F(\vec{x})$ with $F : I\!R^n \longrightarrow I\!R^m$, significances $S_{\vec{y}}(u_j) = \sum_{i=0}^{m-1} S_{y_i}(u_j)$ can be obtained by a single run by registering all output $\vec{y}$ variables. .

Macro	Description
INPUT(x,xl,xu, . . .)	Register input variable x, increment the number of inputs n, set $[x] = [xl, xu]$, associate x with the internal input variable $[u_{-n}]$.
INTERMEDIATE(z, . . .)	Register intermediate variable z, associate it with last computed internal variable $[u_j]$.
OUTPUT(y, . . .)	Register output variable y, associate it with the last computed internal variable $[u_p]$, set the adjoint $[u_p]_{(1)} = 1$,
ANALYSE()	Start adjoint propagation to obtain $\nabla_{[u_j]}[y]$, $j = p - 1, \ldots, 1 - n$, compute significance of all registered inputs and intermediate user variables, report the significance of registered variables.

TABLE 2.1: Macros of the `dco/scorpio` tool

2.3 Fault, Time, and Energy models to facilitate software fault injection

This section discusses the models we use for unreliable execution which were introduced in [71]. 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. We also present the time and energy models that we use in conjunction with our software-based fault injection methodology. These models enable us to simulate and evaluate the execution of computations under unreliable conditions.

Note that it was impossible to conduct the full extent of our research using a purely simulated execution platform, without resorting to software fault injection. 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.

2.3.1 Fault modeling

A key challenge is to associate the operation of a core in an unreliable configuration envelope 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. For example, instructions which activate long paths, which are close to the critical path, tend to fail more frequently [72]. 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 [15]. Moreover, whether a fault manifests as an error also depends on the paths which were activated in previous cycles [97].

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, and take 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 ∼10 million cycles). Prior to this point errors still occur, however at rates that are orders of magnitude lower [15]. If one goes beyond the PoFF, the fault rate increases exponentially, by one order of magnitude for every $10mV$ drop of the supply voltage [9, 15].

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% [29, 75, 39]. We determine the *PoFF* based on Equation 2.13, 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 [29, 9, 15]. Based on the same reports, we model the fault rate as shown in Equation 2.14. The parameters are the voltage V_{PoFF}, which can be obtained using equation 2.13 using as input argument the nominal voltage V_n and the voltage of the requested (unreliable) operating point V_u. The model obtains the constants β, γ via regression[1] on the data provided in [9, 15]. The Equation 2.14 is CPU agnostic and estimates the mean number of cycles between the manifestation of two consecutive faults. Recall that, faults are the results of the unreliable operation of hardware because of the sub-nominal voltage supply V_u. The further away V_u is from V_{PoFF} the fewer the cycles between two consecutive faults.

$$V_{PoFF} = \frac{(100 - \rho)}{100} \times V_n \tag{2.13}$$

$$Err(V_{PoFF}, V_u) = \beta \times 10^{\gamma * (V_u - V_{PoFF})} \tag{2.14}$$

[1]The resulting values are $\beta = 10^7$ and $\gamma = 100$.

$$V_n(f) = \delta \times f + \eta \tag{2.15}$$

Finally, the nominal supply voltage V_n is linearly dependent on the operating frequency, as modeled by Equation 2.15. Parameters δ and, η depend on the system configuration[2]. We deduce their values by monitoring the supply voltage of the CPU of our x86 platform, while commanding changes of the operating frequency.

2.3.2 Simulation-based fault injection

We use the GemFI framework [70] to execute 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; the injection of the fault only lasts for one clock cycle.

Simulated fault injection captures the "average" impact of faults on an application executed on top of unreliable hardware, without employing any protection mechanisms. The number of experiments for each application and pipeline stage (see above) is determined based on the methodology described in [54], for a 99% confidence level and 1% error margin.

For the purpose of our evaluation, we categorize the outcome of program execution in three classes: (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.

2.3.3 Software-based fault injection during native execution

For the native (fast) executions of the benchmarks, we use software-based fault injection. This is designed to have two possible effects: (i) it forces a crash, and (ii) it

[2]We obtain the values $\delta = 0.078170317$ and $\eta = 0.7706445993$ via profiling.

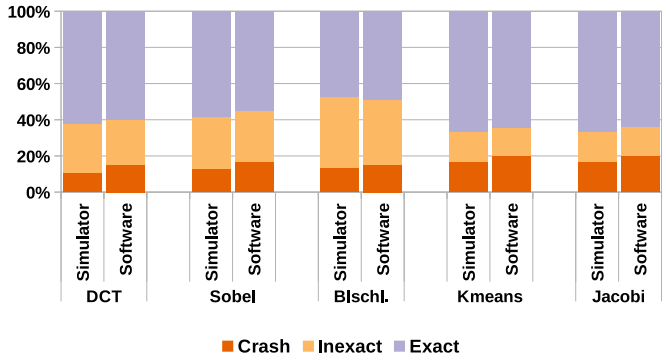


FIGURE 2.2: Effects of single fault injection, using the GemFI simulator at the architectural CPU level, and the software-based approach during native execution.

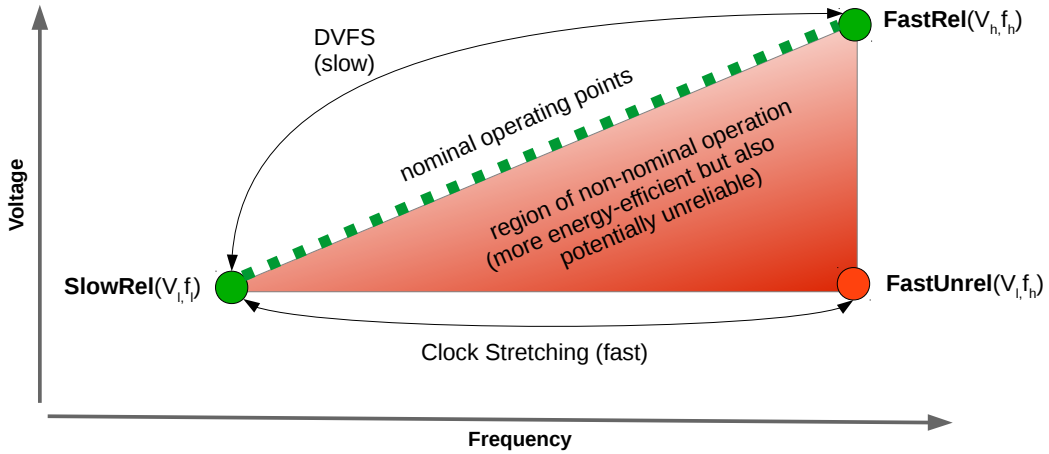


FIGURE 2.3: Hardware configurations with different reliability/performance trade-offs.

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, we inject faults to the whole extend of an application without the use of any protection mechanism. 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 2.2, which presents the results for 5 benchmarks, shows that the software-based fault injection has practically equivalent effects to simulation-based fault injections using GemFI.

Hardware configurations

We consider three different configurations, $FastRel$, $SlowRel$ and $FastUnRel$. The $FastRel$ configuration is a high-performance nominal point of operation, with high voltage/frequency (V_h, f_h), where a core executes code fast, whereas $SlowRel$ is a slower nominal operation point, with lower voltage/frequency (V_l, f_l). Furthermore, cores can be set in the non-nominal and unsafe $FastUnRel$ configuration (V_l, f_h), 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 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 configuration 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 [12]. Figure 2.3 illustrates the principle of operation.

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 the respective nominal voltage V_h from Equation 2.15. We then set $V_l = \epsilon \times V_h | \epsilon < 1.0$. Frequency f_l is derived from Equation 2.15, and the fault rate of the $FastUnRel$ configuration is derived from Equation 2.14, 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.

2.3.4 Execution Time and Energy Consumption Model

We use an analytical model for the performance and energy consumption of a program that is implemented in a task-based fashion. We generate the analytical model 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 that was introduced in [71]. The 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.

2.3.5 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 2.16 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 $SlowRel$).

$$T(N, f, C) = \frac{C}{f} \times N \tag{2.16}$$

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. The analytical time model is shown in Equations 2.17:

$$\begin{aligned} T_{FastRel} &= T(N_r, C, f_h) \\ T_{SlowRel} &= T(N_u, C_{dc}, f_l) \\ T_{FastUnRel} &= T(N_u, C, f_h) \\ T_{vfs} &= N_{FR\rightarrow SR} \times T_{FR\rightarrow SR} + N_{SR\rightarrow FR} \times T_{SR\rightarrow FR} \end{aligned} \tag{2.17}$$

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 $N_{FR\rightarrow SR}$ denotes the number of times the runtime system switches from the $FastRel$ to $SlowRel$, and $T_{FR\rightarrow SR}$ is the average time required to perform this transition. Similar parameters apply for the reverse direction.

2.3.6 Power and energy modeling

The total power dissipation of a CMOS circuit is given by Equations 2.18. 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 approximately equal to the product of the effective switched capacitance (C), the supplied voltage squared (V^2), the frequency (f), and a scaling factor (a). We use regression to estimate the product $a \times C$ which we then plug into our power model[3].

[3]For our Intel Quad Core i7 IvyBridge CPU we calculate that the product $a \times C$, when using 4 cores, is equal to $11.40\text{e}{-09}$.

$$
\begin{aligned}
P_{Total} &= P_{idle} + P_{dyn} \\
P_{idle} &= P_{leak} + P_{shortC} \\
P_{dyn}(V, f) &= a \times C \times V^2 \times f
\end{aligned}
\tag{2.18}
$$

The total energy dissipation E_{Total} is given by Equations 2.19. In general, this depends on the hardware configuration and the time spent to execute the runtime management functions, the application tasks, and their result-check/repair functions, as discussed above.

$$
\begin{aligned}
E_{FastRel} &= P(V_h, f_h) \times \max_{i=1}^{Workers} \left(T_{FastRel}\right) \\
E_{FastUnRel} &= P(V_l, f_h) \times \max_{i=1}^{Workers} \left(T_{FastUnRel}\right) \\
E_{SlowRel} &= P(V_l, f_l) \times \max_{i=1}^{Workers} \left(T_{SlowRel}\right) \\
E_{vfs} &= N_{FR\rightarrow SR} \times T_{FR\rightarrow SR} \times P(V_h, f_h) + N_{SR\rightarrow FR} \times T_{SR\rightarrow FR} \times P(V_l, f_l) \\
E_{Total} &= E_{FastRel} + E_{SlowRel} + E_{FastUnRel} + E_{vfs}
\end{aligned}
\tag{2.19}
$$

2.3.7 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 3.9.3. The parameters f_h and f_l are known while N_r, N_u, $N_{FR\rightarrow SR}$, $N_{SR\rightarrow FR}$ can be measured. C and C_{dc} are profiled using *likwid* [95] by accessing the x86 performance counters. Similarly, $T_{FR\rightarrow SR}$ and $T_{SR\rightarrow FR}$ are profiled using the *FTaLaT* tool [59]. 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, and measure the power consumption. We then perform linear regression using least squares to derive the product $a \times C$ and the parameter 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, which correspond to different P-states[4], by applying a software-driven voltage and frequency transition.

Figure 2.4 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

[4]P-states are voltage-frequency pairs that specify the performance and power consumption of a processor.

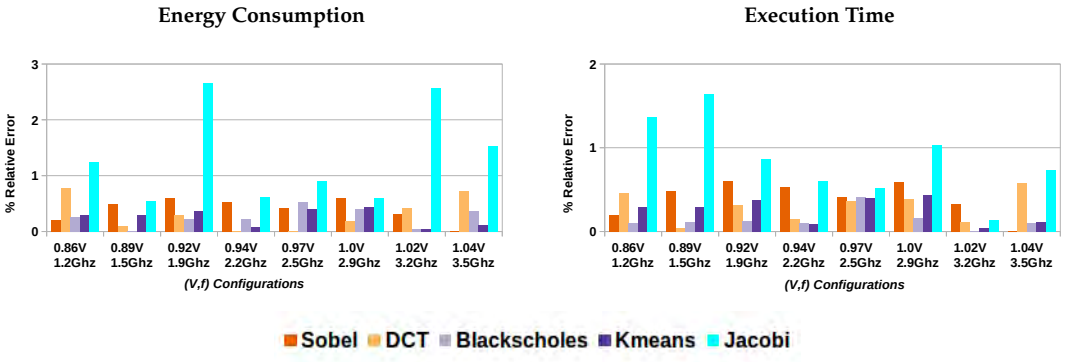


FIGURE 2.4: Relative error for the execution time and energy as predicted by our model vs. a real execution, for our application benchmarks when half of the tasks execute in the $FastRel = (3.7GHz, 1.06V)$ configuration and the other half in a lower-power $SlowRel$ configuration. All $SlowRel$ configurations are shown in x-axis.

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.

Chapter 3

Significance-aware computing framework

As we discussed earlier, part of the energy inefficiency problem that current computing systems are facing is that all computations are treated as equally important, despite the fact that only a subset of these computations may be critical to achieve an acceptable quality of service (QoS). A key challenge though, is how to identify and tag computations of the program which must be executed accurately from those that are of less importance and thus can be executed approximately.

This chapter presents our answer to the question of "**How** to execute code using the principles of significance aware approximate and fault-tolerant computing?" In the remainder of this chapter, we make the following contributions:

(i) We introduce a programming model which allows programmers to exploit the algorithmic property of significance to optimize the execution of their applications. The programming model offers extensions to facilitate the development of applications for two paradigms:

   (a) Significance aware approximate computing: A software-only approach which allows developers to utilize lightweight implementations for codes which do not significantly impact the final output quality

   (b) Significance aware fault tolerant computing: Application programmers can exploit two levels of heterogeneity, at both the level of hardware as well as software, to optimize the execution of their applications with minimal cost to the output quality.

(ii) The programming model is implemented in a source-to-source compiler. The is accompanied with two runtime systems, one for each proposed significance aware computing paradigm.

(iii) We present two case studies on how to efficiently trade-off quality of output to optimize the execution of applications using our basic a) significance aware approximate computing, and b) significance aware fault tolerant computing frameworks.

Our results show that it is possible to gracefully trade-off quality of output with more efficient execution of applications. However, it becomes clear that the application developer needs to spend effort in order to expose the algorithmic property of significance to the underlying layers of system software.

3.1 Programming model objectives

Our vision is to elevate significance characterization as a first class concern in software development, similar to parallelism and other algorithmic properties that are traditionally the focus of programmers. To this end, there is a need for a programming model which enables developers to efficiently expose of the algorithmic property of significance. The main objectives of the programming model are the following:

- to allow programmers to express the significance of computations in terms of their contribution to the quality of the end-result;
- to allow programmers to express parallelism, beyond significance;
- to allow programmers to control the balance between energy consumption and the quality of the end-result, without sacrificing performance;
- to enable optimization and easy exploration of trade-offs at execution time;
- to be user friendly and architecture agnostic.

The programming model requires a number of extensions to facilitate approximate and fault tolerant computing. The objectives for these two flavors are summarized in the following list:

- to allow programmers to specify approximate alternatives for selected computations;
- to allow programmers to specify functions which perform error detection for tasks which have been executed under unreliable conditions;
- to allow programmers to specify functions which correct/re-execute/approximate the outputs of tasks which have been found to be erroneous by an error detector

3.2 Task instantiation

Programmers express significance semantics using *#pragma* compiler directives. Programming models that are based on pragmas facilitate non-invasive and progressive code transformations, without requiring a complete code rewrite. We adopt a task-based paradigm, similarly to OmpSS [17] and the latest version of OpenMP [10].

Task-based models offer a straightforward way to express communication across tasks, by explicitly defining inter-task data dependencies. 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, but is performed at runtime, also taking into account the data dependencies among tasks.

```
1 #pragma omp task [significant(expr(...))]
2   [label(...)] [in(...)] [out(...)]
3   [approxfun(function_ptr)]
4   [taskcheck(function_ptr)]
```

LISTING 3.1: #pragma omp task

Tasks are specified using the *#pragma omp task* directive (Listing 3.1), followed by a function which is equivalent to the task body.

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. Depending on their (relative) significance, tasks may be approximated / dropped or executed on energy efficient yet unreliable hardware at runtime. The special value 1.0 is used for tasks that must be *unconditionally* considered as completely significant. For tasks with significance values in the range [0.0, 1.0) the run-time system partitions tasks into two bins as either the most-significant and the least-significant ones, depending on the significance values of tasks, the execution context, and the $ratio()$ user-supplied value that is discussed in Section 3.3.

Programmers explicitly specify data flow to the task through the *in()* and *out()* clauses. This information is exploited by the runtime to automatically determine the dependencies among tasks.

label() 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 3.3).

3.2.1 Approximate computing extensions to #pragma omp task

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 for a non-accurate computation of the task. It typically implements a simpler, approximate version of the computation, which may even degenerate to just setting default values to the output. If a task is selected by the runtime system to be executed approximately, and the programmer has not supplied an *approxfun* version, it is simply dropped by the runtime. It should be noted that the *approxfun* function implicitly takes the same arguments as the function implementing the accurate version of the task body.

3.2.2 Fault-tolerant computing extensions to #pragma omp task

Significant computations need to be executed correctly on reliable hardware, while non-significant ones may be executed on potentially unreliable hardware. During unreliable execution, errors which manifest on non-significant tasks should be controlled before they propagate to the rest of the computation. To this end, the developer can provide application-specific code for checking and repairing the output of tasks that are executed unreliably, using the *taskcheck()* clause.

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: (i) inspect the task status to see if it completed its execution normally or has crashed; (ii) assess whether the task output is wrong; (iii) assign meaningful default values to the task output; (iv) 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. Task re-execution is an acceptable error correction strategy only in the case of idempotent tasks. The most common error recovery method for non-idempotent codes is checkpointing and restarts. However, this incurs overhead – potentially significant – even in the case errors do not eventually manifest. Therefore, we opted to focus only on idempotent tasks and error recovery through re-execution.

When designing the programming model, we made the educated decision to organize computations in tasks with explicitly defined dataflow between tasks in the form of task arguments. Moreover, we decided task arguments to be unidirectional, either input (*in*) or output (*out*). We do not support bidirectional arguments (*inout*). Both those decisions implicitly promote task idempotence.

3.3 Synchronization

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

LISTING 3.2: #pragma omp taskwait

The programming model supports explicit barrier-type synchronization through the *#pragma omp taskwait* directive (Listing 3.2). 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 minimum quality 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.

3.3.1 Synchronization for approximate computing

The $ratio$ value is used to partition the tasks into two categories, the significant and the non-significant ones. In the case of approximate computing, the runtime system simply executes the approximate body of a task for all non-significant tasks instead of the fully accurate one. The *omp taskwait* barrier simply waits for the execution of all tasks.

3.3.2 Synchronization for fault-tolerant computing

Given that some of the non-significant tasks may be executed unreliably, $taskwait$ also allows for a more *relaxed* synchronization. Namely, the programmer can use the $time()$ clause to define a timeout after which execution will continue, provided that the most significant tasks (as per the $ratio()$ setting) have completed. If some non-significant tasks have not completed their execution yet, they are stopped, and the respective result-check functions are invoked (requests to re-execute a task are ignored). Note that such timeouts are task-dependent, as is the case in most soft real-time applications.

The programmer may introduce a result-check function for all tasks that have been created so far via the $groupcheck()$ clause. This function is called when the conditions of $taskwait$ are fulfilled, to perform checks and repairs on the aggregate output produced by the tasks.

3.3.3 Compiler implementation

The compiler for the programming model is implemented based on a source-to-source compiler infrastructure [108]. It recognizes the pragmas introduced by the programmer and lowers them to corresponding calls of the respective runtime system (supporting approximate or unreliable computing). Finally, the produced source code is compiled into machine code using the standard gcc tool chain.

3.4 Approximate computing example

```
1  int sblX(const unsigned char img[], int y, int x) {
2    return img[(y-1)*WIDTH+x-1] + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
3         - img[(y-1)*WIDTH+x+1]  - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
4  }
5
6  int sblX_appr(const unsigned char img[], int y, int x) {
7    return /* img[(y-1)*WIDTH+x-1]   Ommited taps */
8      + 2*img[y*WIDTH+x-1] + img[(y+1)*WIDTH+x-1]
9      /* - img[(y-1)*WIDTH+x+1]   Ommited taps *//
10     - 2*img[y*WIDTH+x+1] - img[(y+1)*WIDTH+x+1];
11 }
12 /* sblY and sblY_appr are similar */
13 void sbl_task(unsigned char res[], const unsigned char img[], int i) {
14   unsigned int p, j;
15
16   for (j=1; j<WIDTH-1; j++) {
17     p = sqrt(pow(sblX(img, i, j),2) + pow(sblY(img, i, j),2));
18     res[i*WIDTH + j] = (p > 255) ? 255 : p;
19   }
20 }
21
22 void sbl_task_appr(unsigned char res[], const unsigned char img[], int i) {
23   unsigned int p, j;
24
25   for (j=1; j<WIDTH-1; j++) {
26     /* abs instead of pow/sqrt,  approximate versions of sblX, sblY */
27     p = abs(sblX_appr(img, i, j) +  sblY_appr(img, i, j));
28     res[i*WIDTH + j] = (p > 255) ? 255 : p;
29   }
30 }
31
32 double sobel(unsigned char img[], unsigned char res[]) {
33   /*img, and res contain WIDTH*HEIGHT elements*/
34   int i;
35
36   for (i=1; i<HEIGHT-1; i++)
37     #pragma omp task label(sobel) in(img) out(res) \
38     significant((i%9 + 1)/10.0) approxfun(sbl_task_appr)
39       sbl_task(res, img, i); /* Compute a single output image row */
40   #pragma omp taskwait label(sobel) ratio(0.35)
41 }
```

LISTING 3.3: Programming model use case: Sobel filter

Listing 3.3 presents the implementation of a task of the *Sobel* filter. In lines 36- 39 a separate task is created to compute each row of the output image. The significance of the tasks ranges between 0.1 and 0.9 in a round-robin way (line 38), which ensures that there will not be extreme, apprehensible quality fluctuations across different areas of the output image. Care has also been taken in this case to avoid using the special value 1.0. Moreover, an approximate version of the task body is implemented by the *sbl_task_appr* function (lines 22–30). This function implements a light-weight version of the computation, substituting complex arithmetic operations with simpler ones (line 27), while at the same time skipping some filter taps (lines 7, 9). All tasks created in the specific loop belong to the *sobel* task group, using *img* as input and *res* as output (line 37).

```
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 3.4: Programming model use case: DCT pseudo-code

3.5 Fault-tolerant computing example

Listing 3.4 presents a task based implementation of *DCT* using our programming model for fault-tolerance. 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 suspicious to be 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 3.4, 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.

3.6 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. Unlike parallelism however, the programmer is

not required to be familiar with the platform architecture to efficiently specify and exploit task significance.

For any C code which implements $y = f(x)$ the significance of intermediate values x with respect to the output y can be defined using the interval arithmetic and first order adjoint analysis. The significance of x with respect to y is the width of the multiplication:

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

Intuitively, $S_y(x)$ defines the value ranges of y given a specific value range for x. If the range (width) of S_y is large then x highly affects the output value y. As such, computing the value x is highly significant with respect to the accuracy of the final output y. We have provided detailed background information regarding the mathematical definition of significance in Section 2.2.1. Moreover, we present our approach to performing automatic significance analysis of application code in Chapter 4.

In the case of fault-tolerant computing, beyond the tagging of tasks with significance information, 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. We present our approach to automating the process error detection through the use of Artificial Neural Networks in Chapter 6.

Last but not least, the granularity of tasks is another 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.

Coarser task size granularity alleviates the overhead of the runtime to manage and schedule tasks. On the other hand, larger tasks may reduce the number of significance levels, resulting into otherwise non-significant computation to be annotated as significant, thus missing opportunities for aggressive energy optimizations.

3.7 Application Characteristics

As discussed in the introduction, several application domains offer the opportunity to trade-off quality of output for significant improvements in energy consumption. Such applications domains include:

Visualization applications: In our evaluation we use two benchmarks from this category: *DCT* and *Sobel*.

Streaming applications: *Blackscholes*, one of the benchmarks used in the experimental evaluation, falls in this category.

Iterative methods: We include *Jacobi* and *K-means* in our evaluation.

Randomized algorithms: We use *MC*, a Monte Carlo PDE solver.

We wish to clarify that the proposed significance-based computing model does not fit all applications. For instance, task 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 tasks to be executed without any inaccuracy or any chance of data corruption.

Finally, significance-based computing may be impractical for applications for which all parts of the code should be executed precisely to produce a meaningful output, or when task significance is input dependent and cannot be easily characterized by the programmer. Transactional-style applications in which the final output is dependent on the correct execution of a sequence of decision making tasks is such an example.

In the previous sections we discussed the design of a versatile significance aware programming model which can facilitate the implementation of approximate and fault tolerant computing applications through its extensions. In the remaining of this chapter we will present the accompanying significance aware runtimes for approximate and fault tolerant computing.

3.8 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 [96].

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

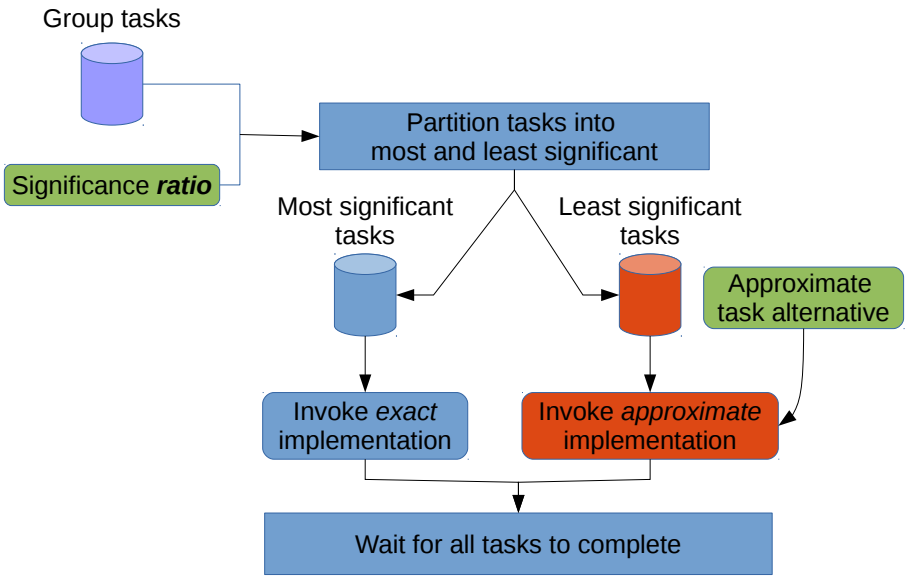


FIGURE 3.1: The typical life of a group-of-tasks in the context of significance aware approximate computing

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 [98]. *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.

3.8.1 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 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.

3.8.2 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

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
Kmeans	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).

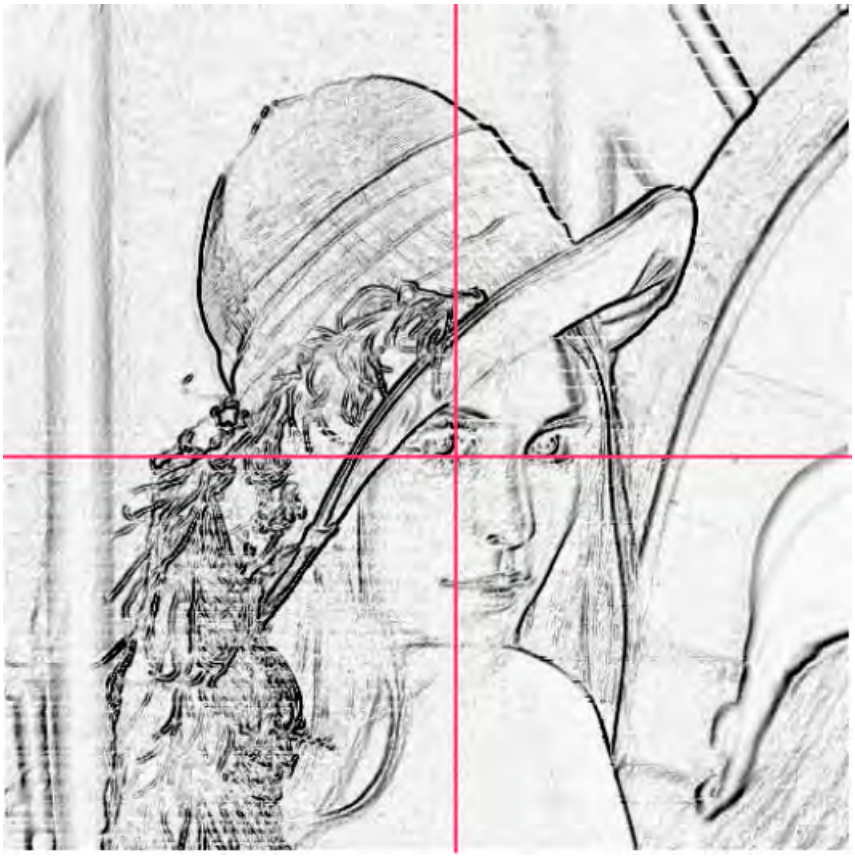


FIGURE 3.2: Different levels of approximation for the Sobel benchmark

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 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.

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) [102].

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. 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, Kmeans, 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 [90], 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 [95] to access the Running Average Power Limit (RAPL) registers of the processors.

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 *Kmeans*.

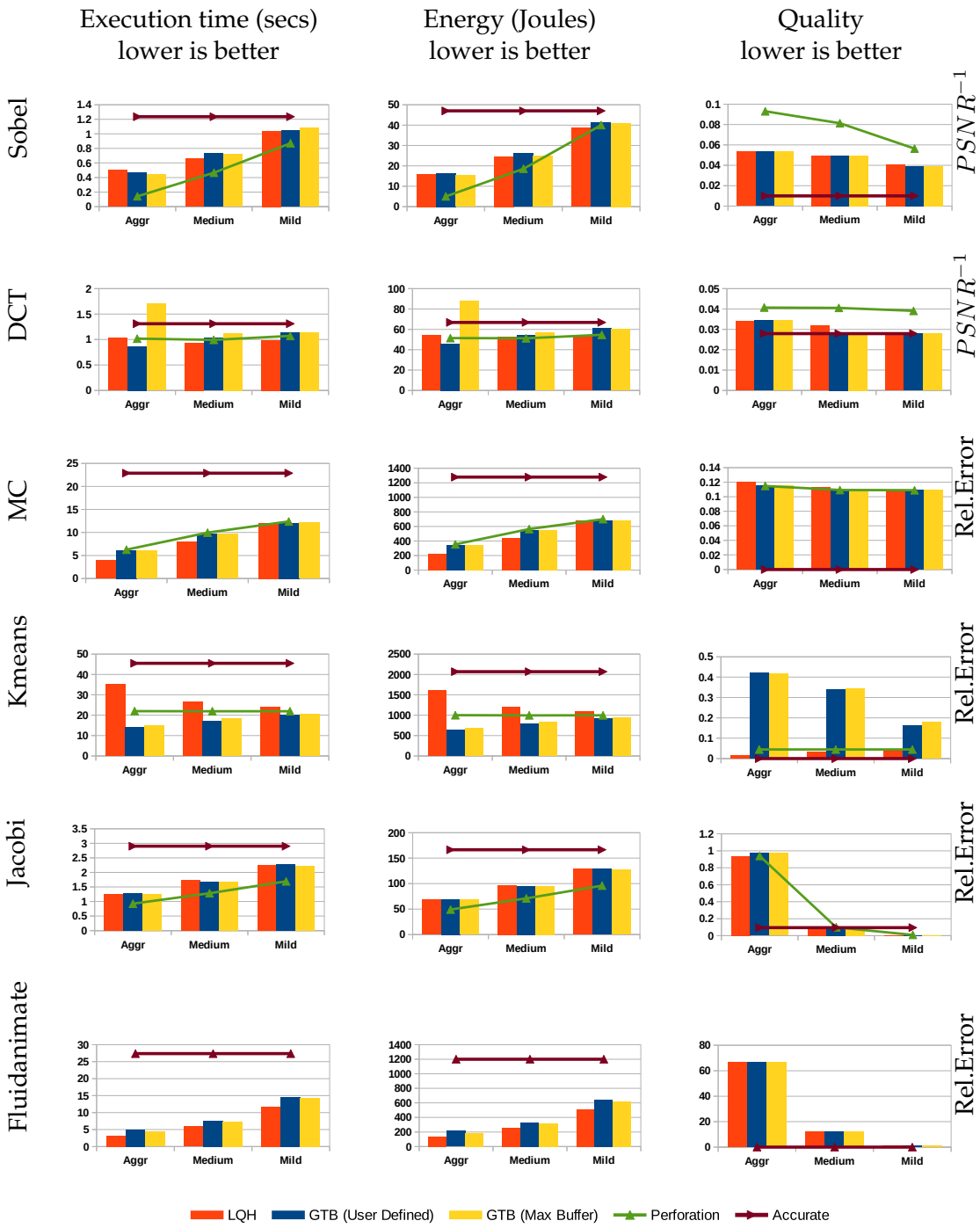


FIGURE 3.3: Execution time, energy and quality of results for the benchmarks used in the experimental evaluation under different runtime policies and degrees of approximation. In all cases lower is better. Quality is depicted as PSNR^{-1} for Sobel and DCT, relative error (%) is used in all others benchmarks. The accurate execution and the approximate execution using perforation are visualized as lines. Note that perforation was not applicable for Fluidanimate.

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

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.

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.

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 terms of performance and energy consumption, yet resulting to higher quality results[1]. 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

[1]Note that PSNR is a logarithmic metric

slightly better results. In this case, we found that the LQH policy undershoots the requested ratio, evidently executing fewer tasks [2]. This affects quality, which is lower than that achieved by the rest of the policies.

Kmeans 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 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 3.8.2, 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.

Following, 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

[2] 4.6% and 5.1% more that requested tasks are approximated for the aggressive and the medium case respectively.

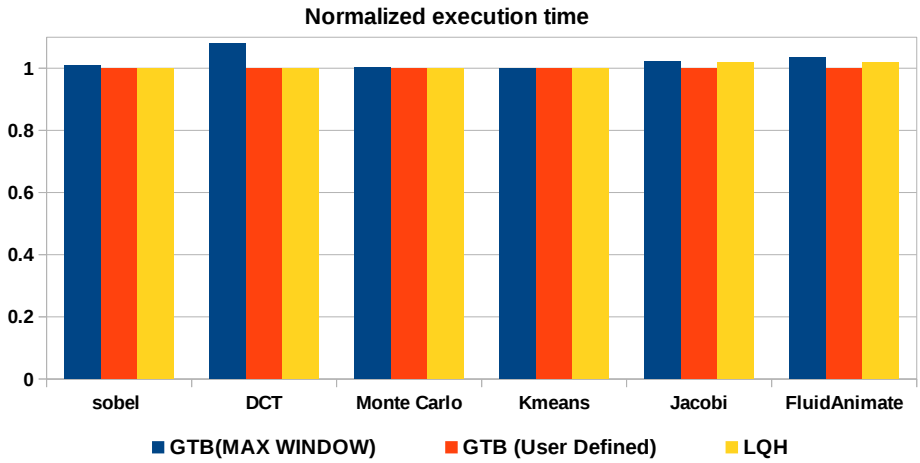


FIGURE 3.5: The normalized execution time of benchmarks under different task categorization policies, with respect to that over the significance-agnostic runtime system

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.

3.9 Runtime support for significance aware fault tolerant computing

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 due to undervolting the core part.

3.9.1 Runtime Execution Management

As we discussed in Section 2.3.3, we consider three different configurations, $FastRel$, $SlowRel$ and $FastUnRel$. The $FastRel$ configuration is a high-performance nominal point of operation, with high voltage/frequency (V_h, f_h), where a core executes

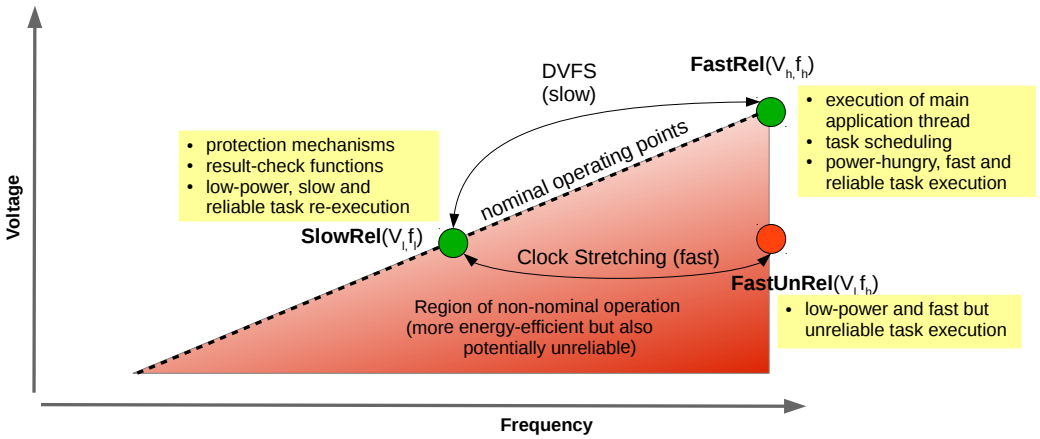


FIGURE 3.6: The configurations $FastRel$, $SlowRel$ and $FastUnRel$ used by the runtime system, to reduce the energy footprint by exploiting the significance of computations. Our approach exploits non-nominal configurations, that are energy-efficient but unreliable.

code fast, whereas $SlowRel$ is a slower nominal operation point, with lower voltage/frequency (V_l, f_l). Furthermore, cores can be set in the non-nominal and unsafe $FastUnRel$ configuration (V_l, f_h), 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 time, execution is potentially unreliable due to timing faults, since $FastUnRel$ is outside the nominal range 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. Figure 3.6 illustrates the principle of execution. 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[3]. Afterwards, the main thread requests the memory allocator 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 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

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

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. Consequently, the total execution time of the application is computed using the components of Equation 2.17. It 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 (Equation 3.9.1).

$$T_{Total} = \max_{i=1}^{Workers} \left(T_{FastRel_i}\right) + \max_{i=1}^{Workers} \left(T_{SlowRel_i} + T_{FastUnRel_i}\right) + T_{vfs} \tag{3.1}$$

The runtime supports the following levels of protection:

No Protection (NP): The runtime system does not employ any error detection/-correction mechanism or significance information supplied by the programmer. 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 result-check function supplied by the programmer 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.

3.9.2 Life of a group-of-tasks

Figure 3.7 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, a desired approximation level in the form of a taskwait $ratio$. 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, whereas the runtime system schedules the least significant ones on hardware that is operating unreliably. 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 a more elaborate 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.

3.9.3 Benchmarks

We use five benchmarks, listed in Table 3.2, 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.

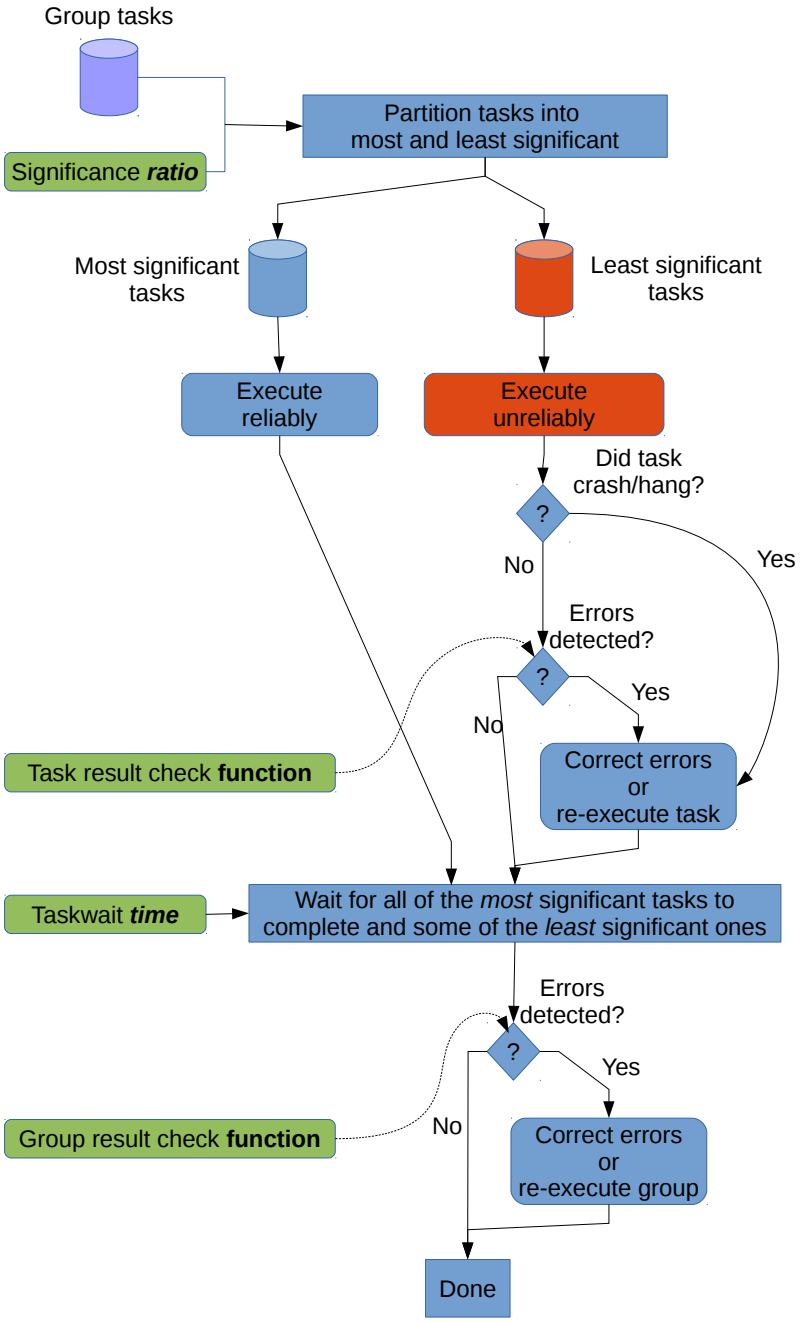


FIGURE 3.7: The typical life of a group-of-tasks in the context of significance aware fault tolerant computing

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

Benchmark	Domain	Sgnf. Characterization	Lines of Code	
			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 3.2: 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.

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.

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; 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 3.2 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

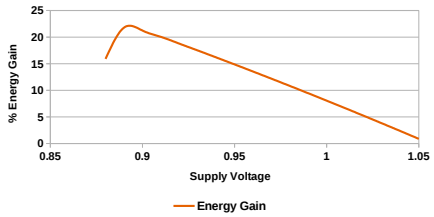


FIGURE 3.8: Energy gains of a single task for *Sobel* executed at voltages $V_l < V_h$ for constant frequency $f_h = 3.7GHz$.

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

TABLE 3.3: $SlowRel$ and $FastUnRel$ configuration settings used in our evaluation, and average fault rates of the $FastUnRel$ configurations.

function was minimal.

3.9.4 Simulated Software Fault Injection

We evaluated our significance-centric framework on top of a Intel Quad Core i7 Ivy-Bridge CPU platform. Our x86 platform does not allow placing individual cores in a non-nominal configuration, where actual timing violations and faults might occur. Thus, we rely on software fault injection to simulate the manifestation of errors during the unreliable execution of selected tasks of a program. We provide a more detailed discussion regarding our software fault injection methodology in Section 2.3.

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%.

We study the behavior of benchmarks for the different protection levels supported by our runtime system (Section 3.9.1), on our i7 4820K CPU which is 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 3.8 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 $0.89V$. If we further undervolt, inducing faults at higher rates, tasks are practically certain to crash. 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 3.3 summarizes the configurations used in our experiments.

Figure 3.9 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 to checkpointing, which costs approximately 2000 cycles. Noticeably, in *Sobel* and *Blackscholes* the overhead of correction is comparable with 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.

Figure 3.10 summarizes our experimental results for a range of voltage settings and protection mechanisms. For each benchmark we present two diagrams. The 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

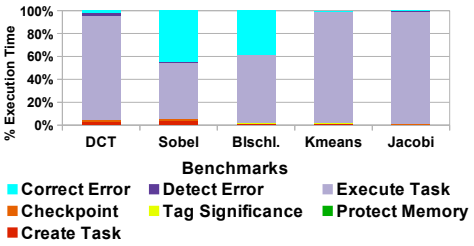


FIGURE 3.9: Breakdown of task execution time, for each benchmark.

Bench.	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 3.4: Average task execution time in cycles (thousands), number of tasks executed reliably/unreliably, and number of voltage and frequency transitions, for each benchmark.

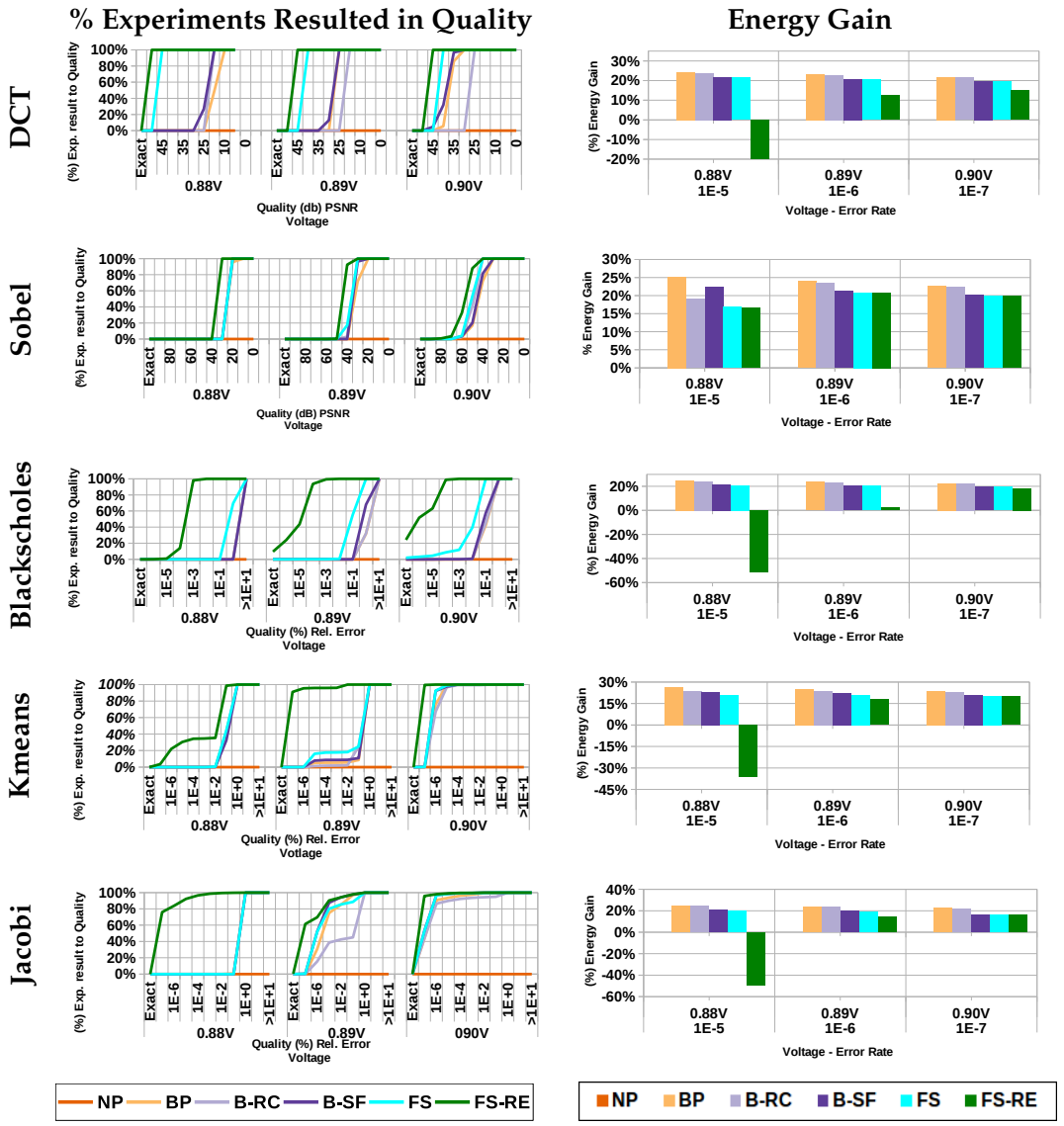


FIGURE 3.10: Experimental results for different V_l values for the *SlowRel* and *FastUnRel* configurations. Percentage of experiments which achieved a certain quality (left), and energy gains with each protection scenario (right).

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 3.4. 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 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, for the specific evaluation 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. A better solution is discussed in Section 6. Another interesting observation is that task re-execution (*FS-RE*) does not guarantee perfect results, as is clearly visible from the CDFs in Figure 3.10. 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)* does 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 fails to detect 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 3.11 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 3.4), 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.

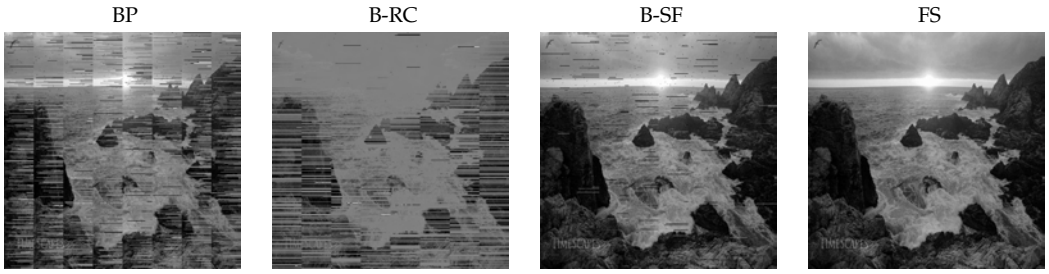


FIGURE 3.11: *DCT* output at 0.89V, with one fault injected every 100,000 cycles. The images correspond (from left to right) to the *BP*, *B-RC*, *B-SF* and *FS* protection configurations, resulting to PSNRs of 12, 13, 15 and 37 dB respectively. A fault free execution leads results in a PSNR of 43 dB. *NP* deterministically leads to crashes.

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 faulty output. For *Sobel*, an approximate version of the task is executed. Therefore, the energy gains due to undervolting are outweighed 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 their definition range. Detecting such errors is easy. Since many faults 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 error rates faults are frequently detected and therefore the respective iterations are discarded. In *Jacobi* it is better to rely on the self healing attributes rather than correcting the result.

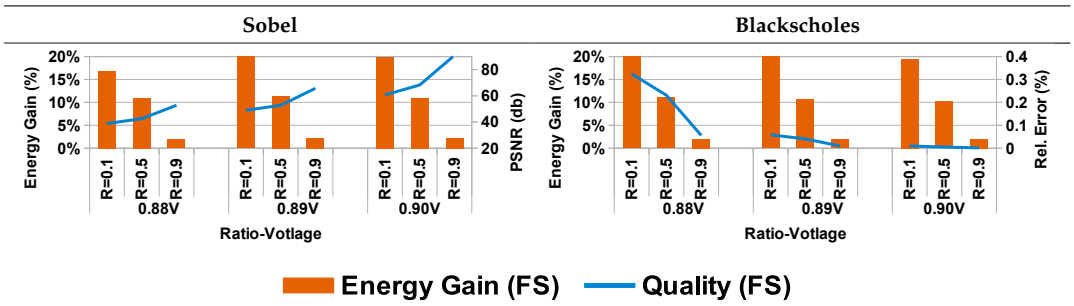


FIGURE 3.12: Quality vs. energy trade-offs using the $ratio$ parameter in the FS configuration.

Figure 3.12 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.

Interestingly, modern processors, with the assistance of our framework can produce acceptable results until they reach the Point of First Failure. Below that point, both additional energy gains are too low and massive failure rates defeat any software-based realistic protection mechanism. More importantly, our results indicate that the energy gains are maximized when we schedule tasks on either reliable or unreliable hardware depending on their significance values. In this scenario we instruct the runtime system to exploit two types of heterogeneity. The first being hardware heterogeneity through the use of mixed-reliability configuration domains, and software heterogeneity by means of exploiting the algorithmic property of significance.

Chapter 4

Automating significance characterization of tasks

In the previous chapter we introduced our basic framework to support the development and optimize the execution of applications which employ the principles of significance-aware approximate and fault-tolerant computing. In this chapter we will provide a methodology to address three key challenges of the proposed computing paradigms. More specifically, we will discuss our approach to answering the questions of **What**/**How** to approximate? as well as **What** to execute unreliably? To this end, we make the following contributions:

(i) We introduce a framework to exploit `dco/scorpio` (section 2.2.3), in order to perform automatic significance analysis. More specifically, `dco/scorpio` employs interval analysis [66] and algorithmic differentiation [26, 67] to automatically quantify the significance of computations. Our framework processes the outputs of `dco/scorpio` to detect variations in significance among parts of code so that it may:

- Characterize tasks with significance information, which qualitatively describes the task's impact to the final output quality.
- Provide hints to application developers in order for them to implement lightweight approximate alternatives for tasks.

(ii) We evaluate our approach and its impact on the quality/accuracy of the end result, on a set of five benchmarks from the domains of imaging, finance and physics.

(iii) We compare our approach with loop perforation [90], a compiler technique for skipping selected loop iterations.

Our results show that it is possible to perform automatic significance analysis and produce similar results to a human domain expert. We also report that it is possible to exploit the algorithmic property of significance to gracefully trade-off application output quality with optimized execution of applications. In fact, our case study resulted in better results in comparison with blindly applying loop perforation.

4.1 Workflow for Systematic Significance Driven Programming

In this section we show how significance analysis is systematically used to guide the programmer to expose significance information at the level of a task through source code annotation.

4.1.1 Significance Analysis Framework

Application developers traditionally focus on performance related concerns such as parallelism, synchronization, load balancing, scheduling, and cache utilization when designing the structure of their code. We observe that this approach may conflict with the optimal application structure when the primary concern is the exploitation of the algorithmic property of computational significance. To this end, we explore the use of `dco/scorpio` as the basis for a framework which assists developers in partitioning the computation to tasks, and at the same time it reveals opportunities for approximation within the body of a task.

`dco/scorpio` performs significance analysis to estimate the impact of intermediate results to the output quality. This information, however, in its raw form is not easily usable by a human. Recall that in section 2.2.3 we discussed the functionality of `dco/scorpio`. The analysis of `dco/scorpio` produces an annotated Dynamic Data Flow Graph (*DynDFG*). Nodes within the *DynDFG* represent the computation of a single intermediate value. Unfortunately, it is often the case for `dco/scorpio` to produce extremely large *DynDFGs*.

Our methodology processes a *DynDFG* that was the output of `dco/scorpio` and produces a *DynDFG* of reduced complexity. It then identifies sibling nodes within the new *DynDFG* whose significance varies significantly. Those points in the *DynDFG* are good candidates for task boundaries. This information may then be used by application developers to structure their code using our significance-aware programming model. In turn, the runtime support system will take advantage of the opportunities for program optimization that significance-aware approximate computing creates. In addition to task partitioning, application developers may use the results of the analysis to implement lightweight alternatives of tasks. The remainder of this section discusses the details of our methodology (*Algorithm 1*).

We will be using a Taylor series computation (*Listing 4.1*) as a toy running example to illustrate all aspects of the workflow:

$$f(x) = \sum_{i=0}^{n} x^i \approx \frac{1}{1-x}, x \in (-1, 1). \tag{4.1}$$

We use the notation introduced in section 2.2.1. Nodes at the top are mapped to input vector $\vec{x}$, leaf nodes at the bottom to the output vector $\vec{y}$ and the remaining nodes correspond to intermediate variables u_j. Starting from the original source

```
1  double maclaurin_series(double x, int N)
2  {
3    double result = 0.0;
4
5    for (int i=0; i<N; ++i )
6    {
7      double term = pow(x, i);
8      result += term;
9    }
10
11   return result;
12 }
```

LISTING 4.1: The Taylor Series original implementation.

code of the application, we produce a simplified *DynDFG* along with significance information for all nodes. An example *DynDFG* is shown in *Figure 4.1*.

Steps *S1* and *S2* identify the output and input data of the algorithm, respectively, and register the input data ranges. To annotate Taylor Series (*Listing 4.2*) we register $\vec{x}$ as the input data and set its value width equal to 1 ($\vec{x} \in [x - 0.5, x + 0.5]$ as seen

Algorithm 1: Significance Analysis Framework (`dco/scorpio`)

Input : Application source code
Output: G_{out} (*DynDFG*) along with significance tags

S1: $\vec{y} = (y_0, \ldots, y_{m-1})^T$
S2: $\vec{x} = (x_0, \ldots, x_{n-1})^T$
S3: $G = \texttt{dco/scorpio}(\vec{x}, \vec{y})$
S4: $G_s = simplify(G)$
S5: $G_{out} = findSgnfVariance(G_s)$

```
def findSgnfVariance(DynDFG G) {
  for ( L = 1; L<G.height; ++L )
    if ( SgnfVariance(L) >δ )
      call G.removeAbove(L+1)
      break
  return G
}
def simplify(DynDFG G) {
  for (L=0; L<G.height; ++L)
    nodes = G[L]
    foreach( v in nodes)
      inputs = v.InputDeps()
      call simplifyDep(G, inputs, v)
  return G
}
def simplifyDep(DynDFG G, nodes, parent) {
  foreach (v in nodes)
    if( v.AntiDependent(parent) )
      v.Parent = parent
      call simplifyDep(G, v.InputDeps(), parent)
    else
      call G.SetDependency(v, parent)
      call simplifyDep(G, v.InputDeps(), v)
}
```

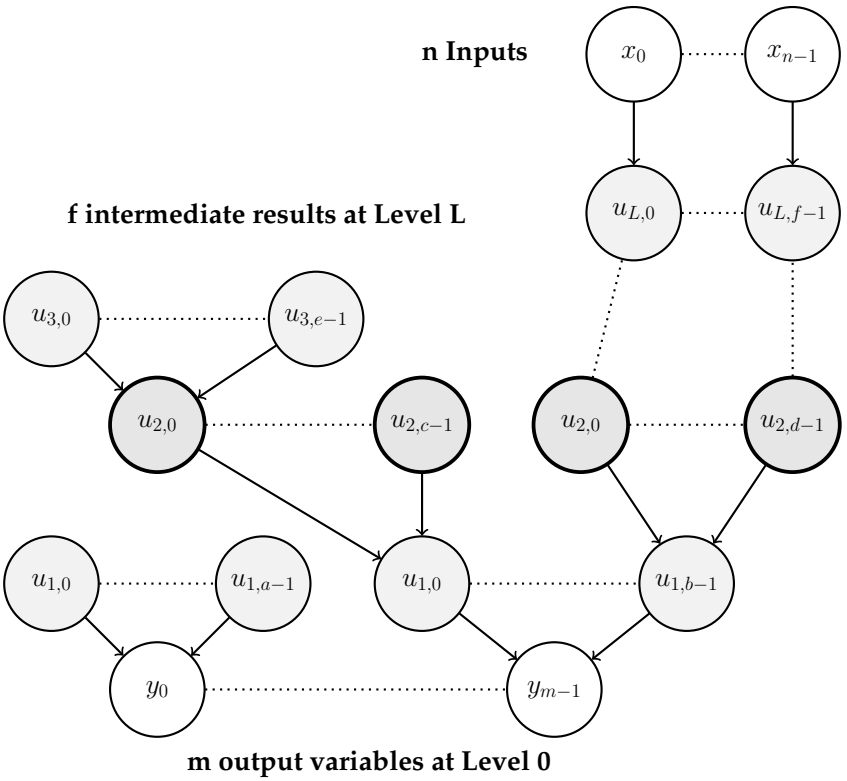


FIGURE 4.1: Dynamic DFG (DynDFG) of the application

```
1  double maclaurin_series(dco::ia1s::type x, int N)
2  {
3    dco::ia1s::type result = 0.0;
4
5    INPUT(x, x-0.5, x+0.5);
6    for (int i=0; i<N; ++i )
7    {
8      dco::ia1s::type term = pow(x, i);
9      result = result + term;
10   }
11
12   OUTPUT(result);
13   ANALYSE();
14
15   return result.toDouble();
16 }
```

LISTING 4.2: Listing 4.1 enhanced with `dco/scorpio` macros.

in line 5). Step *S3* invokes the `dco/scorpio` analysis toolset described in section 2.2.3 to produce a graph following the format shown in *Figure 4.1*. Each node u_j is annotated with the significance value of the corresponding intermediate variable to the output.

Step *S4* post-processes the graph produced by the significance analysis tool to eliminate internal nodes that express anti-dependencies such as: $res = res + term[i]$. These operations aggregate results, and are not really part of the computation. We illustrate these aggregation nodes using a darker color in *Figure 4.2*. Disregarding them is important for the next step *S5*, which is the main step of Algorithm 1.

Step *S5* traverses G_s using Breadth First Search (BFS), starting from the output nodes at level $L = 0$ and moving towards the input nodes, to construct G_{out}. The algorithm detects the level L at which nodes have significance values with statistical variance higher than δ.

Intuitively, when we detect nodes with high statistical variance in their significance values, we have reached a level in the *DynDFG* which can be used to partition the code into tasks of different significance. Nodes $u_{L,k}$ with high significance for the program outputs $\vec{y}$ can be made to correspond to output variables of significant tasks, and the programmer can restructure the code around this information. On the other hand, if the algorithm terminates at the inputs $\vec{x}$ of the code without detecting any significance variations, it is guaranteed that nodes which reside in the same level are (almost) equally important. Parameter δ is dependent on application characteristics and the sensitivity to significance variations required by the programmer. When the analysis terminates, the method produces a *DynDFG* containing nodes up to level $L + 1$, along with their significance value.

In *Figure 4.2* we show the *DynDFGs* produced in steps *S3* and *S4* respectively for the Taylor Series. *Figure 4.2b* is the result of *S5* as well. The last step of our analysis terminates at $L = 1$ since there are large variations between node significances at this level. Note that, the first term has a significance of 0. A significance value of 0 means that the respective computation can be substituted by a constant value, which is 1 in this case since $pow(x, 0) = 1$. The most significant term is the second one and every term computed afterwards is less significant than the one before it.

Figure 4.3 illustrates the boundaries of tasks for the Taylor benchmark. Notice that a single task computes multiple terms and each term has a different significance value. Given that it is neither intuitive nor possible for a task to carry multiple significance values, we use a simple rule to decide the significance of a task that computes multiple output elements: A task is as significant as its most significant output. As a result **Task 1** is tagged with the significance value of **term1** and **Task 2** carries the significance value of **term3**. Finally, note that the the computation of the very first term, as the significance analysis results suggested, is substituted by a constant value.

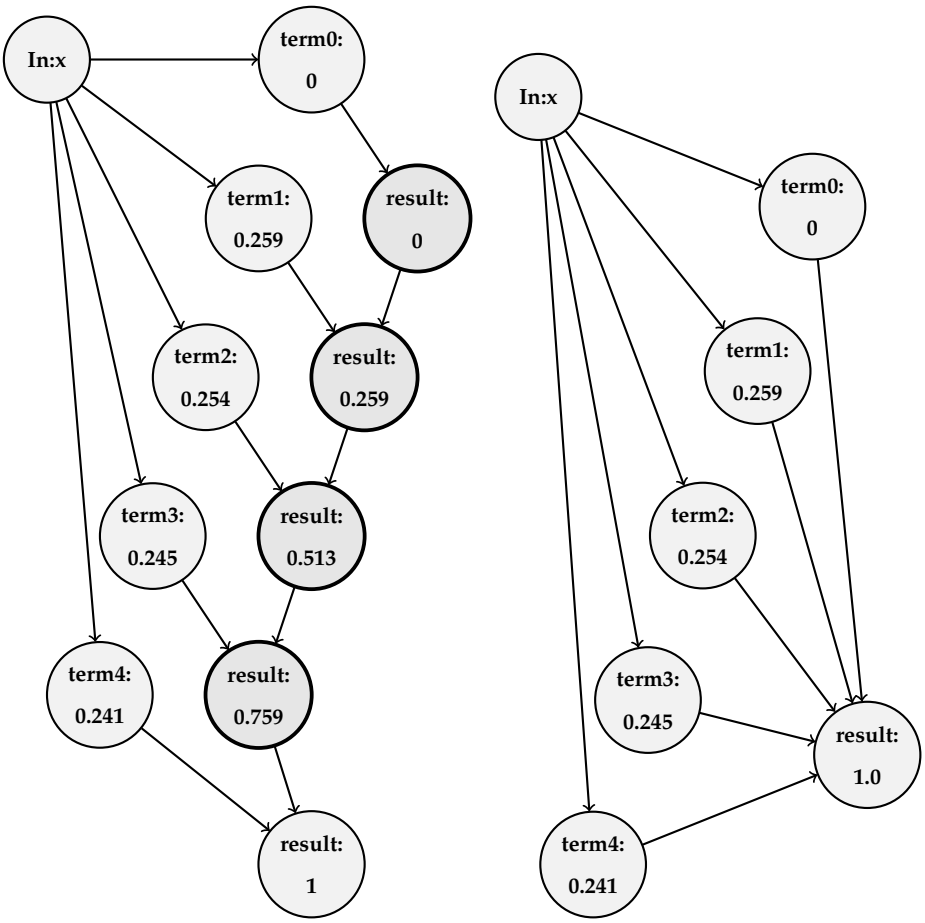


FIGURE 4.2: Figure (a) illustrates the Graph containing the significance values of the elemental computations as produced by `dco/scorpio` during *S3* and (b) The simplified graph after *S4* for the Taylor Series example.

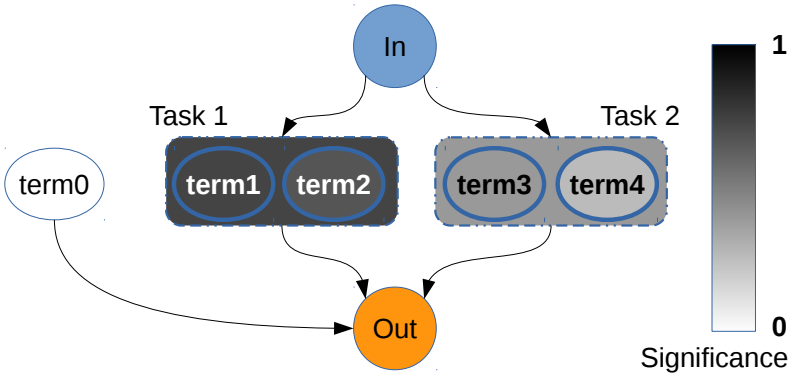


FIGURE 4.3: Task boundaries for the toy benchmark Taylor.

4.2 Experimental Evaluation

We evaluate our approach with five applications, consisting of six different computational kernels. We intentionally include well-known kernels in our benchmark set, so that we can exploit domain expertise to validate the results of the significance analysis.

4.2.1 Method validation

Sobel

For the significance analysis of Sobel we use a set of images used in image compression benchmarking [6]. The analysis indicates that the first level of high variance between the significance of computations in the *DynDFG* is the one at which convolutions take place. Three distinct blocks of computations are identified. The first one (A) uses the filter coefficients 2 and -2 and the other two (B, and C) use 1 and -1. The analysis shows that A is twice as significant as the other two. Finally, the computations which aggregate the outputs of convolutions and produce output pixels show little significance variance across all pixels.

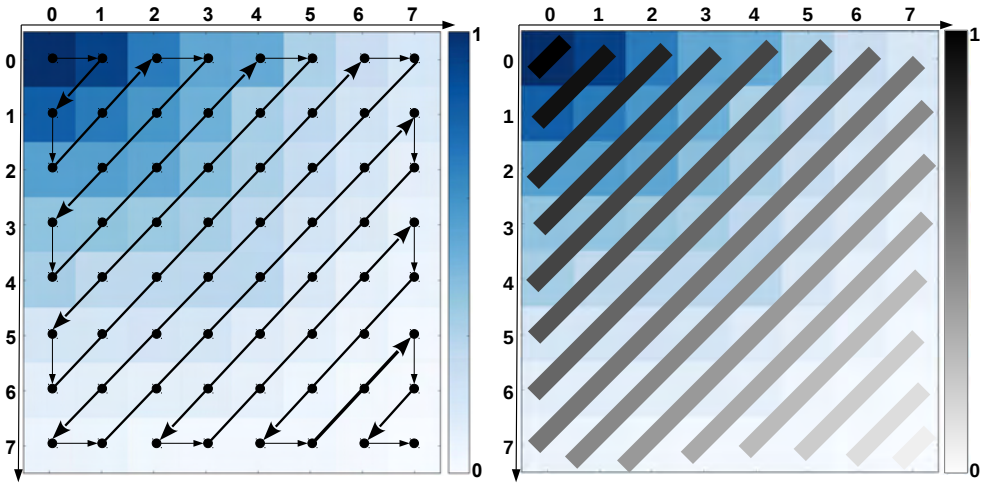
Discrete Cosine Transformation

Analysis reveals a variation in significance at level $L = 1$ of the *DynDFG*, which corresponds to the computation of individual frequency coefficients. We perform the analysis on code which performs the invocation of DCT, quantization, de-quantization of the DCT coefficients and finally inverse-DCT to decompress the image. The resulting coefficient significances are shown in *Figure 4.4a*. The diagonal zig-zag path corresponds to the importance of coefficients according to the wisdom of image/video compression experts. The significance pattern that emerges from the analysis verifies this domain expert wisdom, thereby validating our approach. In turn, we structure DCT using 15 tasks in total as shown in *Figure 4.4b*. As a result, each task operates on coefficients of the same or similar significance. Task significance gradually drops with increasing distance from the top-left corner. We approximate the execution of the least significant tasks using a default value: we set their outputs to zero.

Note that the significance analysis results of DCT indicate that there is a trade-off between tasks that produce outputs of similar significance and work-balancing. Indeed, we can observe that there are conflicting concerns when implementing applications for significance-aware computing. Consequently, in the future we plan to investigate the relationship of significance and task-granularity as well as their effects on application performance and output quality (see Section 8.3).

Fisheye Lens Image Correction (Fisheye)

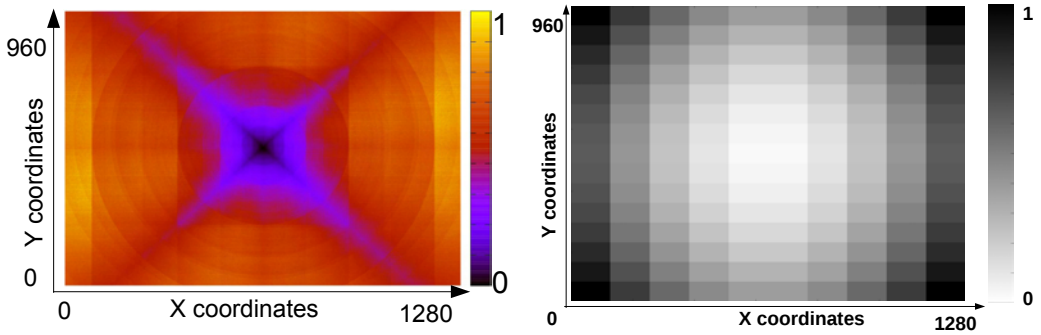
Figure 4.5a depicts the output of the significance analysis for the *InverseMapping* kernel applied on an image the dimensions of which in the natural looking space are



(A) The DCT significance mapped on the 8x8 block of DCT coefficients. The top left corner has the highest value and drops in a wave-like pattern towards the opposite corner.

(B) We partition the computation of DCT into 15 tasks. Each task computes one diagonal of all the 8x8 blocks of coefficients.

FIGURE 4.4: Significance analysis for DCT and task boundaries. Note that in both figures, the darker the color the higher the significance value.



(A) Significance values of the *InverseMapping* kernel.

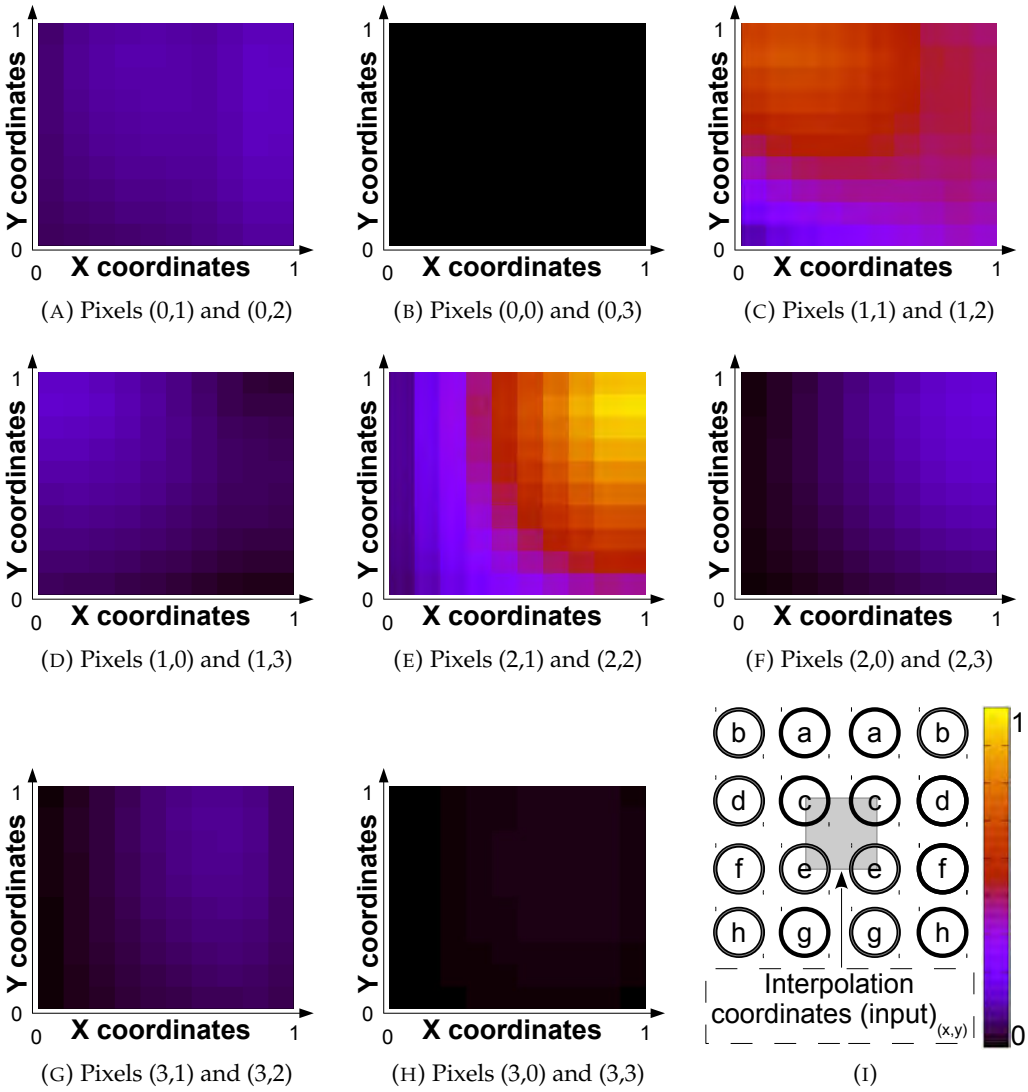
(B) Task boundaries and their respective significance.

FIGURE 4.5: Significance analysis for Fisheye and the resulting task boundaries.

1280x960. The effect of fisheye-shaped lens is to expand the pixels closest to the boundary, and push together pixels that are near the center. Thus, computing coordinates for pixels near the border is more sensitive to imprecision than for those at the center.

BicubicInterp uses weighted averages to produce the interpolated pixel value. The grey rectangle of *Figure 4.6i* shows the area in which the interpolated pixel resides. *Figures 4.6a-4.6h* show the corresponding significance values of the pixel-pairs, mapped on the discretized input coordinate space. The results indicate that the inner 2x2 pixel block, which directly surrounds the coordinates of the input point, contains the two most significant pairs of pixels (*Figures 4.6 c* and *e*).

We use the significance pattern of *InverseMapping* to assign higher significance



(A) Pixels (0,1) and (0,2) (B) Pixels (0,0) and (0,3) (C) Pixels (1,1) and (1,2)

(D) Pixels (1,0) and (1,3) (E) Pixels (2,1) and (2,2) (F) Pixels (2,0) and (2,3)

(G) Pixels (3,1) and (3,2) (H) Pixels (3,0) and (3,3) (I)

FIGURE 4.6: Significance graphs for the pixels in the $4x4$ block of *BicubicInterp* with respect to the interpolated output image; letters in (i) point to the corresponding graphs.

values to tasks which are closer to the image border, and lower to those near the center as illustrated in *Figure 4.5b*. The approximate version of tasks invokes *InverseMapping* only for the pixels which lie on the border of the 128x64 block and uses interpolation to compute the coordinates of internal pixels. For *BicubicInterp* we exploit a transitive property of significance: it is sensible to opt for an approximate execution of computations which use approximate input data. In tasks where *InverseMapping* was executed approximately, *BicubicInterp* uses only the pixels pairs in *Figures 4.6 c* and *e*.

N-Body

We compute the significance of each atom's state with respect to the state of all other atoms. The results, once again, confirm domain expert wisdom: the significance is strongly correlated with the distance between atoms. The greater the distance between atom A and atom B, the less the kinematic properties of one affect the other.

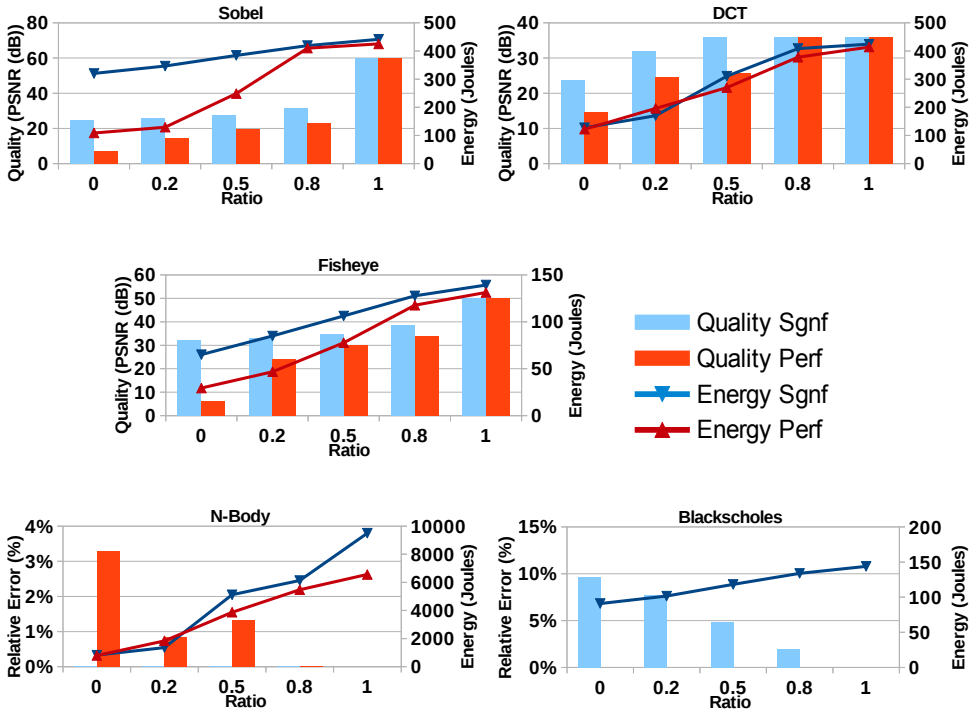


FIGURE 4.7: Output quality (blue bars, left y-axis) and energy consumption (blue lines, right y-axis) for the 5 benchmarks, as a function of the ratio of accurately executed tasks (x-axis). The results obtained by loop perforation are depicted in red.

We assign significance values to tasks based on the results of the significance analysis. As such, tasks which compute forces between neighboring atoms receive higher significance values than those involving distant atoms. The least significant tasks are dropped.

Blackscholes

Significance analysis indicates that the computation of a stock price can be broken down to 4 blocks of code A, B, C, D, with $sig(A) > sig(B) \gg sig(C) > sig(D)$. Each block produces the input to its successor block. Block C performs two Cumulative Normal Distribution Functions (CNDFs) and block D produces the final output by combining the outputs of C. The two least important parts (C and D) are approximated using less accurate but faster implementations of mathematical functions such as exp and $sqrt$ [61].

4.2.2 Performance evaluation

Loop perforation

As a reference, we use versions of the benchmarks which apply loop-perforation [90] to trade-off energy consumption with output quality. We perforate the loops in such a way that the same percentage of computations is skipped as the percentage of

computations approximated by our runtime. Similarly to [90], we find that loop perforation is not applicable on Blackscholes since the application does not have any loops within the computation of a stock's price. The perforated version of Sobel skips the computation for a percentage of the rows of the image. In DCT we perforate the double nested loops which compute the coefficients of an 8x8 block of pixels. In Fisheye we drop the computation of some of the output image rows similarly to Sobel. Finally, the original version of N-body computes the forces affecting a particle by iterating all other particles in a loop, whereas the perforated version skips some iterations of the loop.

Quality, performance and energy quantification

We execute all applications for different degrees of approximation, varying the ratio of tasks that are executed accurately. The blue-colored bars in *Figure 4.7* show the effects on output quality. For Sobel, DCT, and Fisheye we use Peak Signal to Noise Ratio (PSNR) with respect to the fully accurate execution as a quality metric (higher is better). Note that, PSNR is a logarithmic metric. For N-body and Blackscholes we evaluate the relative error (lower is better) with respect to the fully accurate execution.

The quality of the output gradually increases with the number of tasks that are executed accurately, in all benchmarks. This shows that the significance-driven approach can indeed lead to well-behaved approximate applications. Note that DCT and N-body produce high-quality output even for relatively low accurate task ratios. The more distinct the significance properties of an algorithm are, the smaller the quality penalty due to approximation.

We measure performance and energy consumption for an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz CPU with 14 cores and 128 GBs of RAM. The blue-colored lines in *Figure 4.7* show the energy cost of approximate executions (execution times are not shown here for brevity, they follow the same pattern). The energy for a fully accurate execution of each application corresponds to the rightmost data point of the respective plot. The results of loop-perforation correspond to the red-colored bars and lines in *Figure 4.7*.

Executing more tasks in (light-weight) approximate mode results in lower energy consumption, as expected. In some applications (Sobel, Fisheye) perforated versions are more energy efficient that the corresponding significance-based ones due to the overhead of our task-based implementation. However, approximations driven by the analysis results lead to higher quality of results compared with perforation at the same ratio of accurate computations. This effect is more profound for DCT, Fisheye and N-body. This difference in quality significantly outweighs overheads of the task-based implementation. For example in N-body the significance-based approximation achieves a relative error of 0.006% already with a fully approximate execution, at an energy cost of 820 Joules. The perforated version requires 80% of the

iterations to be executed accurately to achieve a relative error of 0.02%, at an energy budget of 5475 Joules. The same effect is observed in Sobel.

Our methodology results in better quality for all benchmarks compared with loop-perforation. On average, Sobel, DCT, and Fisheye produce images with 3.91 dB, 10.96 dB, and 6.9 dB higher PSNR compared to their perforated versions. Similarly, N-body produces relative errors which are on average 6 orders of magnitude lower than those introduced by the perforated version.

Chapter 5

Modeling and Prediction of Application Energy Footprint

In the previous chapter we answered two key question to approximate/fault-tolerant computing (**What** and **How** to approximate?). This leaves us with the question of **When** to actually perform the approximate computations in place of the exact/accurate ones. In our programming model the controlling knob for approximation is the *ratio* clause of taskwait. The lower its value the higher the degree of approximation. It is the job of the programmer or the end-user to decide the value for the *ratio* knob. Unfortunately, this is not a straightforward undertaking. Note that, even though in this Chapter we focus on approximate computing, our methodology can be applied to fault-tolerant computing with minimal modifications.

(i) We introduce an analytic model to predict the energy consumption of an application under different input sizes and execution configurations in terms of number of cores used, processor frequency, and the mix of accurately and approximately executed tasks.

(ii) During execution time, an intelligent runtime system uses the analytic model to choose the appropriate configuration so that the application respects a user specified energy budget.

(iii) We evaluate our approach and its impact on the quality/accuracy of the end result, on a set of nine benchmarks.

(iv) We compare our approach with loop perforation [90], a compiler technique for skipping iterations that are deemed less significant for the output quality while keeping critical loop iterations that must always be executed. We also compare our approach to an Oracle configurator who is always selecting the optimal configuration.

We report that for 7 out of 9 benchmarks our analytical model enables the runtime support system to select configurations which are very close to the optimal ones. We also show that, in most cases, executions perform better in terms of output quality when using significance aware approximation instead of blind perforation.

Next we describe our modeling and prediction approach in more detail. As an underlying platform, we assume a general-purpose shared-memory architecture with multiple multi-core processors/CPUs. All cores within each CPU share the same last level cache and operate at similar frequency levels. We start by presenting the analytical model for the execution time of a multi-tasked computation on top of such a platform, and the expected energy consumption. We then discuss the process that we follow to train the model through an offline profiling and fitting phase.

5.1 Analytic Model of Execution Time

Let a computation employ m task-groups, with each group i consisting of n_i tasks. Let the $ratio$ for group i, namely the minimum percentage of its tasks that need to be executed accurately, be r_i. Also, let the average execution time of accurate and approximate task versions for group i be equal to $\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$, respectively. For simplicity, we assume that a task group is well-balanced, and that all tasks take roughly the same time to execute, subject only to whether they are executed accurately or approximately. Then, the time that is required for the computation to be executed sequentially, is given by Equation 5.1, as a function of the input size s, the CPU frequency f, the ratios $\vec{r}$, and number of tasks $\vec{n}$ for each group.

$$T_{seq}(f, \vec{r}, s, \vec{n}) = \sum_{i=1}^{m} \left(n_i \cdot (r_i \cdot \overline{T_{accurate_i}(f, s, n_i)} + (1 - r_i) \cdot \overline{T_{approx_i}(f, s, n_i)}) \right) \quad (5.1)$$

Note that a larger problem size s may require a higher number of tasks in certain groups, or more work per task, or both. Indeed, the number of tasks n_i and the time it takes for a task of group i to execute in its accurate or approximate version ($\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$) are open parameters of the model. This makes it possible to implicitly account for effects that can significantly affect task execution time, such as locality, caching and memory traffic due to different input, intermediate and output data footprints associated with different problem sizes.

Equation 5.2 estimates the parallel execution time for the same computation, as a function of the number of cores c that are used. The assumption is that all cores run at the same frequency f, which is typically the case in many off-the-shelf platforms, including the one we use in our evaluation.

$$T_{par}(f, \vec{r}, s, \vec{n}, c) = \frac{T_{seq}(f, \vec{r}, s, \vec{n})}{c \cdot scaling(f, s, c)} \quad (5.2)$$

The term $scaling(f, s, c)$ captures the scalability of the computation as a function of input size s, the frequency f at which (all) cores run, and the number of cores c. On a multiprocessor with multi-core CPUs, we assume a "packed" CPU allocation strategy, whereby the runtime exploits all cores in a given CPU before using the

cores in another CPU. Thus, at most one CPU can have unused cores, which is the most energy efficient allocation strategy for common platforms.

5.2 Analytic Model of Power and Energy Consumption

The power consumption of the processing elements is given by Equation 5.3.

$$\begin{aligned} P(f, c, s, \vec{r}) &= P_{idle}(f, c) + P_{dynamic}(f, c, s, \vec{r}) \\ P_{idle}(f, c) &= P_{leak}(f, c) + P_{shortCircuit}(f, c) \end{aligned} \tag{5.3}$$

P_{idle} captures the power consumed by the number of active cores c running at frequency f, when idle. The $P_{dynamic}$ component corresponds to the "dynamic" power consumption, which depends on the computation that is actually being executed. This, in turn, is a function of the number of cores used, the frequency of these cores, the input size and the mix of accurate/approximate tasks. The rationale behind this is that the same task-group might behave differently for different values of *ratio*. The actual accurate/approximate mix affects the instruction mix of the overall application as well as the memory locality and access pattern.

Since $Power$ and $Time$ have been modeled we can now calculate the modeled $Energy$ footprint:

$$Energy(f, \vec{r}, s, \vec{n}, c) = T_{par}(f, \vec{r}, s, \vec{n}, c) * P(f, c, s, \vec{r}) \tag{5.4}$$

5.3 Offline Profiling and Model Fitting

In a profiling phase, the computation is executed with three different, representative input data-sets, of varying size s (and thus also different memory footprints). To account for locality, caching and memory traffic effects, we execute with a small working set that fits in the last level cache (LLC) of a single processor, a large working set that exceeds the total LLC capacity of all processors in the system[1] and, finally, an intermediate working set. For each input, we execute the computation for all possible configurations (varying the number of cores c, the frequency f and the task ratio $\vec{r}$). We measure the average execution time of approximate and accurate tasks for each task group, and the total execution time of each group.

We then perform a step-wise model fitting process, where the performance data that was gathered in the profiling phase is used as input to a regression process. The objective is to train the analytic models introduced earlier, so that they predict execution time and energy consumption of a given computation for different, unseen configurations.

The first step is to produce estimation functions for $\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$ in Equation 5.1. We perform regression to map the average execution time of tasks in a

[1] We skip problem sizes which are unrealistic. This is done for the large data-set in the Monte Carlo and MD benchmarks.

given group i, for both their approximate and accurate versions, to the frequency f, problem size s, and number of tasks n_i. A separate function is created for each of the frequencies that are supported by the platform. We use the average execution time of tasks that is observed when executing across all ratios. Exponential, polynomial and linear fitting functions are all tested, and we use the one which minimizes the prediction error with respect to profiling data used for training. Note that, the data that we use to evaluate the efficiency of the analytic models are unseen during the training phase.

Next, we produce the function for the *scaling* term in Equation 5.2, using the measured sequential and parallel execution times for different combinations of problem sizes and number of cores (the latter for parallel execution times). We also experiment with exponential, polynomial and linear fitting functions. The result is a separate function for each frequency, which correlates scalability to problem size and the number of cores used.

In a last step, a similar approach is followed to produce the function for the dynamic power consumption $P_{dynamic}$ component used in Equation 5.3. Again, a separate function is produced for each frequency, which returns an estimation based on problem size, $ratio$ and the number of cores used. Note that P_{idle} can be computed just once, measuring the power consumption as a function of the number of cores that are turned on, without running any computation. Even though P_{idle} is a function of the number of cores, it essentially quantifies the cost of operating a socket (and thus cores within that socket). In other words, utilizing a single core within the socket, in terms of P_{idle} is just as expensive as utilizing all of the cores within that socket.

The whole profiling and model-fitting process is repeated for each application, yielding different functions for each case. We make this application-specific information available to the runtime system, in order to enable it to proactively select the best configuration for a given energy budget.

5.4 Benchmarks

We use nine benchmarks to validate our framework and its ability to execute applications with a pre-defined energy budget, while gracefully trading off output quality with energy efficiency. The benchmarks have been manually ported to the proposed significance-aware programming model. We compare our framework against loop perforation [90] in terms of quality of results under the same energy constraints. We apply different approximation approaches to each benchmark, subject to the algorithmic characteristics of the underlying computation.

For the Sobel benchmark, significance is assigned to tasks in a round-robin manner, which ensures that approximated pixels are uniformly distributed throughout the output.

In Discrete Cosine Transform (DCT) we assign higher significance to tasks that compute lower frequency coefficients, as the human eye is more sensitive to those frequencies. Should a task be executed approximately, the computation is dropped.

In accordance with the analysis presented in Section 4.2, Fisheye tasks which are closer to the image border receive a higher significance value whereas those closer to the center receive a lower value. The approximate task calculates output pixels using the value of the nearest neighboring pixel instead of interpolating around the corresponding point in the input.

Approximate tasks in K-Means compute a simpler version of the Euclidean distance while also considering only half of the total dimensions. The second phase is significant, as it is harder to recover from a wrong estimate of a cluster center.

Approximate configurations for the MC benchmark (Monte Carlo) drop a percentage of the random walks and the corresponding computations. An approximate, lightweight methodology is also used to decide how far from the current location the next step of a random walk should move.

In Canneal, approximate tasks try less swaps (1/8) than accurate ones. All tasks are assigned the same significance value, so the tasks to be approximated are randomly selected by the runtime, according to the target *r*atio.

A single N-Body task evaluates the Coulomb forces which affect the particles within a home region due to the existence of particles within a second foreign region. The magnitude of the forces between particles decreases the larger the distance between the two particles. It follows that the larger the distance between the home and the foreign regions, the lower the significance of the task.

In Blackscholes the computation of a stock price can be broken down to 4 blocks of code A, B, C, D, with $sig(A) > sig(B) \gg sig(C) > sig(D)$. Each block produces the input to its successor block. Block C performs two Cumulative Normal Distribution Functions (CNDFs) and block D produces the final output by combining the outputs of C. The two least important parts (C and D) are approximated using less accurate but faster implementations of mathematical functions such as exp and $sqrt$ [61].

In Lulesh, similarly to N-body we consider the significance of particles to be diminishing when moving away from the impact site. Computations involving the least significant particles can be dropped at execution time.

5.5 Experimental Methodology

The experimental analysis was carried out on a system equipped with two Intel(R) Xeon(R) E5-2650 processor, and 64 GB shared DRAM. Each processor has 8 cores and can be clocked at 1.2, 1.6, 2.0, 2.4, or 2.8GHz. Energy and power are measured using the Running Average Power Limit (RAPL) registers of the processors.

The profiling phase uses a pool of representative input sets for each benchmark, discussed in Section 5.3. At the end of the profiling and model fitting process, each

benchmark is associated with a model estimating its energy consumption according to the input size and execution configuration. This formula is, in turn, used by the runtime system to take online decisions on the execution configuration.

To evaluate our approach, we use for all benchmarks unseen input sets (and input set sizes) which have not been used during the training phase. All benchmarks are executed accurately, in all possible core and frequency configurations. From those executions we identify the one that consumes the least energy. This is our baseline scenario for each benchmark.

We then perform a number of experiments for each benchmark, while requesting a gradually smaller energy budget, expressed as a percentage of the baseline. The framework uses the model to decide, at execution time, the ratio, and concurrency level with which it can achieve execution within the requested energy budget, while minimizing the impact on output quality by maximizing the ratio of accurate tasks.

We present a comparison of the quality achieved using our framework with a perforated execution of each benchmark targeting the same energy budget. We also present the optimal (oracular) configuration (cores, ratio) for each case and compare it to the one selected by our system.

5.6 Experimental Evaluation and Discussion

Figure 5.1 summarizes our results. In all charts the horizontal axis represents the requested energy budget, as a percentage of the energy consumed by the most energy-efficient accurate execution. The Y-axis of the first set of charts corresponds to the energy that was actually consumed by approximate executions as a percentage of the energy consumed by the accurate execution. Note that the green dotted line also shows the user requested energy budget. The second set of charts is used to quantify output quality.

We use two references to assess the effectiveness of our methodology: a) loop perforated versions of the benchmarks guided by an oracle and b) an oracle (optimal) configurator for the approximate versions of the benchmarks.

The Oracles represent the best case scenarios for each user specified energy budget. They cannot be actually implemented and are simply utilized to illustrate the optimal choice. In practice, Oracle decisions are taken offline, after executing and profiling all possible configurations. In more detail, the algorithms iterate through the configuration space in the following dimensions: a) number of cores, b) loop-perforation/approximation ratio value, c) energy consumption, and d) output quality. Their goal is to identify and report, the highest output quality configuration among the configurations that consume energy within the user specified energy cap. The only difference between the Approximate Oracle and the Loop-Perforation Oracle is that the first one exploits significance-aware approximate computing to optimize the execution of the benchmarks, whereas the second Oracle employs loop perforation.

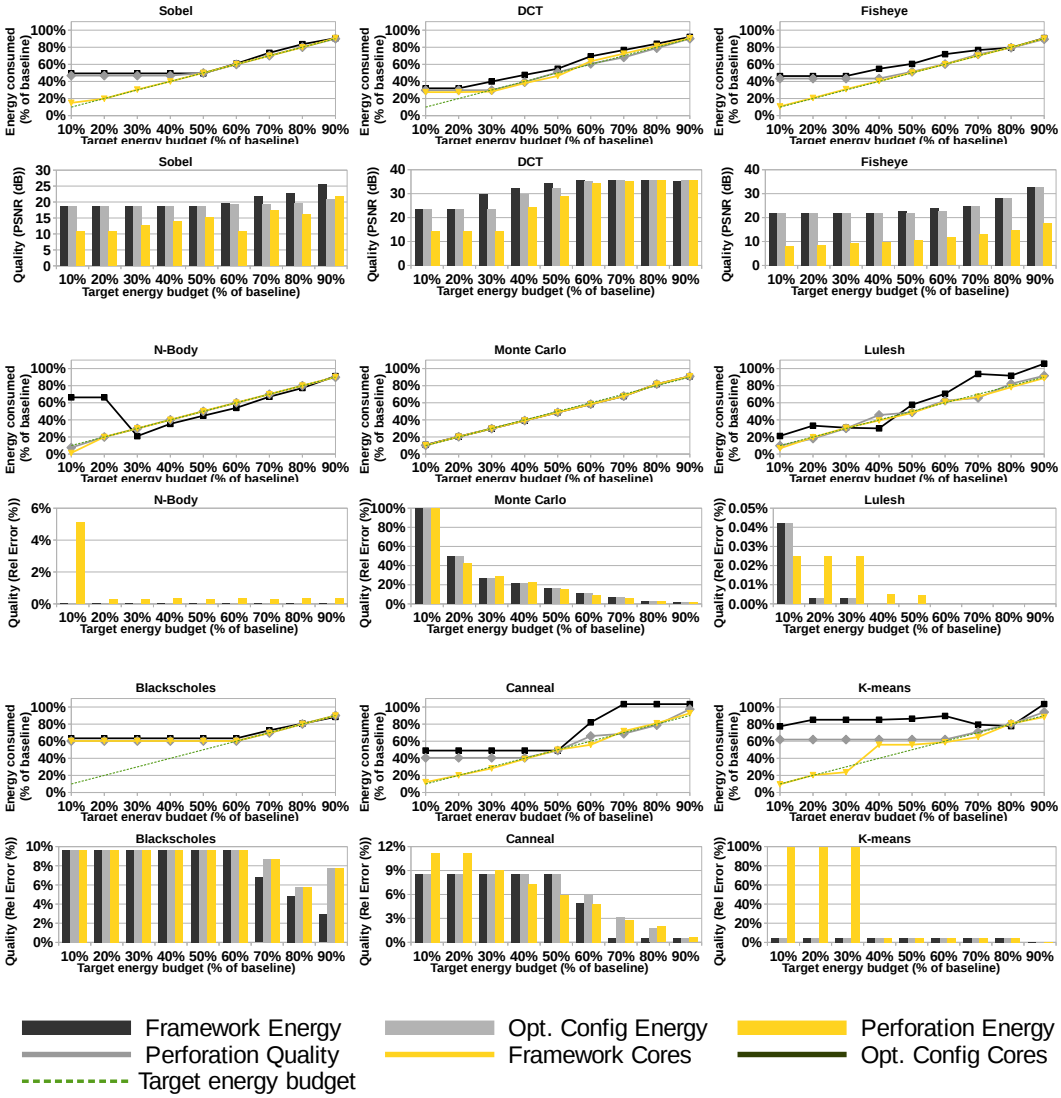


FIGURE 5.1: Quality and energy metrics for different energy targets (as a percentage of the most energy-efficient accurate execution). Energy & quality plots show the results achieved by our system, an oracle selecting the optimal configuration and loop perforation.

For the first three applications *(DCT, Sobel, Fisheye)* output quality is quantified using *PSNR* (higher is better). PSNR is a logarithmic metric. For *Kmeans*, the metric of the quality of output is the relative difference of the average distance between data points and the center of the cluster they are assigned to, compared with that of the fully accurate execution (lower is better). For the remaining five benchmarks we report the relative error with respect to an accurate execution (lower is better).

Our framework produces configurations with energy consumption very close to the optimal one. Even in cases when the runtime opts for a non-optimal configuration, the difference in the achieved energy footprint and quality of results is negligible, with the exception of Canneal, Kmeans, and Lulesh which are discussed in more detail later in this Section. Both our approach and the optimal tend to adapt

concurrency to utilize all cores of both CPUs. Activating the first core within a socket results in a measurable increase of power consumption. However, subsequent activations of cores residing within the same socket come at a reduced cost. The runtime, in most cases, succeeds in determining the appropriate number of cores to both fulfill the parallelism requirements of an application with a reasonable cost in power consumption. Consequently, it is able to efficiently expend the user-specified energy budget.

Imaging and media applications are well-suited for our programming framework, as they take full advantage of the significance and approximation features of the programming model. Moreover, the specific implementations scale to larger inputs by adapting the number of tasks, instead of modifying the work per task. Therefore it is easier for our model to predict their behavior with high accuracy. Finally, the execution of approximate tasks has a straightforward and easy to model effect on execution time: more approximate tasks result in less computation and thus more energy savings.

Sobel DCT, and *Fisheye* can execute with as little as 50% of the energy required by the optimum accurate execution and match the quality achieved by the oracle. The minimum energy required depends mainly on the complexity of the approximation function we use. At the same time, the complexity and sophistication of the approximation function determines output quality for the most aggressive degrees of approximation. When approximating all tasks we observe PSNRs equal to 18.70, 23.64, 22.09 dB for *Sobel*, *DCT*, and *Fisheye* respectively. Perforated executions capped at the same amount of energy produce results of inferior quality, corresponding to PSNRs of 10.75, 14.48 and 8.19 dB respectively. Our methodology clearly results in higher quality of results with the same energy budget. However it sometimes slightly overshoots the energy budget constraints by picking ratio values which are higher than the optimal. In the case of DCT we overspend, on average, by 6.2%, for Sobel by 2.1% and finally Fisheye overspends by 6.4%. This leads to a pitfall in Figure 5.1 where our framework seems to outperform the oracle, which is clearly not possible. Figure 5.2 depicts the Lena portrait compressed and decompressed using DCT with a ratio of 0.3. The resulting output has a PSNR of 34.62 dB (no visible quality loss), at a 45% energy gain with respect to the most energy efficient accurate execution.

N-body is another well-behaved application for our framework. In most cases we choose configurations which result in energy consumption that is very close to what an oracle achieves. In fact, our estimations, excluding the energy budgets 10% and 20% result in energy consumption which differs by 4.5% from the user specified energy budget. Moreover, we always achieve a better quality of results than the perforated version of the benchmark. With just 30% of the energy budget of the most energy efficient accurate execution N-body computes results with a relative error in the order of 0.0006%.

For Monte Carlo we observe that our framework makes optimal choices in almost every case. Approximation in Monte Carlo drops random walks, similarly to

FIGURE 5.2: Lena portrait compressed and decompressed using DCT with a ratio of 0.3.

perforation, therefore we observe similar results with both techniques. A lower energy budget results in pruning some of the random walks of the search space. This reduces energy, albeit with a measurable impact in quality. We can achieve consumption as low as 30% of the energy required by the most energy-efficient accurate execution, using a ratio of 0.2 which results in a relative error of 5.9%.

Regarding Lulesh, we notice that for energy budgets higher than 10% the version executed by our framework always produces results of higher quality than the perforated one, but it tends to overshoot the energy budget. The case of the energy budget being 10% of the optimal accurate is particularly interesting: the perforated version is better in terms of quality and energy than both our framework and the oracle. This is due to the fact that the approximate executions have to spend some of their energy budget to compute the significance of tasks. Furthermore, approximate tasks do not access the memory with a regular pattern. Elements are visited according to their distance from the point of blast. Unfortunately, this access pattern negatively affects memory locality. On the other hand, the perforated version has a regular memory access pattern and the respective energy drops linearly with respect to the number of dropped iterations. However, for higher – and realistic – energy budgets our approach always produces results of better quality compared with the perforated executions. We do have to note that the two issues described above limit the accuracy of our framework's estimations. Figure 5.3 depicts the positions of particles calculated by a small-scale approximate execution with ratio set to 0.2. Particles are colored according to the relative error of their final position with

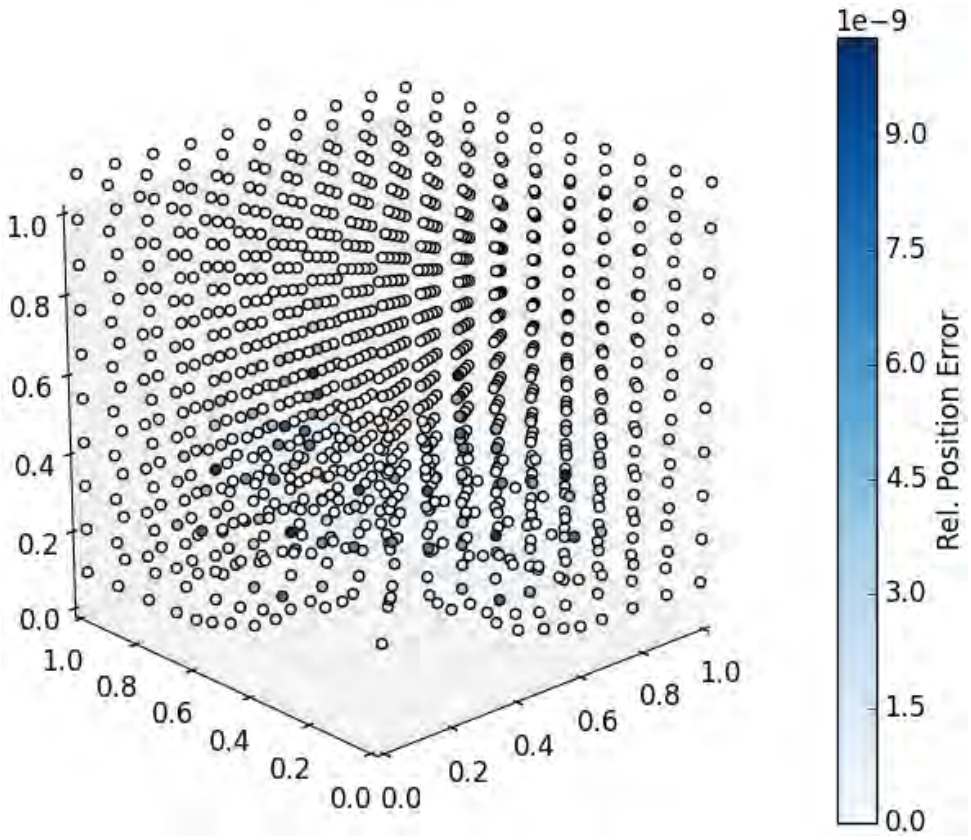


FIGURE 5.3: Final positions of particles for an approximate execution with ratio 0.2. Particles have been colored according to the relative error of their position with respect to an accurate execution.

respect to the fully accurate execution. The maximum relative error is negligible (in the order of 10^{-8}).

Blackscholes calculates prices for a number of assets. The main loop iterates across different assets, however there is no loop involved in the calculation of each particular asset [90]. As a result, perforation is not applicable and we limit our comparison between the proposed framework and the optimal configuration by an oracle. Because of the computational cost of approximate tasks, the lowest energy budget obtained by the Oracle is 60% of the accurate execution; our framework follows closely at 63.3%. Once again, we produce results of higher quality than the oracle for energy budgets higher than 60% due to slightly overshooting the target energy budget by executing more accurate tasks.

Our model is less accurate in its predictions for Canneal. This is a consequence of the bad, unpredictable locality pattern if the application. Canneal uses large data structures to store information on net-list elements. The random way each task accesses memory locations increases cache misses, in particular false sharing misses that introduce excessive data transfers between the last-level non-shared caches of cores. This unpredictable behavior cannot be modeled accurately by our framework. As a result, we underestimate the execution time of the application and often select configurations that do not satisfy the energy constraints.

K-means reveals the limitations of our approach. It can not be modeled effectively, as it is iterative, with the number of iterations being heavily dependent on the characteristics of the input set (and not just the input set size). Moreover, wrong decisions in the approximate tasks (point classification) tend to increase point movement between clusters, and thus the workload of accurate tasks (cluster center calculation). In addition, even when we approximate 100% of the point classification tasks, we can only reduce the energy footprint by at most 60% because our approximation disregards half of the coordinates of each point. For such applications, a blind approach such as loop perforation proves to be a viable solution for medium-to-large energy budgets as it produces solutions which are as good as our framework using less energy.

To summarize the results of our experimental campaign we note that there are scenarios in which it is simply impossible to arbitrarily decrease the energy footprint of an application, due to the fact that even the approximate versions of tasks come with computational cost. We do observe however, that in the bulk of the test cases our framework succeeds in gracefully trading quality to reduce the cost of executing an application.

5.7 Conclusions

This chapter concludes the introduction of the necessary tools for approximate computing. We first introduced a significance aware programming model for approximate computing and its accompanying runtime system in Chapters 3. Then, in Chapter 4 we presented a methodology for discovering the significance of computations as well as implementing approximate alternatives to code which are less costly in terms of computational resources. Finally, in this Chapter we demonstrated the use of application modeling to automate the process of deciding on a value for the *ratio* clause of the taskwait directive which controls the level of approximation. An application developer equipped with this arsenal of techniques and frameworks discussed earlier, is significantly assisted in using the significance aware approximate computing programming paradigm.

Chapter 6

Automatic result checking for fault-tolerant computing

A major challenge is detecting errors before they irreversibly modify application state. Ideally, a detection mechanism should be able to distinguish between correct and incorrect computation results, at a very low overhead. Lower levels of the system such as circuit, micro-architecture and architecture largely mask hardware faults before they incur changes to the application layer [105]. Unfortunately, errors do manage to propagate to the higher levels of the system and they need to be handled before they modify the outputs of applications. To this end, researchers in the past have devised methodologies to automatically detect timing-errors at the level of hardware through the use of extra circuitry [9, 15]. Their work focuses on detecting and correcting timing errors before their effects propagate further into the arhitecture state.

In our case, we follow a software-only approach to error detection. Note that, we are not concerned about all possible errors. In fact, we are explicitly relaxing the expectations of developers regarding the output quality of their applications so that we can effectively trade-off output quality with improved performance. As such, out of the errors which may modify the application state, our work involves detecting the errors which significantly affect the output quality of the application, and correct them before they propagate to the output. Depending on application characteristics, minor deviations from correct intermediate results may not be worth paying the correction cost.

In this Chapter we introduce our solution to the question of "**How** to detect errors?". More specifically, we present a methodology for automatic error detection, based on Artificial Neural Networks (ANNs). ANNs have been successfully deployed in pattern matching and classification problems, sometimes even outperforming human accuracy [32, 2]. Given the configurability of their architecture, one can flexibly trade-off performance and classification accuracy by means of modifying the number of layers and the number of nodes per layer. Moreover, ANNs are highly parallel and can be designed to operate using only simple operations. In particular, the ANNs used in our work employ at most five different operations: floating point addition, subtraction and multiplication, as well as binary shift and

logical AND. This simplifies the automatic compiler-driven generation of vectorized implementations of ANNs for improved performance. Of course, additional performance improvements can be unlocked through the use of FPGAs [3], GPUs, or even specialized processing units such as Google's Tensor Processing Units (TPUs) [43].

This chapter makes the following contributions:

(i) We propose using ANNs to detect hardware errors during program execution. To the best of our knowledge, this is the first work that uses ANNs for hardware error detection.

(ii) We evaluate the overhead of our approach and its impact on the quality/accuracy of the end result, on a set of eight applications from the domains of imaging, finance and physics, via software fault injection experiments.

(iii) We compare our approach with Topaz, a state of the art approximate error detection mechanism [1].

(iv) We introduce a metric for capturing the efficiency of different error detection mechanisms (in this particular case, ANNs vs. Topaz).

(v) We present an indicative case study on how the overhead and error-detection accuracy of ANNs vs. Topaz can affect performance and result quality, for a system configuration where computations can be executed on overclocked cores at the risk of unreliable hardware operation.

Our results show that ANNs can be quite effective for error detection purposes, offering a good trade-off between accuracy and execution overhead. At the same time, they can be generated in a highly automated manner with little effort from the application developer.

6.1 Artificial Neural Networks for Error Detection

The manual implementation of accurate and low cost error detection is a time consuming and intricate process. The developer should be highly familiar with the application, in order to be able to take educated decisions on how to detect errors in the output of each task. Also, it can be quite hard to find the desired balance between execution complexity/cost and error detection accuracy.

Previous work has mainly explored the use of manually implemented approximate error-detectors which are low-cost but may potentially produce wrong estimations of error presence [31, 44, 27]. In contrast, we propose to use ANNs as automatic and light-weight error detectors. The main advantage of ANNs is that they require little manual intervention, and offer the opportunity to trade-off error detection accuracy with performance overhead in a flexible way. In the following, we describe our methodology for the generation of ANN-based error detectors for task-based computations as illustrated in *Figure 6.1*.

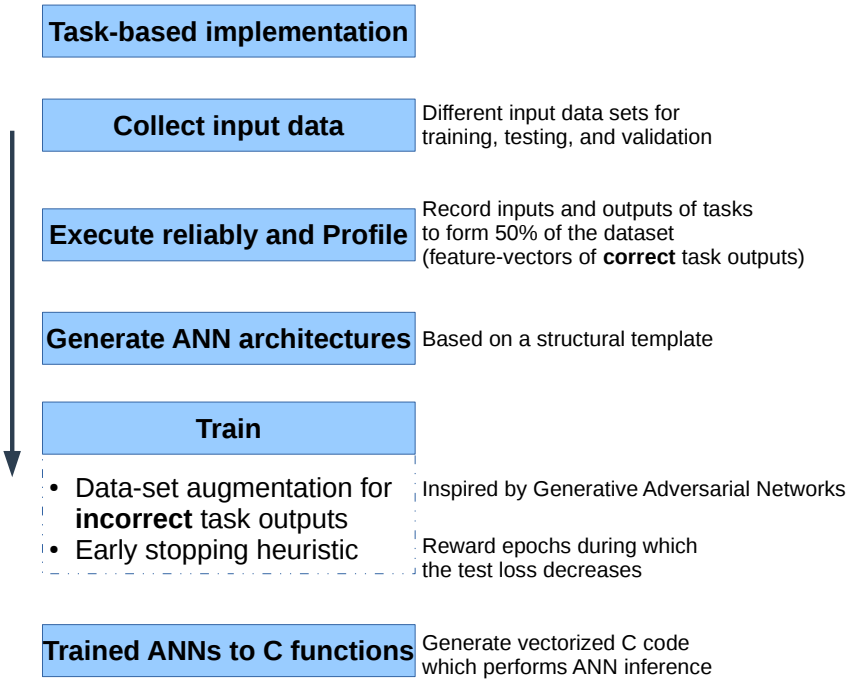


FIGURE 6.1: Training Artificial Neural Network classifiers for online error detection

6.1.1 Application profiling

We begin by partitioning the application code into fine-grained tasks. The outputs of those tasks are subjected to error detection. The motivation for fine-grained tasks is that if the task size is rather small, then its output is more likely to exhibit a detectable pattern. This, in turn, makes it feasible to train an ANN so that it can identify this pattern (or deviations from it).

The data that is fed into the ANN to determine whether the output of a task is correct or incorrect are referred to as *feature vector*. The feature vector always includes the task output. This can be sufficient for tasks with distinctive output patterns, irrespectively of their input. If the output pattern is highly input-sensitive, the feature vector may also have to include parts of the task input, or in the extreme case the entire input. In the latter case the ANN essentially becomes an approximation function for the code of the task. In our work, the feature vector is manually defined by the application developer, although in principle this step could also be largely automated.

The application is then executed in a reliable way for different inputs. In every execution, we record the feature vector of each task. These data are aggregated for each task across all executions to produce a so-called *profile* data set. We select $80-90\%$ of the profile data set to construct the *training* set, while the remaining $10-20\%$ of the profile set is used as the *test* set.

Both data sets are necessary for the training phase of the ANNs to minimize the probability of over-fitting to the particular data set that was used for training. Overfit ANNs do not generalize well to unseen data because they have overly adapted

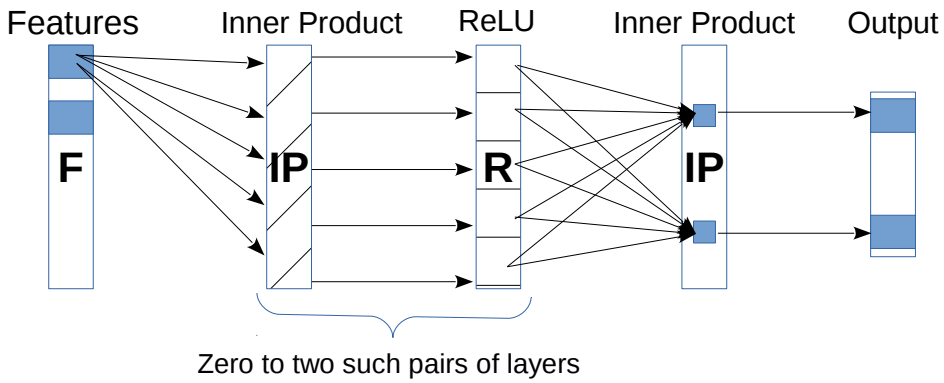


FIGURE 6.2: Structural template for the generated ANNs. The input is the feature vector of the task. The output consists of two values corresponding to the one-hot classification of the task output as correct or incorrect. There can be zero up to two intermediate pairs of IP and ReLU layers.

to the particular data sets that were used during their training. Training an ANN is an iterative process; firstly, the training set is fed forward to the ANN so that it generates some output. Afterwards, the output is compared against the labels of the training data via the loss function[1] and subsequently the weights of the ANN are updated appropriately, via back-propagation of the loss, so that the output of the loss function is optimized (minimized). The test set is only used to feed forward the ANN and its loss is an indication of how the ANN performs on unseen data. The above is repeated throughout the training phase as described in Section 6.1.3.

6.1.2 ANN structures

For online error detection, we consider the least complicated type of ANNs: Multi-layer Perceptrons that consist of InnerProduct (IP) layers and Rectified Linear Units (ReLUs) to serve as activation functions between IP layers.

IP layers accept M inputs and produce N outputs. Each such layer contains an MxN matrix of weights (W) and a vector ($\vec{b}$) that stores N bias values. Assuming vector $\vec{x}$ is input to an IP layer, the resulting output vector is $\vec{y} = W * \vec{x} + \vec{b}$. ReLU activation layers are placed between two IP layers to introduce non-linearities which improve the generalization of the ANN as well as its classification performance. A ReLU with an input vector $\vec{x}$ and output vector $\vec{y}$ simply computes $y_i = max(0, x_i)\ \forall\ x_i \in \vec{x}$.

In terms of operations, IP layers comprise floating point additions, subtractions and multiplications. ReLUs can be implemented via the gcc built-in function $fmaxf()$, or using binary arithmetic. This relatively simple ANN template enables us to easily produce vectorized C functions which can then be compiled into appropriate vectorized instructions for maximum performance.

[1] A loss function is some function of the difference between the output of an ANN and the test/train data labels. The smaller the evaluation of the loss function, the closer the ANN's predicted output to the true output (data-labels) is

Notation	Structure
N,2	$f[N] \rightarrow IP[2] \rightarrow out[2]$
N,B/2,2	$f[N] \rightarrow IP[B/2] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$
N,B,2	$f[N] \rightarrow IP[B] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$
N,B*2,2	$f[N] \rightarrow IP[B*2] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$
N,B/2,B/2,2	$f[N] \rightarrow IP[B/2] \rightarrow ReLU \rightarrow IP[B/2] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$
N,B,B/2,2	$f[N] \rightarrow IP[B] \rightarrow ReLU \rightarrow IP[B/2] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$
N,B*2,B/2,2	$f[N] \rightarrow IP[B*2] \rightarrow ReLU \rightarrow IP[B/2] \rightarrow ReLU \rightarrow IP[2] \rightarrow out[2]$

TABLE 6.1: The seven different ANNs used for error detection. f is the feature vector of the task, N is the size of the feature vector, B is the power of two closest to N, and out is the 2-dimensional output. The dimension of the input/output vectors and IP layers are given in brackets

For every task for which we wish to build an error detector, we experiment with different ANN structures, according to the template shown in *Figure 6.2*. The first IP layer always takes as input the feature vector for the task in focus (the size of the vector depends on the task). The last IP layer produces exactly two values, which encode information about whether the output of the task is considered to be correct or incorrect. The ANNs we employ here work as so-called *one-hot classifiers* where each component of the output vector gives a score for the correctness and the incorrectness of the input feature vector, respectively. The component with the highest score is considered to be the output of the one-hot classifier. In the unlikely event of a tie, we assume that the output is incorrect. With each ANN we explore a different combination for: (a) the number of IP layers and ReLUs, and (b) for the sizes of these layers, where we try out different powers of two as this facilitates vectorization. These ANNs vary in their internal computational complexity and thus may have different error-detection accuracy. More specifically, for each task we try out seven ANNs listed in *Table 6.1*. As a shorthand notation, the structure of each ANN can be encoded using a tuple of integers that represent the size of the feature vector, the size of the (zero, one or two) intermediate IP layers (and ReLUs) and the size of the output IP layer which is always two. These ANN structures are produced automatically, based on the size of the feature vector.

6.1.3 Training the ANNs

We train the resulting ANNs using Caffe [41]. Training is done in so-called epochs: an epoch is over when the training set has been fed-forward, the loss function for the resulting ANN outputs has been evaluated, and the weights have been adjusted by back-propagation of the loss on the test data. The completion of the training process is decided using the following heuristic. Initially, when training starts, we issue 100 *tickets*. At the end of each epoch we check the loss of the ANN on the test data. If the test loss has decreased compared to the previous epoch, then the number of tickets is increased by one, else it is decreased by two. When there are no tickets left, the

ANN is assumed to have reached an acceptably low loss and training terminates. Rewarding loss decrease enables recovery from local test-loss minima. Consecutive training epochs can increase the test loss before it is eventually reduced beyond its past minimal value.

For a task output that comprises N values, each one consisting of k-bits, there can exist $N * (2^k - 1)$ different incorrect output variants. If, during training, one considers all possible incorrect values for a single correct output, this would bias the ANN to always infer that a feature vector is incorrect, as that would minimize the loss function. To counter-balance all the possible incorrect output patterns, one could repeatedly feed the correct feature vector multiple times, but this would dramatically increase the size of the training and test data sets, leading to unacceptably long training times.

As a more practical approach, we use data augmentation [2] to periodically generate data sets that contain an equal number of correct and incorrect instances of the feature vector. Our approach is inspired by Generative Adversarial Networks (GANs) [24]. A GAN comprises two NNs, the Generative Network (GN) and the Discriminative Network (DN). Both NNs are trained simultaneously: the GN generates artificial data which appear to be realistic, whereas the DN determines whether data has been produced by the GN or is an actual real-world sample. The ANNs used for error detection play a similar role to a DN of a GAN. In our case, instead of training a GN we perturbate the correct feature vectors (from the profile data) to produce incorrect ones, every few training epochs. This way the ANNs are trained to classify feature vectors that have a strong similarity (but are not necessarily identical) to the correct ones as correct, and to classify widely different patterns as incorrect.

6.1.4 Deployment

We have implemented a Caffe-to-C python script which takes an ANN model from Caffe and produces C code that performs just the inference (feed-forward) operation. The C code consists mostly of *gcc* vector extension intrinsics, and we rely on *gcc* to automatically generate vectorized implementations for maximum performance.

The last step is to add the code of the ANN-based error-detectors in the application, so that they are invoked after tasks complete, in order to check their output and decide whether they need to be re-executed. However, recall that tasks are intentionally fine-grained, and it is unrealistic for the application to be executed at such an extremely fine level of granularity. Due to the system-level task management overheads, this would incur significant performance penalty. To limit effects to performance, we group a large number of independent tasks into large *gangs*, which are the actual, much coarser scheduling unit for the underlying runtime system. When a gang completes its execution, the output of each task is checked individually via

[2] Data augmentation is the practice of generating new data samples from old ones to increase the training/testing data set.

the corresponding ANN, and if it is classified as incorrect then the task is flagged for re-execution.

We also explore the use of ANN-based error detectors over the aggregated outputs of several tasks, referred to as *batches*. A task batch typically comprises just a few tasks (which may have dependencies), and for all practical purposes it can be treated as if it was a single, coarser task. The process of producing the respective ANNs for detecting erroneous outputs remains similar to the one described above. Application profiling and ANN training are exactly the same. The only difference is that the feature vector has to be defined to include the aggregated outputs of the task batch and possibly parts of the aggregated inputs.

6.2 Evaluation approach

We evaluate the efficiency of ANNs acting as error detectors using the framework described in [71]. The runtime system executes the main application thread under reliable conditions and schedules tasks on 4 worker threads which are mapped on the cores of an Intel i7 4820k CPU. Unreliable task execution is simulated by means of software fault injection. Beyond measuring the number of cycles spent to execute tasks unreliably, perform error detection, and error correction we also introduce a metric to evaluate the efficiency of error detectors in terms of precision and error detection overhead.

6.2.1 Fault injection approach

The programming model and runtime system require that at least one core always operates under reliable conditions to boot the machine, run the OS, and execute the heavyweight operations of the runtime system. We also assume that system configuration parameters related to hardware reliability (such as CPU operating voltage and frequency) may change while the machine is operating. Unfortunately, commodity hardware does not support fine-grained dynamic reconfiguration outside the nominal working envelope of the processor. Although sub-nominal voltage/frequency configurations are attainable in x86 via BIOS configuration, they are effective for the entire CPU instead of at a per-core granularity.

We resort to software fault injection to simulate the unreliable execution of code using the methodology described in [71]. A fault-injection experiment injects random, multiple bit-flip faults on CPU registers once every 10^7 cycles. We select the fault-rate based on the findings of Section 3.9.4. Evidently, operation at the specific configuration point leads to a nice trade-off between performance improvements and the number of extra CPU cycles spent to correct erroneous outputs. Recall that, a higher fault-rate translates to a higher CPU frequency with more frequent errors. One the one hand, a high CPU frequency means increased performance for code regions which execute unreliably. On the other hand, the higher the fault-rate the

fewer tasks execute till completion, and the larger the number of tasks which compute erroneous outputs. A side-effect of which is increased overhead CPU cycles that are spent to correct the erroneous task outputs.

For each fault-injection, we first generate a random mask which covers all bits within a randomly chosen register and subsequently use this mask to flip the corresponding bits. The total number of fault injection experiments is 256000 for a confidence level of 99% and average margin of error 1.025% across the eight benchmarks.

6.2.2 Metrics

We measure the number of CPU cycles required to execute, error-check, and correct tasks as well as the resulting output quality.

At the same time, we introduce a new metric, which can be used to evaluate the relative effectiveness and efficiency of a set of error detection mechanisms in order to select the most appropriate for each use-case. The goal of the metric is to quantify and combine the application developer-apprehendable effects of performing error detection and correction. The two most profound ways that an error detection mechanism affects the execution of an application are (i) its efficiency as a binary classifier for *Incorrect* and *Correct* computation outputs and (ii) the overall overhead that it introduces. These two properties are not independent. In fact, the total overhead a mechanism introduces depends on both its computational complexity, as well as its level of precision and recall. Each time a task output is considered to be *Incorrect*, the cost of performing corrective action (task re-execution in our evaluation) is added on top of the detection overhead itself.

From the perspective of binary classification, the goal of the error-detection mechanism is to discover the *Incorrect* ones out of all task feature vectors. To this end, we use the terms *true/false positive/negative* for the case where the ANN, when used as a binary classifier, rightly (*true*) / wrongly (*false*) identifies a feature vector as incorrect (*positive*) / correct (*negative*). The True Positive Rate (TPR) indicates the probability that the error detection mechanism properly characterizes a feature vector containing erroneous output values as positive (*Incorrect*). False Negative Rate (FNR) is the probability that the detection mechanism will falsely tag a positive (*Incorrect*) feature vector as negative (*Correct*). In the context of error detection, TPR is more intuitively referred to as *Detection Coverage*.

The *Expected Error* quantity combines a detector's FNR with its $\overline{\textit{Missed Relative Error}}$ ($\overline{MRE}$). The latter denotes the average relative error across all of its false negative outputs.

$$\begin{aligned} \textit{E}\text{xpected Error} &= FNR * \overline{\textit{Missed Relative Error}} \\ \textit{E}\text{xpected Error} &= (1 - \textit{Detection Coverage}) * \overline{\textit{Missed Relative Error}} \end{aligned} \tag{6.1}$$

Intuitively, the *Expected Error* quantifies the average relative error of false negative, thereby *Incorrect*, outputs. In other words, it expresses the extent of errors which are missed by an error detector. To this end, the smaller the *Expected Error*, the better.

The Expected Error/Overhead Product (*EEOP*) combines the accuracy and the overhead of a detection mechanism. It can be used to choose the fittest out of a set of different result-checking mechanisms. The lower the *EEOP* of a mechanism the better/more efficient it is.

$$\text{Expected Error Overhead Product} = \textit{Expected Error} * Overhead \tag{6.2}$$

We define overhead as the percentage of cycles spent to detect and correct errors with respect to the cycles required to execute a benchmark under reliable conditions. Note that both error detection and error correction are performed reliably. Unfortunately, the *EEOP* metric treats the *Expected Error* quantity and the overall overhead of a detector in the same way. As such, it does not consider error-detectors with extreme overheads as exceptionally bad choices. Such overheads might occur due to either the complexity of the error-checking mechanism or an abnormally high False Positive Rate (FPR). Recall that, when an error detector produces a false positive decision for a task it results in unnecessary re-execution of the respective task, which increases the total overhead. We modify the metric so that it discards obviously inefficient error-detectors using a user-supplied value (ϵ) that specifies the highest tolerated error detection and correction Overhead:

$$\text{Expected Error Overhead Product} = \begin{cases} \textit{Expected Error} * Overhead, & \text{for } Overhead \leq \epsilon \\ \infty, & \text{for } Overhead > \epsilon \end{cases} \tag{6.3}$$

In the rest of this chapter, we set ϵ equal to 33% in order to discard extremely inefficient error detection mechanisms.

EEOP is a composite metric, which does not directly translate to a physical property. Therefore, reasoning on individual metric scores is not valid. However, the relative comparison of *EEOP* values for different error detectors enables developers to evaluate them using a single scalar value that combines both overhead and error-detection related properties. This is particularly useful in our case, because we automatically generate a number of ANN error detectors which typically represent different trade-off points between output quality and execution overhead. *EEOP* enables us to automatically select the fittest error detector out of multiple generated, without requiring human intervention beyond specifying the maximum overhead threshold (ϵ).

6.3 Evaluation

In this section, we discuss the experimental evaluation of the ANN error detection methodology we introduce. Furthermore, we compare our methodology against

Topaz [1], both in terms of accuracy and performance. Topaz executes computations using a heterogeneous computing platform that comprises a reliable main worker thread and multiple unreliable worker threads. It includes an approximate outlier detector which checks for the existence of errors on Abstract Output Vectors (AOVs). AOVs are constructed using either just the outputs of tasks or a developer implemented function. The latter operates on both inputs and outputs of a task to generate an AOV[3] which encodes the output vector of a task in a space of lesser dimension. Unlike ANNs, Topaz does not require an offline phase. When Topaz detects an error at the output of a task (false/true positive) it re-executes the task reliably, updates its error detection model, and then integrates the correct result in the main computation.

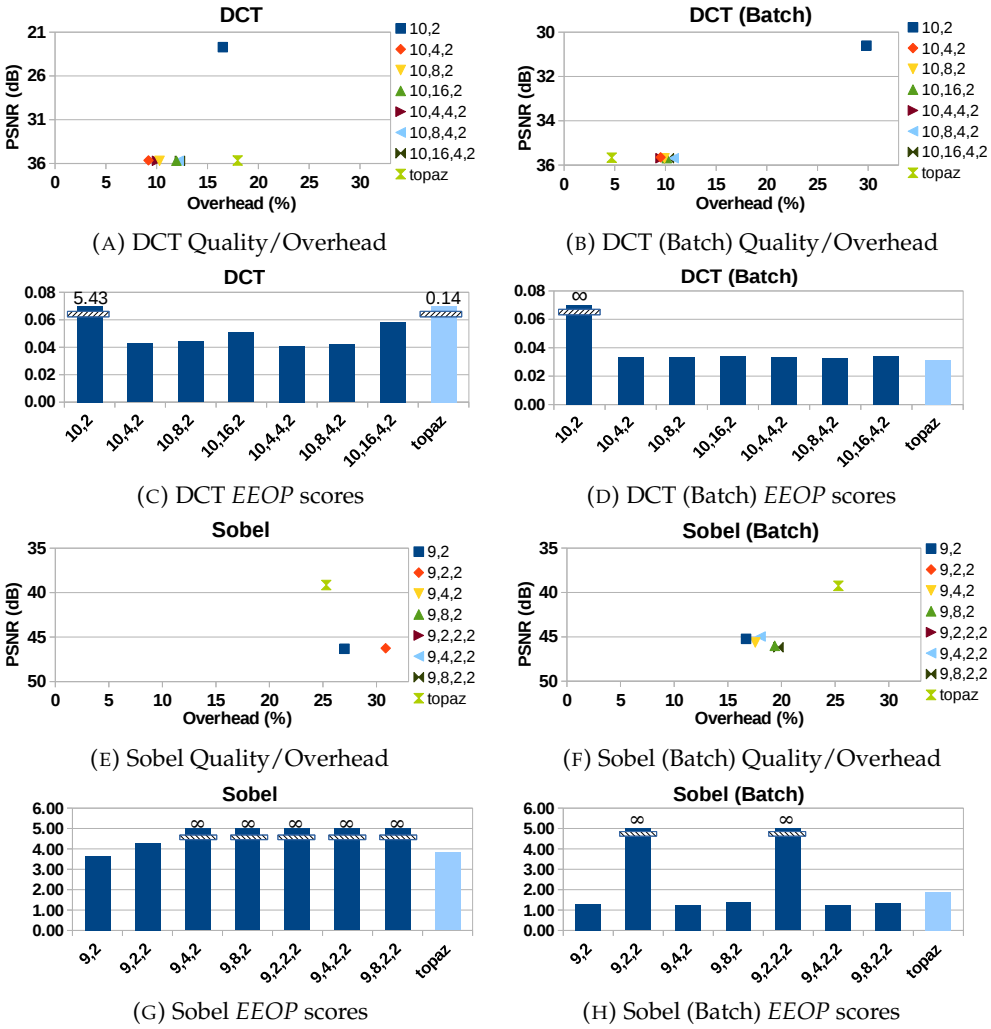
Both Topaz and ANNs can use more sophisticated AOVs/feature vectors which are generated via programmer provided functions that operate on the inputs and outputs of tasks. This comes, however, at an increased cost in terms of human effort. Because the focus of this work is error detection without the involvement of the programmer, we will not explore more intricate feature vectors beyond the inputs and outputs of tasks. Furthermore, the authors of Topaz argue in favor of reducing the number of AOV dimensions before performing the outlier detection test. In the spirit of keeping the automation level as high as possible we resort to the simplest way of dimensionality reduction, which involves batching the AOVs/Feature-Vectors of N tasks into a single bundle, before checking for errors on an aggregate value produced by the outputs of tasks. If the feature vector/AOV of the batch is found to be positive (is suspected to be erratic) by an error detector, all N tasks within the batch are re-executed.

6.3.1 Benchmarks

We use eight benchmarks from the domains of imaging, finance, and physics. For all benchmarks but two (DCT and Sobel), the quality metric is the Relative Error between the output of the unreliable execution and the baseline error-free execution. In DCT we measure the overall quality of the benchmark execution as the Peak Signal to Noise Ratio (PSNR) between the input image and the image that is the outcome of a sequence of DCT, Quantization, De-quantization and Inverse-DCT operations. For the Sobel benchmark, we measure the PSNR of the output of the unreliable execution with respect to a baseline error-free execution. For all benchmarks, the baseline error-free execution involves the scheduling of tasks on hardware which is configured to operate under reliable conditions.

The remainder of this section presents and discusses, for each benchmark, output quality and performance for the ANN and the Topaz error detection methodologies. Performance overhead is calculated as the percentage of CPU cycles required for

[3]When it comes to using ANNs, the equivalent of an AOV is a feature vector



(A) DCT Quality/Overhead

(B) DCT (Batch) Quality/Overhead

(C) DCT *EEOP* scores

(D) DCT (Batch) *EEOP* scores

(E) Sobel Quality/Overhead

(F) Sobel (Batch) Quality/Overhead

(G) Sobel *EEOP* scores

(H) Sobel (Batch) *EEOP* scores

FIGURE 6.3: Evaluation with DCT and Sobel. *Figures (a) and (b)* show the overhead and quality measurements for the non-batched/batched version of DCT respectively (similarly, *Figures (e) and (f)* for Sobel). The Overhead denotes the fraction of cycles required to perform error detection and correction divided by the cycles required for a fully reliable execution. The Y-axis for the PSNR metric increases going down. The closer a data point is to the bottom left of the figure, the better the detector it represents is. *Figures (c) and (d)* present the *EEOP* values for DCT (similarly, Figures (g) and (h) for Sobel). Quality/Overhead figures always omit error detectors whose overhead is larger than 33%; recall that their *EEOP* is Infinity (worst case scenario)

error detection and correction with respect to the cycles required to execute the application reliably. Note that, both error detection and correction are executed under reliable conditions.

DCT

We construct the train and test data sets using a set of images frequently found in the literature [23], the validation data set contains images from the Image Compression data set [6].

Each task feature vector contains the 8 DCT coefficients of the $2x4$ block as well as its offset within the $8x8$ block of coefficients. All tasks but those which compute the DCT coefficients residing in the upper left corner of the $8x8$ block are computed unreliably. The upper left corner significantly affects the final output quality, as such, it would be highly inefficient to subject the respective computations to unreliable execution conditions [102]. An error-free execution of DCT has an output PSNR 35.6916 dB. The Quality (PSNR) and Overhead of the benchmark for all 7 tested ANNs and Topaz are illustrated in *Figure 6.3a*. The *EEOP* scores are shown in *Figure 6.3c*. We use the notation described in *Section 6.1.2* when we refer to a particular ANN. For example, (10,8,4,2) operates on a feature vector of size 10 and has two intermediate IP layers of sizes 8, and 4. Its output vector is the one-hot encoded vector of size 2.

All error checking mechanisms, with the exception of the least complex ANN $(10, 2)$, behave similarly in terms of quality. However, they incur different overheads in terms of cycles spent to detect and correct errors. The best mechanism in terms of *EEOP* is (10,8,4,2) which results in a near golden output PSNR at 35.6905dB; Topaz closely follows at 35.6595dB.

For the batched version of DCT, we aggregate the AOVs/feature vectors of 8 tasks into a single batch. The resulting Quality/Overhead measurements are shown in *Figure 6.3b*, *Figure 6.3d* presents the respective *EEOP* values for the result-checking mechanisms.

The overhead of Topaz after batching decreases by a more than a factor of $3x$, down to 4.70%, without any penalty to output quality. Similar results are observed for the efficiency of ANNs, however at a smaller magnitude. Overhead reduction for the best ANN (10,8,4,2) is negligible when we resort to batching because the False Positive Rate (FPR) increases by 1.49% compared to the non-batched version, thus, nullifying any performance improvements. Note that Detection Coverage (i.e. rate of correct positive prediction) is very high for both ANN and Topaz at $97,7\%$ and $95,5\%$, respectively. The $\overline{MRE}$ (i.e. the average non-detected error magnitude), however, is very high at 14% and 15%, respectively. In other words, the very few errors that escape detection, may be very detrimental to the output quality.

Sobel

The Sobel benchmark (*Figures 6.3e - 6.3h*) is a bad candidate for error detection using lightweight ANNs as well as Topaz. Both the best ANN (9,2) and Topaz suffer from relatively low *Detection Coverage* (67.3% and 64.7%), incur a high performance overhead (27% and 25.3%), and result in high $\overline{MRE}$ (undetected errors skew the output by 41.2% and 42.8% on average).

Performance and Quality improve when we consider a batch of 8 tasks, mainly because using larger task sizes results in a better *Detection Coverage* and lower aggregate overhead for most ANN configurations (but not for Topaz). *EEOP* for most ANN configurations is much lower than in the non-batched version, and lower than Topaz. In most cases, the batched ANN methodology performs better than Topaz

(A) Blackscholes Quality/Overhead

(B) Blackscholes (Batch) Quality/Overhead

(C) Blackscholes *EEOP* scores

(D) Blackscholes (Batch) *EEOP* scores

(E) Bonds Quality/Overhead

(F) Bonds (Batch) Quality/Overhead

(G) Bonds *EEOP* scores

(H) Bonds (Batch) *EEOP* scores

FIGURE 6.4: Blackscholes and Bonds evaluation results. The Quality/Overhead figures omit error detectors whose overhead is larger than 33%

both in terms of quality and performance overhead. Moreover, simpler ANNs such as (9,2) are almost as good as higher complexity ANNs.

Blackscholes

In Blackscholes, the training data set contains 400,000 assets and the testing-data set contains 40,000 assets. The validation data set comprises 100,000 assets. The three data sets are generated using a modified version of the PARSEC [8] Blackscholes input generator, which produces permutations of its bundled 2000 asset entries. Each data set is constructed using different data-ranges, so that the training/testing data and validation data are not the same. Each task is represented by a feature vector, of 8 values, which contains the task inputs and output.

Topaz imposes significant overhead (157.53%) to the execution time of the benchmark (note that figures showing the overheads of detectors such as *Figure 6.4a* omit

those with overheads larger than 33%). In contrast, all ANNs result in overheads ranging from 18.43% to 40.22%, without sacrificing quality as they also outperform Topaz in terms of output relative error. Even though both Topaz and the ANNs feature an *Detection Coverage* higher than 99.98% they have high $\overline{MRE}$s; 56.59% for Topaz, whereas for ANNs it ranges from 14.18% to 47.55%. The *EEOP* scores are presented in *Figure 6.4c*. Even though the $\overline{MRE}$ is in the same scale as Sobel, error detectors for Blackscholes are much better as is evident by their *EEOP* score, which is orders of magnitude lower (better).

The batched implementation of Blackscholes combines 10 tasks into a batch. All result-checking mechanisms get a reduction in overhead as shown in *Figure 6.4b* with an unnoticeable effect in quality, due to the fact that *Detection Coverage* is greater than 99.98%. However, $\overline{MRE}$ increases to more than 54.22% for all ANNs and drops to 55.28% for Topaz. The *EEOP* scores are shown in *Figure 6.4d*. Due to the overhead reduction and slight improvement of the $\overline{MRE}$, Topaz outperforms all ANNs.

Bonds

Bonds includes a random bond generator used to generate 440,000 bonds for the training and testing data sets. The validation data set is 100,000 bond prices. Similarly to Blackscholes, we generate the input data using different value ranges. The feature vector of a Bonds task comprises both its inputs and outputs. For this benchmark, an erroneous computation typically results in an infinite or NaN output. *Figure 6.4e* illustrates the quality/overhead measurements for Bonds. All mechanisms result in output relative errors lower than 0.002%. For the few false negatives (less than 0.34% of the total number of tasks) the $\overline{MRE}$ is relatively low, ranging from 1.27% to 6.8%.

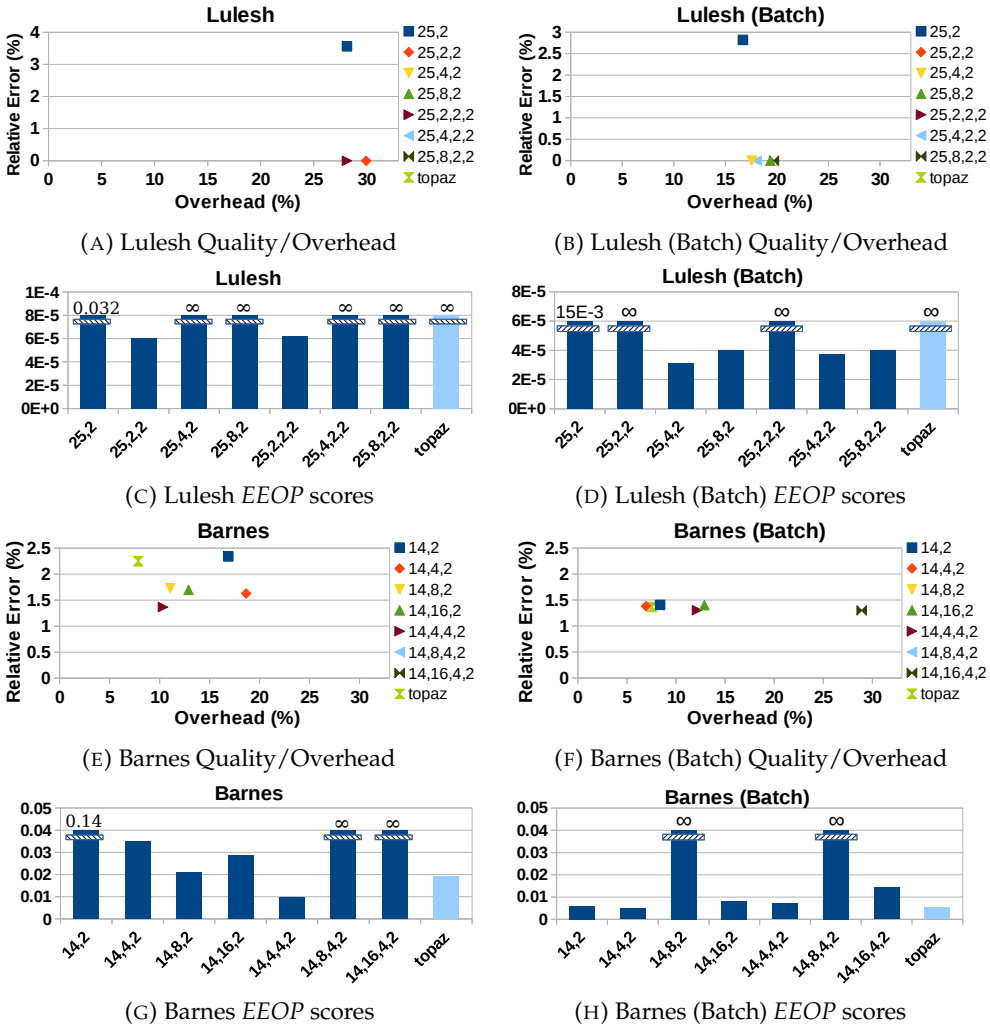
This application is a prime candidate for fault-tolerant computing. It results in a low overhead of error correction and detection with excellent output relative errors that are lower than 0.002%. *Figure 6.4g* shows the *EEOP* scores for the error detection mechanisms.

We implemented the batched version of Bonds by bundling 10 tasks into a batch before error detection. The overhead of the ANNs slightly improves at the cost of slightly worse $\overline{MRE}$ and *Detection Coverage*. The quality/overhead measurements can be seen in *Figure 6.4f*. Even though batching reduces the overhead of the detectors, their *EEOP* scores (*Figure 6.4h*) are slightly worse due to the slight deterioration of their classification performance.

Lulesh

The training data set profiles the execution of 4 different problem sizes[4] (N=5, 10, 15, and 20). The testing data set contains profile data for a problem size equal to N=18.

[4]In Lulesh, the problem size determines the number of elements involved in the computation, e.g the problem size of 10 involves 10^3 elements.



(A) Lulesh Quality/Overhead

(B) Lulesh (Batch) Quality/Overhead

(C) Lulesh *EEOP* scores

(D) Lulesh (Batch) *EEOP* scores

(E) Barnes Quality/Overhead

(F) Barnes (Batch) Quality/Overhead

(G) Barnes *EEOP* scores

(H) Barnes (Batch) *EEOP* scores

FIGURE 6.5: Lulesh and Barnes evaluation results. The Quality/Overhead figures omit error detectors whose overhead is larger than 33%

The validation input-data is a problem size of N=50. Feature vectors for Lulesh contain the output of a task, which is the computed forces for 8 bodies, along with the time since the beginning of Sedov blast (which is just a task input parameter). In this benchmark, all tasks are executed unreliably, apart from a random 10% which are always executed reliably for improved numerical stability.

The Overhead/Quality measurements of Lulesh are shown in *Figure 6.5a*. Lulesh tasks, much like those of Sobel, are too fine-grained for efficient error detection. Extreme granularity is the reason why just three out of the eight error detection mechanisms result in an overhead lower than 33%. *Figure 6.5c* presents the *EEOP* scores for the error detection mechanisms. Topaz overhead skyrockets to 208.45% but it produces results of high quality with a relative error of $3.1 * 10^{-42}\%$. On the other hand, (25,2,2) requires an additional 29.95% CPU cycles for error detection and correction and results in an output quality of $2.66 * 10^{-7}\%$. Regardless of the overhead cost, the *Detection Coverage* for all mechanisms is greater than 99.98%. Moreover, $\overline{MRE}$ of (25,2,2) is 2.62% and Topaz has an average $\overline{MRE}$ of $4.45 * 10^{-11}\%$.

The batched version of Lulesh aggregates 10 tasks into a single bundle prior to checking for errors. The output quality for (25,4,2) and Topaz are $3.32 * 10^{-7}\%$ and $1.95 * 10^{-28}\%$ respectively. With batching the cost of error detection is reduced, thereby driving the total Overhead in acceptable levels for all eight mechanisms (*Figure 6.5b*). The output relative error of the best six mechanisms is bellow $10^{-6}\%$. Interestingly, batching does not affect the $\overline{MRE}$, which remains below 3.5% for the top 6 mechanisms. The *EEOP* scores are shown in *Figure 6.5d*, with (25,4,2) being the top option.

Barnes

The distribution of Barnes [5] includes a number of input data sets. We use inputs describing systems with 1K, 2K, and 4K bodies as the training data set and the input of 8K bodies as the testing data set. The input set of 16K is used as the validation input data. A Barnes feature vector contains the outputs and inputs of the task. We present the Quality/Overhead measurements for Barnes in *Figure 6.5e* and the *EEOP* scores in *Figure 6.5g*. The best performing error detection mechanism is (14,4,4,2). It leads to an output relative error of 1.37% and an 10.33% overhead. Topaz, results in a relative error of 2.25% at a lower overhead of 7.89%.

The batched version of Barnes, combines 10 feature vectors/AOVS together prior to performing error detection. The results are shown in *Figures 6.5f and 6.5h*. The best mechanism is (14,4,4,2) with an output quality of 1.3%, overhead of 12.13% and $\overline{MRE}$ of 32.13. Topaz, follows closely, with an output quality of 1.36%, overhead of 7.55% and $\overline{MRE}$ of 33.55%. Note that, batching tasks in Barnes decreases the detection overhead, however it results in higher correction overheads due to a decreased recall rate for *Correct* tasks.

Inversek2j

We generate 1.1 million starting points to construct the training and testing data sets. For Inversek2j we include the inputs and outputs of each task in the feature vector. *Figure 6.6a* illustrates the Overhead/Quality measurements for the different error detection mechanisms. Because tasks of this application are exceptionally lightweight, Topaz detector leads to an overhead of 185.13%. On the other hand, all but one ANNs result in overheads lower than 33%. In all cases the average relative error is bellow $1 * 10{-}4\%$. *Figure 6.6c* illustrates the *EEOP* scores for the different mechanisms. The best scoring ANN (4,4,2,2) results in an overhead of 25.45%, a relative error of $5.17 * 10^{-6}\%$ and its $\overline{MRE}$ is 4.87%. Topaz, leads to a $1.17 * 10^{-5}\%$ relative error, and has an $\overline{MRE}$ of 10.12%. All result-checking mechanisms detect more than 99.99% of the errors. This is also illustrated in their *EEOP* scores: they are the best out of all the benchmarks.

(A) Inversek2j Quality/Overhead (B) Inversek2j (Batch) Quality/Overhead

(C) Inversek2j *EEOP* scores (D) Inversek2j (Batch) *EEOP* scores

(E) Bodytrack Quality/Overhead (F) Bodytrack (Batch) Quality/Overhead

(G) Bodytrack *EEOP* scores (H) Bodytrack (Batch) *EEOP* scores

FIGURE 6.6: Inversek2j and Bodytrack evaluation results. The Quality/Overhead figures omit error detectors whose overhead is larger than 33%

The batched version of this benchmark, aggregates 10 tasks into a bundle prior to checking for errors. The results of the fault-injection campaigns are shown in *Figures 6.6b* and *6.6d*. Because the error detection overhead is amortized across 10 tasks, the overhead is significantly reduced; for ANNs it ranges from 7.68% to 9.28%. With task-batching, Topaz becomes a viable choice for error checking as well. It imposes an overhead of 19.52%. Unfortunately, this performance improvement comes at the expense of the output quality of the application. The $\overline{MRE}$ for the ANNs ranges from 22.67% up to 27.15%, whereas Topaz performs slightly worse at 28.59%. Additionally, the *Detection Coverage* remains relatively unchanged, at more than 99.9% for all mechanisms.

Bodytrack

We use 40 such frames to construct the training data set and 4 for the testing data set. As validation input data we use a single frame. All frames are extracted from

a sequence of 262 frames that is distributed with the benchmark. Even though both Topaz and the ANNs detect nearly none of the *Incorrect* outputs, the output quality is only mildly affected. Specifically, the *Detection Coverage* score for all mechanisms ranges from 0.019% to 0.025% and the resulting relative error is less than 1% in all cases, as shown in *Figure 6.6e*. Moreover, true positives typically have a relative error of 17.16%-22.1% whereas false negatives, have an average relative error of 0.36% - 0.53%. This fact leads to low *Expected Error* scores, which translates to low output relative error. The latter, coupled with overheads less than 1.6% results in the relatively good *EEOP* scores shown in *Figure 6.6g*.

For the batched implementation we bundle 10 tasks together before checking for errors. Batching slightly deteriorates *Detection Coverage*, and reduces the overall overhead of ANNs by a factor of 10 but increases the overhead of Topaz to 2.78% (*Figure 6.6f*). The increased overhead of Topaz is mainly attributed to an increase of 5.9% to the number of false positives. Moreover, batching negatively affects the $\overline{MRE}$ increasing it to $1.4\% - 2.1\%$. As a result, the *EEOP*s for the batched version of Bodytrack are worse than the original version (*Figure 6.6h*).

In 7 out of our 8 benchmarks (excluding Bodytrack) the best scoring ANNs delivered an average *Detection Coverage* of 94.85% whereas Topaz scored marginally worse at 94.11%. Bodytrack, is an outlier because it mostly produces *Incorrect* task outputs which escape error detection due to them only slightly deviating from their respective *Correct* values. Consequently, neither the ANNs nor Topaz detect the majority of the errors but the end-result of the application has high quality (less than 1% relative error).

Both ANNs and Topaz are most efficient for applications with tasks that handle few data and execute for a long time. One good example of such an application is Bonds. Applications with tasks that handle large sets of data and require little execution time (i.e. Sobel) tend to behave poorly with this type of error detection, as their large feature vectors lead to an increased overhead for error detection. For such applications we expect that there are more appropriate alternatives to Multilayer Perceptrons, like Convolutional Neural Networks (CNNs) [51]. CNNs have been designed to exploit the spatial information of data contained within a feature vector. This information is implicitly defined by the indices of the data within the vector. A great use case scenario for CNN error detectors Layers would be multimedia applications.

6.3.2 Case study - Analysis for an unreliable configuration at the PoFF

Section 6.3.1 presented and discussed the performance overhead for error detection and correction by the ANN and Topaz methodologies. This subsection discusses a case study, deploying these methodologies in a fault-tolerant computing environment to speed up computation through CPU overclocking.

Figure 6.7 illustrates the voltage and frequency settings for a variety of operational configurations. A CPU operates nominally (i.e. error free) on a (V, f) line.

The CPU may dynamically move to multiple configurations between a lower performance point (V_{low}, f_{low}) and a higher performance point (V_{high}, f_{high}) under the control of the Operating System (OS). For example, when the workload is low or memory-bound, the OS can switch the CPU core(s) closer to the (V_{low}, f_{low}) point to save power.

For our experiments on the Intel i7 4820k CPU, we set (V_{high}, f_{high}) = (1.06 V, 3.7 GHz) and (V_{low}, f_{low}) = (0.9 V, 1.67 GHz). The latter point is the reliable configuration, whereas the (V_{low}, f_{high}) = (0.9 V, 3.7 GHz) point is set to be the PoFF (note that $0.9 \approx 0.85 * 1.06$) and falls well into the unreliable area (*Figure 6.7*) due to overclocking. Overclocking provides lower execution time under unreliable conditions, which may result in crashes or SDCs. Error detection and correction mechanisms are then deployed to alleviate the effects of unreliable configurations, at the expense of performance (as shown in 6.3.1). It is this interplay between output quality and performance (faster clock vs. correction/detection overhead) that we evaluate in this subsection. Note that, in Section 3.9 we evaluated the use of unreliably configured hardware to optimize the power and energy efficiency of applications. In this Section we utilize unreliability in a slightly different way. We evaluate whether we can reliably operate the CPU below the line of nominal operation to reduce execution time using the proposed error detection technique.

We also present the speedup obtained through the use of an unrealistic oracle-like error detection mechanism. We synthetically calculate the expected speedup of the Oracle using three simple rules: the Oracle has a) perfect *Detection Coverage* (TPR), b) zero FNR, and c) zero detection overhead. Consequently, the Oracle results in the best possible speedup assuming that all errors are corrected. *Table 6.2* compares the best ranking ANNs and Topaz against the Oracle for each benchmark.

The speedup baselines are error-free executions of the applications using the fully reliable voltage/frequency configuration of (V_{low}, f_{low}). Mixed-reliability applications schedule the unreliable tasks on cores which operate under the unreliable

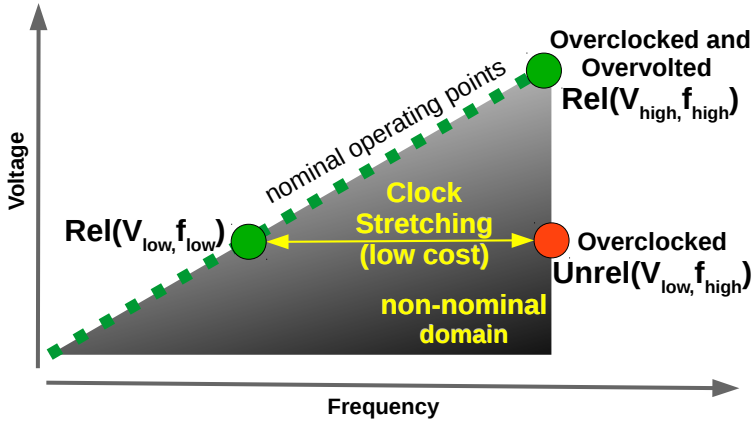


FIGURE 6.7: Reliable and Unreliable configurations

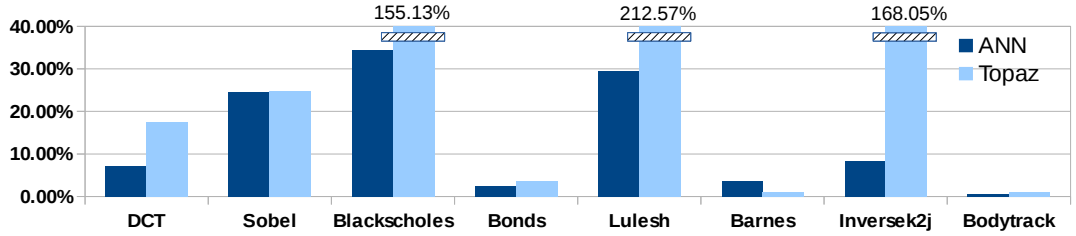


FIGURE 6.8: The overhead of performing error detection defined as the number of cycles spent to perform error detection over the number of cycles required for a fully reliable execution of the application

voltage/frequency configuration of (V_{low}, f_{high}). The average speedup of the theoretical Oracle error detector is 1.99x and the resulting output is by definition bitwise exact since the Oracle always detects an *Incorrect* task output. Even though both ANNs and Topaz exhibit similar behavior in terms of *Detection Coverage*, the ANNs exhibit higher speed up at 1.51 vs. 1.15 for Topaz. The difference is mainly due to the higher error-detection overhead in Topaz (which is true also in the batched version of the benchmarks). Note that Bodytrack, is the only benchmark for which an ANN results in higher speedup than the Oracle error detector, because neither ANNs nor Topaz perform task re-execution to correct the majority of the *Incorrect* task outputs.

The best scoring ANNs in batched configuration, on average, achieve an execution speedup of $1.72x$ which is larger than the 1.44x achieved by Topaz. Again, the difference is mainly due to the high error detection overhead of Topaz for most benchmarks. *Figure 6.8* shows that the average error detection overhead with respect to the execution time of the application on a reliable CPU is 13.83% and 72.98% for the optimal ANN (the one with the lowest *EEOP*), and Topaz, respectively. Three benchmarks (Blackscholes, Lulesh, inversej2k) have such a high detection overhead, that the whole application is slowing down as shown in *Table 6.2*. For these benchmarks, Topaz is only a viable approach if task-batching is utilized to amortize the cost of error detection. A similar trend is observed across all benchmarks: performing error detection at a coarser level via batching reduces the average overhead of error detection. More specifically, the cost of error detection for ANNs after batching comes to 6.45% and for Topaz to 12.81%.

We showed that it is possible to rely on using ANNs for automatic result-checking which outperforms the previous state of the art approximate outlier detector Topaz [1]. For most applications evaluated, an ANN incurs less computation overhead and results in either better or equivalent output quality compared to Topaz. This is reflected in the *EEOP* scores of the error-detectors. ANNs score better (lower) compared to Topaz for 7 out of the 8 non-batched versions of the benchmarks. The trend is also present in the batched version of the benchmarks: ANNs outperform Topaz in 6 out of 8 benchmarks.

Moreover, we observed that batching can significantly reduce the overhead of error detection. Unfortunately, that comes with a cost to the $\overline{MRE}$. This increased $\overline{MRE}$ does not severely affect the overall output quality of applications because

		Original		Batched	
Benchmark	Oracle	ANN	Topaz	ANN	Topaz
DCT	1.88	1.54	1.42	1.51	1.74
Sobel	2.16	1.36	1.40	1.53	1.40
Blackscholes	2.11	1.17	0.51	1.72	1.62
Bonds	2.14	2.03	1.99	2.10	1.20
Lulesh	1.93	1.21	0.35	1.48	1.16
Barnes	1.82	1.49	1.77	1.64	1.75
Inversek2j	2.06	1.39	0.43	1.85	1.41
Bodytrack	1.82	1.90	1.31	1.92	1.26
Average	1.99	1.51	1.15	1.72	1.44

TABLE 6.2: Comparison between the execution speedups achieved through overclocking in conjunction with Artificial Neural Networks, Topaz, and Oracle error detector. The baseline is the time required to complete an error-free execution under the reliable configuration (V_{low}, f_{low}) = (0.9 V, 1.67 GHz). Overclocking results in the execution of unreliable tasks under the configuration (V_{low}, f_{high}) = (0.9 V, 3.7 GHz)

the *Detection Coverage* of the error detectors tends to remain very high even after aggregating multiple feature vectors into a single one via batching.

Our methodology enables a number of interesting future research directions for designing dynamic runtime systems. For example, one can automatically generate a number of error detectors which vary in terms of overhead and resulting output quality. At execution time, the runtime system can choose the best error detector out of many, depending on a) their *EEOP*, b) $\overline{MRE}$ scores, and c) user supplied constraints such as a maximum energy budget for error detection. Another possible research direction would exploit the low execution overhead and small size of ANNs for error-detection. At execution time a runtime system could choose appropriate occasions to fine-tune an ANN in order to trade-off additional overhead for increased *Detection Coverage*. The overhead of fine-tuning the ANN can be limited using the principles of Elastic Weight Consolidation [48] (EWC). EWC would allow an ANN to only change some of its weights to improve its classification performance at a reduced computational cost.

Finally, recall that the ANNs that we employ comprise a small number of different operations. Additionally, we need ANNs to be executed reliably so that the runtime system can trust ANNs to detect errors at the outputs of tasks which have been executed under unreliable conditions. Interestingly, the ANNs that we employ involve a rather small number of different operations. As such, it would be possible to harden this subset of computations and as a result reduce the computational cost of error detection even further.

Chapter 7

Related work

This chapter discusses related work and the differentiation between our research and previous efforts. As approximate and unreliable computing have been very hot research subjects lately, we organize literature discussion across three different axis: Approximate computing, Unreliable computing, and Power & energy-aware optimization.

7.1 Approximate computing

Quickstep [62], 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, QuickStep does not enable algorithmic and application-specific approximation and does not include energy-aware optimizations in the runtime system.

Variability-aware OpenMP [73] and variation tolerant OpenMP [74], 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 [4] 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. [92] 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 [60] 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 [90] 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 [110] 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 [109] 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 [85] is a compiler and runtime environment for automatic generation of approximate kernels in machine learning and image processing applications. Paraprox [84] implements transparent approximation for data-parallel programs by recognizing common algorithmic kernels and then replacing them with approximate equivalents. ASAC [81] 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 [104], neural networks that approximate the results of code regions in hardware [21], and low-voltage probabilistic storage [82]. 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.

7.2 Fault tolerant computing

Gschwandtner et al. [28] 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. [88] 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 [63] selects approximate kernel operations to minimize an application's energy consumption while satisfying its accuracy specification.

Rinard *et al.* [79], in one of the chronologically earlier efforts on task-based error-tolerant 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 [74]) 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.* [76] 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. [83] 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.

[77] 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 [49] 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 [18] 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 [33]. A number of simple, yet effective code transformations that reduce error rate are introduced.

In [38] 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 [52]. 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 [86] 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 [19]. 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 [22, 58, 16, 50]. 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 [50] 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 [31, 27, 44] rely on manually implemented Light-Weight Checks (LWCs) to detect errors at the outputs of computations. [27] use manual LWCs to determine when an approximate alternative to a function computes outputs which severely differs from the exact implementation. [44] relies on manually implemented LWCs to detect errors on the output of unreliably executed code. [31] 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 [80]. 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.

[47] 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 [47] 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.

7.2.1 Power and Energy-Aware Optimization

Dynamic quality control of non-functional application properties including power and energy has been explored in HeartBeats [34], a framework for user-directed execution steering; PowerDial [35], an environment for adapting applications to execute efficiently under power and load fluctuations; Metronome [91], an operating system substrate for dynamic performance and power management; and the Angstrom processor [36], 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 [14], message

passing and hybrid models [55, 56], new parallel programming languages that natively support transparent adaptation to dynamic execution conditions such as [94] as well as distributed programming frameworks [40].

Cohen in Energy types [11] 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.

Chapter 8

Concluding remarks

8.1 Retrospective

In the previous chapters we discussed our work on significance aware approximate and fault tolerant computing. We began by identifying the core challenges for both computing paradigms and formulating them into two sets of intuitive questions.

The very first challenge is to answer the question of "**How** to implement and execute applications for significance-aware computing?". We designed and implemented a programming model which is user friendly but is also expressive enough to support the implementation of applications using the principles of significance-aware computing. We chose to base our design on the popular OpenMP task-based programming model. Tasks are an intuitive vehicle of computations, and they also facilitate the compartmentalization of code. This particular characteristic is extremely helpful especially in the fault-tolerant flavor of significance-aware computing. After all, the significant portions of an applications must not be affected from errors which occur during the execution of the least significant ones.

A versatile programming model must also be accompanied by intelligent runtime system support, which efficiently exploits the opportunities to gracefully trade-off output quality with program optimization. To this end we designed and implemented two runtime systems, for significance-aware approximate as well as fault-tolerant computing.

Beyond the framework enabling the implementation and execution of significance-aware applications we also introduce methodologies which aim to reduce the human effort involved in implementing such applications for approximate and fault tolerant computing. One of the key challenges that we targeted is "**What** to approximate/execute unreliably?" and "**How** to approximate?". Answering these questions requires an automatic significance analysis methodology which is applied on the code of applications. To this end, we showed that it is possible to automatically provide a qualitative assessment of different parts of a code with respect to their contribution to final output quality, for applications whose code is differentiable.

Our work to that point provided the means to implement and execute applications – significance-characterized at the granularity of tasks – using either software-only approximations or unreliable hardware to optimize their execution. We then

considered **when** significance-aware approximate/fault-tolerant computing is actually necessary. It makes sense to resort to significance-aware computing when a user specified constraint, i.e. energy budget, cannot be met with conventional computing paradigms. This, in turn, raises the question "**How much** of an application should be approximated/executed unreliably to meet a specified user constraint?" We modeled the performance of applications to deduce the minimum degree of approximation so that the resulting energy footprint is a fraction of the footprint when executing the application in the most efficient and fully accurate configuration. Of course, our approach can be augmented by including energy models for fault tolerant computing and is thus applicable to this computing paradigm as well.

Finally, we targeted the last key challenge of significance-aware fault tolerant computing, which is also arguably the most difficult one. **How** to detect errors? To this end, we use a runtime system which uses OS standard services, as well as runtime watchdogs to identify “noisy” indications of errors (such are crashes, hangs etc.). Silent Data Corruptions (SDCs), on the other hand, are much more challenging; they can propagate through computations and potentially to the output without producing “noisy” symptoms. Therefore, the application developer, or ideally the runtime system, need to actively monitor for such errors in appropriate places of the computation graph. The solution to this problem is difficult to automate in a generic way. In this Thesis we focused on automatically generating error detectors to detect errors at the outputs of application tasks. Our methodology trains Artificial Neural Networks (ANNs) which automatically detect errors that have propagated at the outputs of tasks. Our approach involves only limited human effort, has proven highly effective, and resulted in low computation overhead.

8.2 Conclusions

Significance of computations is an algorithmic property which can serve as the supporting basis for effective approximate and fault-tolerant computing. One of the goals of this Thesis was to elevate significance to a first-class concern during algorithm and application development, in par with, for example, parallelism. The challenges of these computing paradigms can be largely addressed by frameworks which make intelligent dynamic decisions which would otherwise have to be made statically by humans. In fact, we have shown that it is possible to automate the process of selecting the appropriate level of approximation to meet user constraints, as well as the process of detecting errors in intermediate results using machine learning techniques.

Investing in approximate computing eases the transition to fault-tolerant computing for future systems. Unless assisted by software, the task of implementing algorithms using the fault-tolerant computing programming paradigm can be exceptionally taxing on the application developer. Therefore, novel software frameworks/tools are required to lift some of the burden of implementing algorithms which

are expected to execute under partially unreliable conditions.

Much of software development is performed today in the form of reusing pre-packaged software components. Essentially, unreliable computing requires tools to facilitate the implementation and adoption of software in the form of libraries. Beyond custom developer code, libraries also need to carry extra information such as their robustness guarantees, methods for error-checking, energy/performance efficient mechanisms for error-correction, and functions to judge the quality of output of library functions.

This thesis, concludes with the following suggestion towards the realization of fault-tolerant computing on next generation hardware:

The transition to fault-tolerance requires effort from the side of application developers, to design their algorithms to operate under unreliable conditions. Of course, not all parts of an application are amenable to fault-tolerant execution. There are critical regions of software which would significantly deter software behavior and expected output in the event that they exhibit some kind of error. Examples of such kinds of software are critical regions of operating systems, runtime systems, multi-threaded synchronization, etc. Such software regions need to be executed in a reliable way, even if such a decision implies that their execution suffers in terms of performance/energy/power. Fortunately, there are classes of applications the bulk of computations of which have inherent fault-tolerance characteristics. The largest portions of such software can benefit from execution on hardware which is configured beyond its nominal point and rely on principles of fault-tolerant computing to deliver acceptable outputs.

To this end, computations need to be executed using a heterogeneous platform which comprises mixed-reliability hardware. Ideally the hardware should be configurable in two reliability modes:

a) best-effort mode of (pseudo-)reliable operation. Similar to the reliability level of modern commodity hardware, errors are highly improbable and infrequent and the hardware/software stack detects and corrects all those errors so that the execution of code is predictable.

b) fault tolerant operation. At this configuration the hardware operates at its highest potential of energy efficiency/performance but may result in the manifestation of errors.

Figure 8.1 illustrates our approach towards realizing our vision of significance aware fault tolerant computing. We consider a computing platform that comprises two kinds of processors. One type operates reliably, much like current commodity hardware. However, the second type of hardware is optimized for performance- and energy-efficient fault-tolerant computing. In other words, it will enable developers to dynamically change its configuration between nominal and points beyond

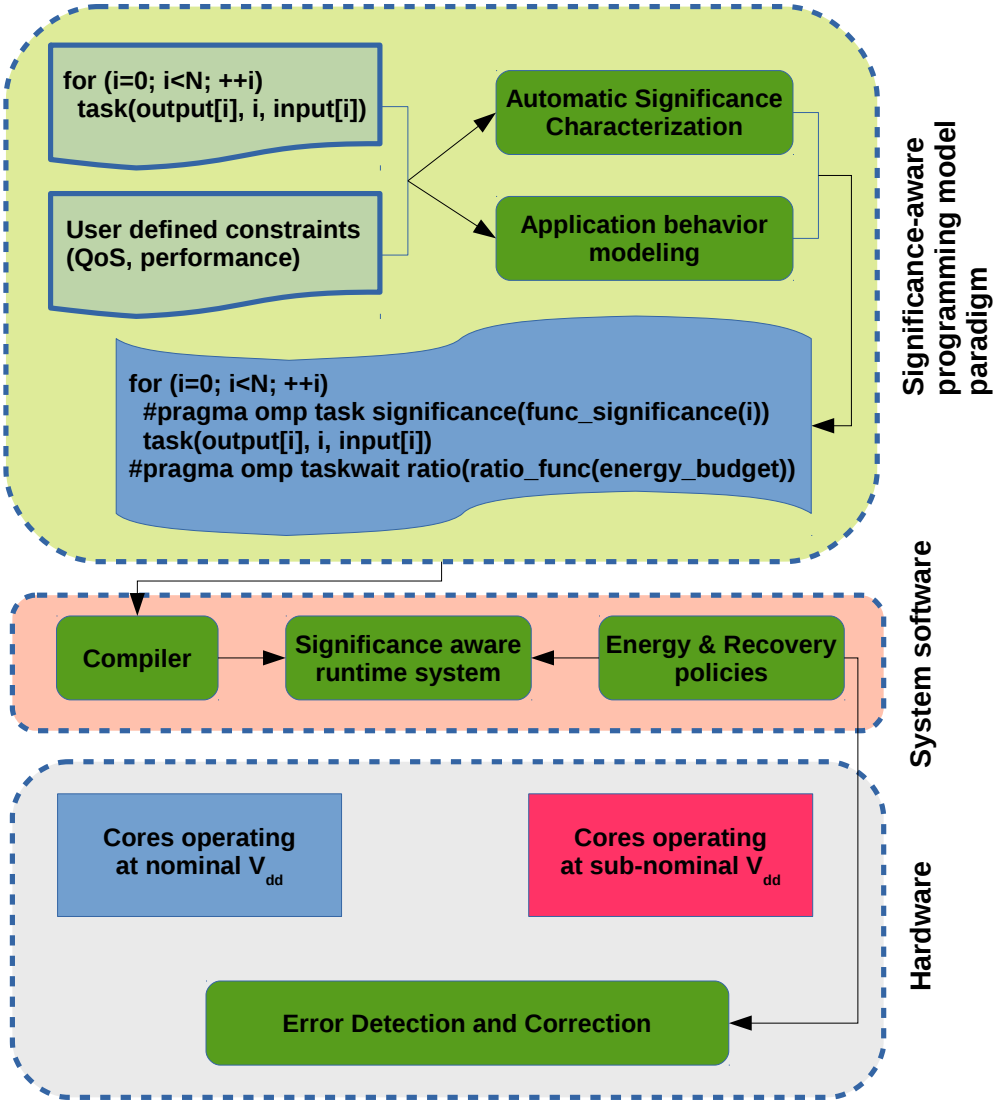


FIGURE 8.1: Envisioned design for significance-aware fault tolerant computing on mixed-reliability heterogeneous platforms.

its nominal operation. This way it will be possible to create opportunities for improved performance and power, as well as energy consumption, using intelligent significance aware system software.

All critical software regions, including the bulk of the operating system, and fault-tolerant runtime system, will execute on the reliable processor(s). The computation intensive regions of a program which are amenable to fault-tolerant computing can be executed using the unreliable processor(s). This approach is similar to how GPU-assisted heterogeneous computing works. GPUs are used to optimize the performance of applications by executing code which is embarrassingly parallel. In analogy, in our case we propose the use of unreliable hardware for the least significant parts of applications. The main difference is that it takes a few more steps to harden applications so that they always produce results of acceptable quality.

8.3 Future work

There are several open research opportunities regarding our work on significance aware approximate and fault tolerant computing. We intend to investigate the feasibility of significance analysis of source code at the level of libraries. This kind of improvement would enable developers to reuse code already annotated for significance.

Another 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 underutilize the hardware or even produce output of lesser quality.

During the early stages of our automatic error detection methodology we opted to use re-execution to correct the detected errors at the outputs of idempotent tasks. It makes sense to pay the cost of re-executing a task as long as its output contains errors which would significantly reduce the final output quality unless corrected. However, the selection of the error correction methodology is also an opportunity to explore the output quality vs program optimization trade-off. If the error-correction is too complex (with reliable re-execution being the worst case scenario) and too frequent the benefits of unreliable execution can be significantly hindered. Given that we have explored the use of approximate error-detectors we also aim to investigate the use of approximate task-alternatives as a form of error correction to reduce the overall cost of error-detection and correction.

We are also considering ways to extend our methodology for automated error detection using Artificial Neural Networks (ANNs). For example, we plan to modify our approach to target errors which occur when executing code on a specific hardware architecture. Architecture-specific ANNs could then be more effective in capturing the effects of executing code on that particular hardware platform. We also plan to study a stronger integration of ANNs with runtime systems for significance aware fault tolerant computing. Our current approach, treats all detected errors as equally important. We want to investigate whether it is possible to relax the requirements of error detection even more, to further optimize the execution of applications. To this end, we envision a runtime policy which takes into account the confidence of the error detector together with the user specified energy/quality constraints.

Finally, we are also planning to study the effects of unreliability on the quality and performance of approximate alternatives to exact code. We have shown that it is possible to optimize the execution of applications at a small penalty to their output quality via either significance aware approximate or fault tolerant computing. We wish to explore the scenario of executing the approximate alternatives of tasks under

unreliable conditions and analyse the resulting behavior of applications.

Related publications

[1] Vassilis Vassiliadis, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas. Artificial Neural Networks for online error detection. *(Under review)*

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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 [100, 71, 102, 101, 99, 98]. This appendix discusses my contribution to each of the aforementioned publications.

In [98] and [71] I contributed to the design of the significance aware programming model for fault tolerance computing. Additionally, I contributed to the process of benchmarking, as well as the analysis of the results. I also modified an existing source-to-source compiler [108] to augment it with significance-related extensions. These extensions facilitate the development of applications using the significance aware programming model.

In [102] I designed a methodology to utilize the `dco/scorpio` tool and exploit the significance analysis of computations to automatically partition computation graphs into tasks when significance characterization at the task granularity is the primary concern, as well as to provide hints to an application developer about regions of code which are amenable to approximation. The resulting method is described in detail under Section 4.1. Furthermore, I implemented the benchmarks as well as the supporting runtime system, applied our methodology, and evaluated it.

For [101, 99], I contributed the analytical models, designed and applied the machine learning approach, 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, in [100] I was responsible for designing the methodology and training the Artificial Neural Networks. I also modified the significance aware fault-tolerant computing runtime system of [71] so that instead of simulating undervolted hardware configurations it simulates overclocked ones. Finally, I was in charge of performing the benchmarking and analysis on the results of the software fault injection experiments.

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] Yannis M Assael et al. "LipNet: Sentence-level Lipreading". In: *arXiv preprint arXiv:1611.01599* (2016).

[3] Utku Aydonat et al. "An OpenCL™ Deep Learning Accelerator on Arria 10". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021738. URL: http://doi.acm.org/10.1145/3020078.3021738.

[4] 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.

[5] Josh Barnes and Piet Hut. "A hierarchical O (N log N) force-calculation algorithm". In: *nature* 324.6096 (1986), pp. 446–449.

[6] Axel Becker et al. *Image Compression benchmark*. 2015. URL: http://imagecompression.info/test_images (visited on 09/18/2015).

[7] Nikolaos Bellas et al. "Real-time fisheye lens distortion correction using automatically generated streaming accelerators". In: *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines*. FCCM '09. IEEE Press, 2009, pp. 149–156. ISBN: 978-0-7695-3716-0. DOI: 10.1109/FCCM.2009.16. URL: http://doi.org/10.1109/FCCM.2009.16.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] 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.

[17] Alejandro Duran et al. "Ompss: A Proposal for Programming Heterogeneous Multi-core Architectures". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[18] 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.

[19] 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.

[20] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 3. ACM. 2011, pp. 365–376.

[21] 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.

[22] 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.

[23] *Frequently found test images in literature*. URL: http://www.imageprocessingplace.com/downloads_V3/root_downloads/image_databases/standard_test_images.zip.

[24] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[25] Scott Grauer-Gray et al. "Accelerating financial applications on the GPU". In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM. 2013, pp. 127–136.

[26] A. Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd. SIAM, 2008.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

[41] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[42] J. E. Jones. "On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature". In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 106.738 (1924), pp. 441–462. ISSN: 0950-1207. DOI: 10.1098/rspa.1924.0081.

[43] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit^{TM}". In: *Proceeding of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. 2017.

[44] 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.

[45] 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.

[46] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. 2013, pp. 1–9.

[47] 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.

[48] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the National Academy of Sciences* (2017), p. 201611835.

[49] 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.

[50] 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.

[51] Yann LeCun et al. "LeNet-5, convolutional neural networks". In: *URL: http://yann. lecun. com/exdb/lenet* (2015).

[52] 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.

[53] Michael Lerch et al. "FILIB++, a fast interval library supporting containment computations". In: *ACM Trans. Math. Softw.* 32.2 (2006), pp. 299–324.

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

[55] 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.

[56] 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.

[57] Johannes Lotz et al. "Higher-order Discrete Adjoint ODE Solver in C++ for Dynamic Optimization". In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, Computational Science at the Gates of Nature, pp. 256–265. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.237. URL: http://www.sciencedirect.com/science/article/pii/S1877050915010455.

[58] 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.

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

[60] 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.

[61] Paul Mineiro. *fastapprox*. http://code.google.com/p/fastapprox/. 2012.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Vol. 24. Siam, 2012.

[68] Uwe Naumann et al. "Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations". In: *ACM Trans. Math. Softw.* 41.4 (2015), 26:1–26:21. ISSN: 0098-3500. DOI: 10.1145/2700820. URL: http://doi.acm.org/10.1145/2700820.

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

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

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

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] Hermann Schichl and Arnold Neumaier. *Interval Analysis on Directed Acyclic Graphs for Global Optimization*. Tech. rep. J. Global Optimization, 2004.

[88] 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.

[89] 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.

[90] Stelios Sidiroglou-Douskos et al. "Managing Performance vs. Accuracy Tradeoffs 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.

[91] 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.

[92] 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.

[93] Software and Germany Tools for Scientific Engineering RWTH Aachen University. *Derivative Code by Overloading in C++ (dco/c++)*. http://fsnew.stce.rwth-aachen.de/research/software/dco-c.

[94] 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.

[95] Jan Treibig, Georg Hager, and Gerhard Wellein. "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments". In: *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE. 2010, pp. 207–216. ISBN: 978-0-7695-4157-0. DOI: 10.1109/ICPPW.2010.38. URL: http://dx.doi.org/10.1109/ICPPW.2010.38.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] Vassilis Vassiliadis et al. "Artificial Neural Networks for online error detection". In: *ACM Trans. Archit. Code Optim.* xx.x (2017), 0:0–0:0. ISSN: 0000-0000. DOI: 10.1145/0. URL: http://doi.acm.org/10.1145/0.

[101] 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.

[102] 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.

[103] Manolis Vavalis and George Sarailidis. *Hybrid-numerical-PDE-solvers: Hybrid Elliptic PDE Solvers*. http://dx.doi.org/10.5281/zenodo.11691. 2014. DOI: {10.5281/zenodo.11691}.

[104] 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.

[105] Nicholas J. Wang et al. “Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline”. In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. DSN ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 61–. ISBN: 0-7695-2052-9. URL: http://dl.acm.org/citation.cfm?id=1009382.1009722.

[106] Steven Cameron Woo et al. “The SPLASH-2 programs: Characterization and methodological considerations”. In: *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE. 1995, pp. 24–36.

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

[108] 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.

[109] 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.

[110] 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.