Queen's University Belfast

Thesis Deposit Form

To be completed by Student

Degree:	PhD
Full Name:	Charalampos Chalios

Thesis title:

Software-defined Significance-Driven Computing

Summary: (max. 300 words)

Approximate computing has been an emerging programming and system design paradigm that has been proposed as a way to overcome the power-wall problem that hinders the scaling of the next generation of both high-end and mobile computing systems. Towards this end, a lot of researchers have been studying the effects of approximation to applications and those hardware modifications that allow increased power benefits for reduced reliability. In this work, we focus on runtime system modifications and task-based programming models that enable software-controlled, user-driven approximate computing.
We employ a systematic methodology that allows us to evaluate the potential energy and performance benefits of approximate computing using as building blocks unreliable hardware components. We present a set of extensions to OpenMP 4.0 that enable the programmer to define computations suitable for approximation. We introduce task-significance, a novel concept that describes the contribution of a task to the quality of the result. We use significance as a channel of communication from domain specific knowledge about applications towards the runtime-system, where we can optimise approximate execution depending on user constraints.
Finally, we show extensions to the Linux kernel that enable it to operate seamlessly on top of unreliable memory and provide a user-space interface for memory allocation from the unreliable portion of the physical memory. Having this framework in place allowed us to identify what we call the refresh-by-access property of applications that use dynamic random-access memory (DRAM). We use this property to implement techniques for task-based applications that minimise the probability of errors when using unreliable memory enabling increased quality and power efficiency when using unreliable DRAM.

To be completed by Examiner

EXAMINER CERTIFICATION OF SUBMITTED WORK

I hereby certify that this is the final accepted copy of the submitted work and that all required amendments have been completed and submitted within the required deadline.

Name of Examiner: Dr. Ivor Spence

Signature of Examiner: [illegible]

Date: 11/12/2017

To be completed by Student

Student Declaration:

I give permission for my thesis to be made available, under regulations determined by the University, for inclusion in the University Library, consultation by readers in the School, inter-library lending for use in another library and to be photocopied, electronically reproduced and to be stored and made available publicly in electronic format

Please tick as appropriate:

i) Immediately ☑

Or

ii) After an embargo period of 1 year ☐ 2 years ☐ 3 years ☐ 4 years ☐ 5 years ☐

Reason for embargo: (applies to both print and e-thesis)

☐ The thesis is due for publication, either as a series of articles or as a monograph

☐ The thesis includes material that was obtained under a promise of confidentiality

☐ Would substantially prejudice the commercial interests of the author, the University or an external company

☐ Contains information which may endanger the physical/mental health or personal safety of an individual(s)

I wish to embargo the e-thesis copy permanently: (applies to e-thesis only)

☐ The thesis contains material whose copyright belongs to a third party and the gaining of approval to publish the material electronically would be onerous or expensive; and the removal of the copyright material would compromise the thesis

---

CONFIRMATION OF DATA/HUMAN TISSUE SAMPLES HANDOVER

All research involving human participants, their tissue (e.g. blood, saliva, urine) or their data (interviews, consent forms, questionnaires) must be retained by the University for at least five years. These sources must be handed over to your supervisor. Please confirm, by ticking the appropriate box, that:

☐ I have provided my supervisor with all laboratory notebooks and/or primary source material pertaining to the study, including electronic data.

☐ I have identified for my supervisor the location of stored human tissue samples and provided an inventory of these.

✓ Due to the nature of the project I am not required to handover any data or samples relating to my thesis.

Signature of Student: [illegible] **Date:** 10/12/2017

To be completed by Supervisor

SUPERVISOR CONFIRMATION AND APPROVAL

I approve any embargo request above and confirm that the information given regarding Data/Human tissue samples is correct and that, where applicable*, the final copy of the thesis has also been made available in electronic format (e-thesis) via PURE.

Name of Supervisor: **Prof Dimitrios S. Nikolopoulos**

Signature of Supervisor: [illegible] **Date:** **11/12/2017**

* RCUK funded students and all students who commence research from September 2016 must also submit an electronic copy of their thesis via Pure

Software-defined Significance-Driven Computing

Charalampos Chalios

A thesis presented for the degree of

Doctor of Philosophy

School of Electronics, Electrical Engineering and Computer Science

Queen's University of Belfast

United Kingdom

Wednesday 20th December, 2017

I dedicate this to my family, my friends and my teachers.

Acknowledgements

The research included in this thesis was conducted during my stay in the School of Electronics, Electrical Engineering and Computer Science of Queen's University of Belfast in United Kingdom. Queen's welcomed me and offered me everything a young researcher needs in order to be able to carry out the challenging task of doing research and writing a thesis for the degree of Doctor of Philosophy. Moreover, I have to acknowledge the Department of Employment and Learning for my scholarship and the European Commission which through EU projects funded my research.

I would like to express my deep gratitude for my mentor Professor Dimitrios S. Nikolopoulos who was the first person to believe in me, in my life as a researcher, and gave me the opportunity to develop my skills and knowledge. He has been a role model for me as a researcher and through his guidance I believe I have become a better scientist. Moreover, I would like to thank my supervisor Dr. Hans Vandierendonck who has been a true teacher to me. His clarity of thought, technical integrity and most importantly his inexhaustible patience in transferring knowledge has been invaluable during the course of my studies.

I would like also to thank my colleagues in Belfast and Athens who have been great collaborators and friends. From the CSLAB of the School of ECE of the National and Technical University of Athens, Dr. Nikos Anastopoulos, Dr Kostis Nikas and Dr Georgios I. Gkoumas that were the people that first stirred in me the interest in research and till today have been helping me in every step of my research. Dr. Giorgis Georgakoudis, who was the first friend I made in Belfast and has been there for me from the first day of my PhD till now. Our discussions, more often than not heated, have been particularly inspiring and moments of true relaxation during the burdens of the academic life. Finally, I would like to thank Dr. Lev Mukhanov, Dr Kiril Dichev, (soon to be Dr.) Kosta Tovletoglou and Dr. Georgios Karakonstantis for the collaboration and the nice moments we had over that pint of Guinness after work.

I feel sincerely grateful for all the beautiful people I met during this trip, who honoured me with their friendship. Life in Belfast would not have been the same were it not for number 13 Chadwick Street. I would like to thank Davide, Ioanna, Leonardo and Lisa for their friendship, their tolerance of my anti-sociality, the dinners, the games, the trips and the BBQs. Athanasia thank you for showing me, even maybe without realizing it, that there are few things you cannot do if you really love them. You are a beautiful person and valuable friend. Miguel you are one of the most clever, humorous, creative and good-hearted people I have ever met. Stephania, thank you for some of the most insteresting, constructive and heated discussions I have had over a glass of wine. Vincenzo and Giuliano I am really greateful for all the beautiful moments and the fact that you made Belfast look a bit more like home. Mauro, thanks for the relaxing and creative nights on this red sofa under the Bob Marley painting. Finally, Ntina thanks a lot for your hospitality and really good heart. No matter how far away, I am really happy you are my friends.

Circumstances made a lot of us to move around the world, in order to find whatever it is we were looking for. Even if we weren't physically close to each other, I need to thank these people that over the years we didn't let distance to keep us apart. Taso, Panteli and Odyssea, in a sense if it weren't for you I would probably be still back to Athens trying to pass that SAE course. Polyzo, Kosma, Vasili, Riga and Mike thanks for the studying, drinking, eating, playing, etc. during these beautiful 7 years. Mikou, special thanks to you for all the early coffees and late drinks. Finally, Athina, Thoma, Alexandre (Takou), thank you for these, brief but beautiful, moments we spent together in "our house" and out of it, and deeply changed who I am.

Many thanks I owe to my parents, Kostas and Rodokleia, and my brothers Stratos and Giorgos for their unconditional love and support throughout my life. You have been always there for me through good and bad moments and I know that I would never have achieved as much in life if it were not for you.

This whole journey has been even more beautiful because of one person. Paola, my life partner, has been there every single step of the way. Somehow, she managed to make these three years the most beautiful years of my life. Being the most strong person I have ever met, she helped me keep going, during the most difficult and hard moments of this endeavour. Thank you for all your support and patience.

Abstract

Software-defined Significance-based Computing

by

Charalampos Chalios

A thesis presented for the degree of Doctor of Philosophy

School of Electornics, Electrical Engineering and Computer Science

Queen's University of Belfast

April 2017

Approximate computing has been an emerging programming and system design paradigm that has been proposed as a way to overcome the power-wall problem that hinders the scaling of the next generation of both high-end and mobile computing systems. Towards this end, a lot of researchers have been studying the effects of approximation to applications and those hardware modifications that allow increased power benefits for reduced reliability. In this work, we focus on runtime system modifications and task-based programming models that enable software-controlled, user-driven approximate computing.

We employ a systematic methodology that allows us to evaluate the potential energy and performance benefits of approximate computing using as building blocks unreliable hardware components. We present a set of extensions to OpenMP 4.0 that enable the programmer to define computations suitable for approximation. We introduce task-significance, a novel concept that describes the contribution of a task to the quality of the result. We use significance as a channel of communication from domain specific knowledge about applications towards the runtime-system, where we can optimise approximate execution depending on user constraints.

Finally, we show extensions to the Linux kernel that enable it to operate seamlessly on top of unreliable memory and provide a user-space interface for memory allocation from the unreliable portion of the physical memory. Having this framework in place allowed us to identify what we call the refresh-by-access property of applications that use dynamic random-access memory (DRAM). We use this property to implement techniques for task-based applications that minimise the probability of errors when using unreliable memory enabling increased quality and power efficiency when using unreliable DRAM.

Related publications

[1] Charalampos Chalios, Dimitrios S. Nikolopoulos, Sandra Catalán, and Enrique S. Quintana-Ortí. Evaluating fault tolerance on asymmetric multicore systems-on-chip using iso-metrics. *IET Computers & Digital Techniques*, 10(2):85–92, 2016.

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

[3] Philipp Gschwandtner, Charalampos Chalios, Dimitrios S. Nikolopoulos, Hans Vandierendonck, and Thomas Fahringer. On the potential of significance-driven execution for energy-aware HPC. *Computer Science - R&D*, 30(2):197–206, 2015.

[4] 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 Napoli et al. [5], pages 9:1–9:8.

[5] Claudia Di Napoli, Valentina Salapura, Hubertus Franke, and Rui Hou, editors. *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*. ACM, 2015.

[6] José Ignacio Aliaga, Sandra Catalán, Charalampos Chalios, Dimitrios S. Nikolopoulos, and Enrique S. Quintana-Ortí. Performance and fault tolerance of

preconditioned iterative solvers on low-power ARM architectures. In Joubert et al. [7], pages 711–720.

[7] Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans J. Peters, and Mark Sawyer, editors. *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, volume 27 of *Advances in Parallel Computing*. IOS Press, 2016.

[8] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos 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 Cohen and Grove [9], pages 275–276.

[9] Albert Cohen and David Grove, editors. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. ACM, 2015.

[10] Charalampos Chalios, Dimitrios S. Nikolopoulos, and Enrique S. Quintana-Ortí. Evaluating asymmetric multicore systems-on-chip using iso-metrics. *CoRR*, abs/1503.08104, 2015.

[11] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos 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. *CoRR*, abs/1412.5150, 2014.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

The unprecedented increase in computing capabilities that we experienced in the past decades was founded in mainly two principles. First, the number of transistors that we can fit into an integrated circuit doubles approximately every two years. This property, which is commonly known as the "Moore's Law", stems from the ability of material scientists to halve the size of metal-oxide semiconductor field-effect transistor (MOSFET) transistors, used for manufacturing central processing units (CPUs), every two years. At the same time, Dennard scaling indicates that, as transistors get smaller, their power consumption in proportion to their area stays constant (i.e., approximately every two years, we pack twice the number of transistors in the same chip, consuming the same amount of power as before).

These trends have translated into significant performance improvements with every new generation of processors, since hardware architects have scaled the CPU clock without significantly increasing the power consumption of the chip. The CPU frequency increase along with the architectural optimisations are made possible by the abundance of transistors in the chip, boosting applications without the need for major software optimisations. However, in the middle of the last decade, we experienced the end of the Dennard scaling era. In small manufacturing processes, the leakage current, which could previously be ignored, now poses significant challenges to hardware designers.

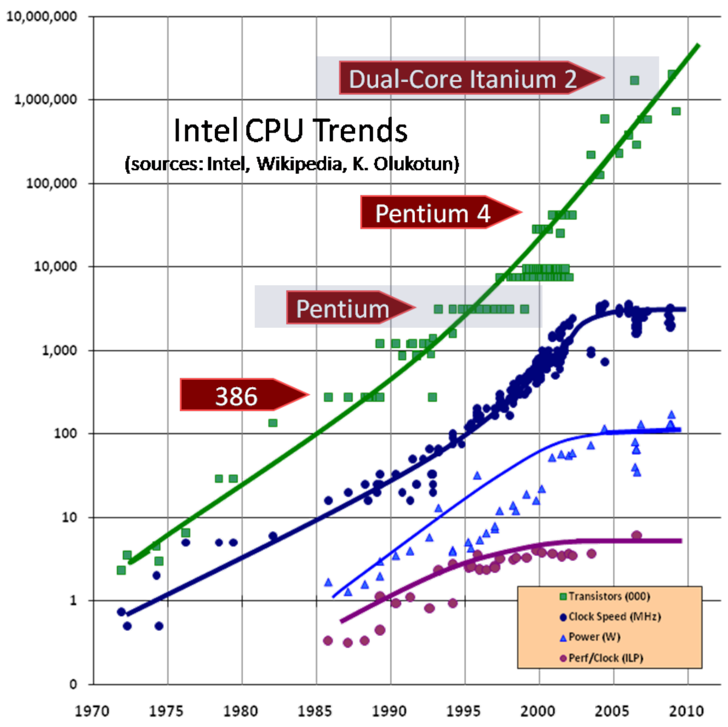


Fig. 1.1 Intel CPU introductions, clock speed and power consumption [83].

Leakage current consumes non-negligible static power and contributes significantly to the heating of the chip, which further burdens the power budget of the processor. To restrict static power consumption due to leakage, designers kept the supply voltage of the chip high, and at the same time, the switching frequency stopped following the increasing trends of previous generations. Thus, while our needs for computing and memory capacity continue to increase, our ability to build systems that cover those needs is hindered by the unsustainable power consumption these devices would require, a problem that was named the ‘power-wall’. Consequently, the power-wall significantly hurt the performance of the processors, which already suffer from the saturation of the instruction-level parallelism (ILP). The free lunch was over.

This trend is depicted in Figure 1.1. Although the number of transistors in the chip keeps increasing as predicted by Moore’s law, the increase of both the ILP and frequency have halted. Consequently, architects started using the extra transistors to fit more processing elements (cores) into one chip, causing a revolution in the computer science world. People from the software layers, systems software, and applications needed to put effort to take advantage of more cores found in the processors to make up for the performance loss due to the CPU clocks not scaling as they did previously. Thus, new programming models and advanced operating and runtime systems were created in the software labs of academia and industry.

However, the electrical power that is required to operate all the cores of the CPU concurrently can exceed the available power budget of the chip. This means that only some of the available cores can operate simultaneously. This phenomenon is called 'dark silicon'. Part of the cores (silicon) are powered off during operation to not exceed the available power budget. This new reality makes power consumption a problem that we cannot expect the semiconductor technology to solve for us in the near future. Instead, both academia and industry focus on holistic approaches where power consumption is a first-order concern for everyone from the hardware architect and software developer to the system administrator.

According to Dennard scaling, reduction of supply voltage has been the most useful tool. Hardware designers must reduce the power consumption of complementary metal-oxide semiconductor (CMOS)-based chips. Lately, hardware vendors have allowed software mechanisms to control CPU voltage and frequency. These mechanisms were used in the core of several tools that tried to take advantage of opportunities for voltage scaling in situations where the CPU is idle. Several researchers [19, 54] have pointed out that the most energy-efficient voltage region of operation for CMOS transistors lies in the *sub-threshold voltage* region. However, operating in the sub-threshold region comes with significant performance reduction due to aggressive scaling of the CPU frequency. Thus, it is deemed more beneficial to operate in the *near-threshold voltage* (NTV) region, sacrificing a small amount of energy efficiency to maintain some of the performance. This system design is called *near-threshold voltage computing* (NTVC). Even in the NTV region, CMOS operation poses significant challenges related to system reliability due to the increased process variation. Consequently, a lot of research focuses on tackling the reliability challenges connected with NTVC.

Problems with reliability do not appear only in NTV operation. State-of-the-art systems consume a considerable amount of power to ensure correct operation of logic and memory circuits. Manufacturers apply pessimistic guard bands to circuits to ensure correct operations. This pessimistic approach is exacerbated due to process variation, which has worsened in current technologies. Process variation (i.e., the variation of the transistor characteristics during the manufacturing process) creates a spread in the optimal operating point (e.g., supply voltage) across transistors in the chip. This means that, to avoid incorrect operation of some of the transistors, we need to select a supply voltage higher than required for most transistors. In other words, the requirement for reliability (high voltage margins) clashes with the significant objective of power efficiency.

Similarly, dynamic random access memory (DRAM) specifications define the average refreshing period to be uniform for all DRAM cells, regardless their retention time (i.e., the time after which the cell will lose the information it holds if it does not get refreshed). For example, the double data rate fourth generation (DDR4) specification defines that the whole memory should refresh every 64 ms, whereas studies have shown that, for many DRAM devices, we observe a worst-case scenario retention time of approximately 1 s. Therefore, DRAM devices pessimistically consume an amount of power that, in future DRAM generations, is projected to be a significant proportion of the total DRAM power consumption.

The requirements for high reliability and accuracy also affect performance. Moreover, CPU overclocking has been known to increase the probability of timing failures within paths of the processor. Supply voltage determines the length of the critical path of the circuit. The critical path, in turn, determines the maximum frequency that we can use without timing violations. Similarly, refresh operations in DRAM interfere with normal memory accesses. When a refresh operation is on-the-fly in a bank of the DRAM chip, this bank is unavailable for loads and stores, which decreases the effective bandwidth of the main memory. There is an obvious trade-off between reliability and power consumption and performance.

The struggle between reliability against power consumption and performance is not restricted to the hardware domain. Software developers often must compromise power and/or performance to ensure sufficiently accurate computations. For example, using higher precision floating-point data types (e.g., `double` vs `float`) increases the execution time and the amount of memory we need to store the representation of our data.

Recently, researchers have proposed relaxing the requirements for correctness and accuracy of computing systems, when this is acceptable, as a step towards overcoming the power-wall problem. Many applications can tolerate reduced accuracy in their output, while studies also show that it is possible to handle errors at the hardware level within the software stack. Thus, these observations triggered a great amount of research related to how we can systematically harness approximation to save other precious resources (e.g., power consumption and CPU cycles) and opened the way to a new computing paradigm, called approximate computing.

1.2 Approximate Computing

Approximate computing is a computing paradigm that allows an acceptable amount of inaccuracy in the result of a computation in exchange for reduction in the resources required for that computation. What makes approximate computing useful is the existence of applications that can afford bounded levels of inaccuracy in their results (e.g., the human eye can perceive up to a certain frame rate while watching a video). As a result, a video decoder can drop frames, as long as it ensures this critical frame rate is achieved.

Approximation can be realised both at the software and hardware level. Software schemes substitute a code block with an alternative implementation that produces the same result semantically, only with reduced accuracy. However, the approximate version of the computation consumes less resources in terms of power consumption or CPU time. For example, the loop perforation technique skips some of the iterations of a loop when this does not affect the quality significantly. Similar algorithmic approaches are followed by hardware designers. Alternative hardware implementations of an algorithm trade off accuracy for power consumption, execution time, or area. However, hardware approximation can be also implemented in general purpose CPUs as well as memory systems, such as DRAM or static random access memory (SRAM).

To cope with technological limitations, hardware designers enforce strict guard bands in the operating points of chips to minimise the failure probability of computing systems. Approximate computing allows chips to be configured outside the nominal operating points (e.g., CPU is undervolted or overclocked, an SRAM cache undervolted, and a DRAM chip refreshed less frequently). The consequence is increased failure probability, meaning that the probability of an error happening during the lifetime of an application is a possibility that can no longer be neglected. The responsibility of dealing with the errors that will occur due to operating the hardware in unsafe configurations is left to the software layers.

Since the burden of dealing with errors shifts towards the software stack, a significant amount of research has been carried out for studying application-specific methods to handle errors or algorithmic-based fault tolerance (ABFT). The ABFT approaches are methods specific to each algorithm that ensure the correct (or approximate) termination of the algorithm, even in the presence of errors or inaccuracies. Despite the strength of ABFT methods for dealing with errors, for approximate computing to become a prominent computing paradigm, there must be more general and systematic ways to handle errors. Towards this end, researchers have made efforts towards designing system-wide solutions for fault tolerance.

1.3 Contribution - Synopsis

The rest of the thesis is organised as follows. In Chapter 2, we discuss the background of approximate computing. We present the fundamental problem approximate computing attempts to solve and the different paths and research related to this field. Furthermore, we show how the work presented in this thesis deviates from this research.

In Chapter 3, we present an experimental evaluation of fault-tolerance characteristics of a popular iterative solver, Jacobi. We present a first case study of the energy efficiency of Jacobi when executed on an NTV processor. For our evaluation of NTVC, we account for the consequences of unreliability in the execution time of Jacobi. We show that it is possible to attain energy savings between 35% and 67% compared to a system operating at nominal voltage ranges. Next, we present a more systematic approach to evaluate the energy efficiency of an NTV system, using two additional iterative solvers, conjugate gradient (CG) and the generalised minimal residual method (GMRES). We use the concept of isometrics to show that it is theoretically possible to build a system based on NTV components that is more energy efficient than a state-of-the-art high-end Xeon processor. In our evaluation, we consider the performance penalty of unreliability. We show that a system using unreliable NTV components can provide better energy to solution (ETS) for performance penalty, due to unreliability between 7% and 129% for a fault tolerant version of the CG algorithm and between 178% and 359% of the GMRES algorithm, compared to execution with an off-the-shelf server at nominal operating points.

In Chapter 4, we show a programming model based on OpenMP 4.0 tasks that allows the programmer to express approximation in task-based parallel applications and the detailed implementation of a runtime system that implements the semantics of the programming model. We introduce the concept of task significance, which describes the contribution of a task to the quality of the result of the algorithm. Significance describes the semantics of approximation systematically and allows the runtime system to optimise the execution by enabling acceptable, according to the programmer, trade-offs between accuracy and energy efficiency. Moreover, we present a framework that optimises approximate execution based on application profiles and a runtime-defined energy budget. We compare our framework to *loop perforation*, a well-known approximation technique, for many selected benchmarks. We show that our framework can achieve similar energy savings to loop perforation. However, we also achieve a graceful degradation of the quality, depending on the enforced level of approximation, a feature that loop perforation cannot achieve due to the lack of information regarding the approximation characteristics of the application.

In Chapter 5, we study the potential for energy savings by relaxing the requirements for refreshing the DRAM memory. Requirements for refreshing the DRAM of devices are often pessimistic and are predicted to be a roadblock for the next generation, higher density devices. We design and implement a system architecture that can reliably operate on top of DRAM devices that are refreshed less frequently than nominal or are even not refreshed at all. From the point of view of software, we identify a novel property, *refresh-by-access* that allows us to minimise the errors occurring during the lifetime of an application because of the relaxed refresh rate. We design a scheduling policy for task-based applications on top of the *refresh-by-access* property that manages to reduce the number of DRAM errors observed by applications up to an order of magnitude. Our policy enables additional energy savings since it doubles the amount of data we can store unreliably for a given level or user-defined quality of output.

Finally, Chapter 6 concludes the thesis. We discuss the potential and prerequisites of approximate computing to emerge as a useful computing paradigm. We also discuss potential future work.

Chapter 2

Background

2.1 Near Threshold Voltage Computing

Traditionally, voltage scaling has been the most prominent means of reducing the power consumption of CPUs. Dennard scaling allowed manufacturers to reduce the supply voltage of chips and provide abundance of computational power, remaining within the available power budget. Moreover, when the power-wall started becoming a visible obstacle, vendors implemented mechanisms for dynamic voltage and frequency scaling (DVFS). These mechanisms were the building blocks of several techniques and tools for system administrators and software developers to reduce the power consumption of their systems without sacrificing significant performance [16, 29, 43, 50, 62, 69, 92, 93].

The CMOS transistors can operate, even using supply voltage V_{dd} below the threshold voltage V_{th}, with a theoretical minimum being determined as early as 1972 [84]. However, reducing V_{dd} does not come without a penalty. To avoid timing errors within the CMOS circuit, the operating frequency needs to scale proportionally to V_{dd}, which diminishes performance. However, because power consumption depends quadratically on voltage, scaling down V_{dd} while in the super-V_{th} region ($V_{dd} > V_{th}$) saves energy. When moving in the sub-threshold region ($V_{dd} < V_{th}$), leakage energy increases exponentially with V_{dd}. The leakage energy eventually dominates, and a sweet spot for energy consumption forms (Figure 2.1).

Scaling V_{dd} from the super-V_{th} region to the NTV region ($V_{dd} \approx V_{th}$) yields an energy reduction of $10\times$ with a performance penalty of the same order of magnitude.

In contrast, further scaling V_{dd} towards the sub-V_{th} region and the point of minimum energy consumption provides only an additional $2\times$ improvement in energy efficiency with a significant increase of performance degradation of an additional $50 - 100\times$. Thus, several researchers [19, 54] have proposed that operation in the NTV region is a good trade-off between energy efficiency of the CMOS chip and performance degradation.

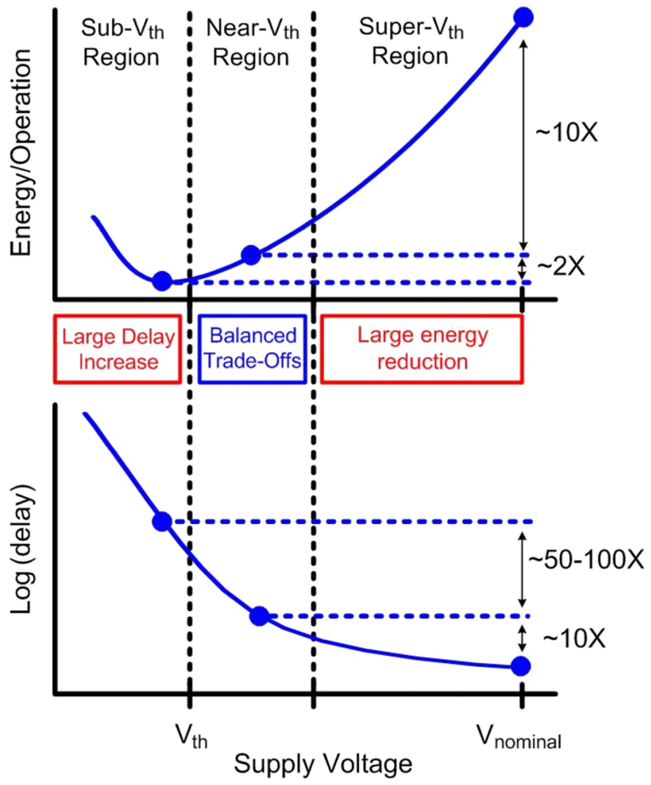


Fig. 2.1 Energy and delay in different supply voltage operating regions [19]

The requirement to scale the operating frequency of CMOS-based chips proportionally to the voltage scaling stems from the need to produce robust chips. Voltage scaling increases the delays of CMOS circuits, so without equivalent scaling of the frequency, timing errors will occur within the circuits, harming their correct operation. Moreover, decreased V_{dd} is responsible for increased failure probability of SRAM memory cells, a $5\times$ increase of bitcell failure rate when operating at near-V_{th} region according to [19]. To make things worse, reliability issues are further exacerbated due to increased process variation [19, 42, 64, 78] when operating at the sub-V_{th} region. Figure 2.2 shows a $\pm 2\times$ frequency variation of a chip operating when operating in the near-V_{th} region as opposed to a $\pm 18\%$ at the super-V_{th}

region. Moreover, chips operating in the near-V_{th} region are increasingly sensitive to temperature variations, which hinders their robustness.

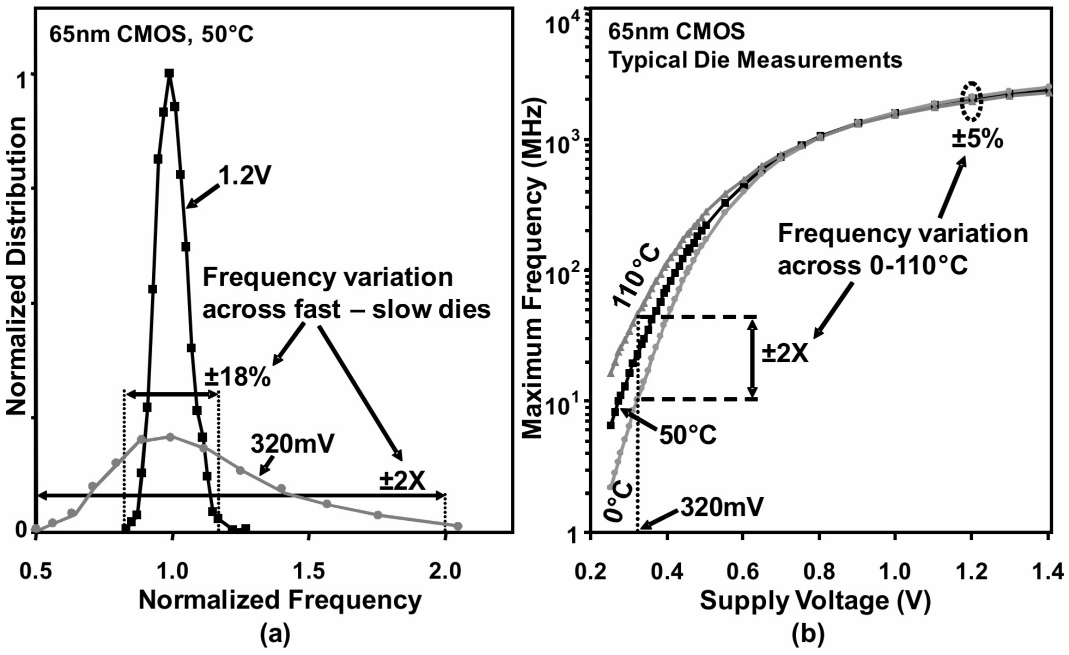


Fig. 2.2 Modeling and measurements of variation [42]

The attractiveness of NTVC has given rise to a significant amount of research to address the challenges that NTVC presents. [19, 54] have established that operating in NTV presents a nice trade-off between energy efficiency and performance loss. [54] proposed yielding some of the energy savings of the sub-V_{th} region to gain back some of the performance, and simultaneously, they utilised a new family of logic circuits to further increase energy savings. [19, 41, 42, 64] suggested that the lost performance due to frequency reduction in NTVC will be recovered by manycore architectures. They proposed using group silicon islands, and depending on the variation, they presented and applied different operating frequencies, instead of applying a worst-case scenario universal frequency across the die. [19, 40, 78] also tried to apply circuit-level optimisations to reduce the variability for chips operating in the near-V_{th} region, while [12] examined various SRAM designs considering the process variation under NTVC to increase the yield of SRAM-based chips. The common ground of this research line is that they tried to optimise chip designs for NTVC to cope with the parametric variations and avoid errors.

On the other hand, over the past couple of decades, there have been researchers that have attempted to understand how errors at the circuits reflect at the microarchitectural level. For example, [72] studied the effect of errors from the perspective of individual hardware units within microprocessors, nonetheless, without examining their consequences for the software. Moreover, [15, 45] proposed allowing errors to occur in selected hardware units instead of struggling to harden microprocessors against errors, using pessimistic, worst-case scenario guard bands and fault-tolerance mechanisms within the microarchitecture. Their proposal is based on the observation that there are applications that can tolerate these errors without compromising much of their output quality. They focus on the so-called recognition, mining, and synthesis (RMS) applications that present algorithmic and cognitive resilience. Algorithmic resilience stems from the fact that imprecision in their computations does not significantly affect the result. Cognitive resilience relates to the fact that imprecision at the algorithm's result is acceptable as long as this is acceptable for human users.

Therefore, we are presented a novel and interesting trade-off, one of reliability versus energy efficiency and performance. To build robust hardware, we need to forsake potential for additional energy savings and performance improvements. For example, we retain the V_{dd} and high frequency of chips to avoid issues with process variation and timing errors. Alternatively, we could try to design systems that relax the requirement for reliability in a controlled way to allow energy and performance optimisation opportunities.

In this thesis, we propose a methodology based on isometrics for evaluating the viability of NTVC as an efficient computing paradigm. We assume an architecture that exposes the software layer to faults occurring at the microarchitecture level and consider the effect of these errors on the execution time of the applications running on the unreliable platform. In our study, we focus on iterative solvers in which errors might slow convergence, but our methodology can be extended to applications where the overhead of unreliability can occur from other sources as well (e.g., checkpointing and restarting).

We introduce isometrics[82] as a framework for comparing two platforms regarding their energy efficiency. We use a low-power ARM Cortex-A7 processor as an emulation of an NTV processor. Cortex-A7 features very low power consumption and a very simple in-order pipeline. This design is in line with the description of an NTVC single core in the literature [40], where advanced microarchitecture features are sacrificed for simplicity and stability against process variation. We assume a manycore architecture built of many such cores and compare it against a more powerful ARM Cortex-A15 processor and a high-end Xeon processor. We extend the concept of isometrics to account for the performance overhead that is a consequence for using unreliability.

2.2 Approximate Computing

Approximate computing is the system design and programming paradigm that can perform a computation in an imprecise way, if the quality of the result of the computation is acceptable by the final user of the system or the application. The approximation can originate at the hardware level, where some of the hardware units operate in an unreliable way, or at the software level where programmers define parts of the application that will use a less accurate algorithm to perform a task.

Approximation is inherent in many applications (e.g., probabilistic algorithms or algorithms that produce a result in which the quality depends on the perception of the user of the algorithm). For example, video decoding applications are required to produce a fixed number of frames per second depending on the quality requirements of the decoded video. Even if the system can provide enough throughput to produce a higher frame rate (better quality) it is acceptable that the decoder does not compute the extra frames to save resources (e.g., power consumption in embedded devices). Moreover, there are algorithms that present fault-tolerance properties that allow them to produce acceptable results even in the presence of errors. For example, iterative solvers repeatedly improve the accuracy of the output across iterations; thus, the effect of an error can be mitigated by subsequent iterations of the algorithm.

These observations have led to the advent of research that studies fault-tolerant and approximation properties of algorithms that enable the use of approximate frameworks. These frameworks are based either in the relaxed reliability of the hardware or the explicit substitution of certain computations with equivalent code blocks that perform the same task semantically, only with reduced accuracy. In both cases, the goal is to reduce resource utilisation, that being power consumption or

CPU time. As a result, research regarding approximate computing can be separated into two categories:

1. Application-based fault tolerance that studies the effects of errors on applications and methods to minimise those and

2. Approximate computing frameworks that endeavours to create programming models that allow programmers to develop approximate applications and runtime systems that manage the approximation-related information provided by the programmers to optimise execution.

2.2.1 Application-based Fault Tolerance

Application fault tolerance has been of interest for researchers and engineers for decades. Computing systems of any scale, from embedded devices to the highest ranked supercomputers, presents hardware failures despite the efforts of system designers to protect them from those using advanced error protection mechanisms within the hardware (e.g., error detection and correction codes, high guard bands in voltage and frequency, high refresh rate in DRAM memories, etc.). Thus, researchers of software-level error mitigation techniques have proposed methods that primarily rely on periodically checkpointing the state of applications and restarting from the latest saved state when an error has been detected [30]. Other methods [21] replicate the state of the algorithm to be able to detect errors, such as silent data corruption. [53] proposed more advanced replication schemes that enable both detection and recovery from such errors.

Apart from system-level methods for software resilience, there are solutions [1, 13, 20, 28, 57, 61, 76] that investigate algorithmic properties that we can presume upon to increase the resilience of the algorithm. This body of work can be exploited to enable the adoption of a system design paradigm that allows increased error probability in selected hardware components. For example, [28, 76] made use of certain algorithmic properties of two iterative solvers, GMRES and CG, to enable selective reliability. The algorithms are split in two parts, one that is allowed to run on hardware with relaxed reliability and a second one that always runs reliably. Even if an error occurs during the execution of the former, it is guaranteed that, once the latter part is executed reliably, it will bring the algorithm back on track for convergence. However important, this method makes no assumption regarding the effect of the error on the convergence rate of the algorithms (i.e., the algorithm will converge, albeit, it might take a significantly increased number of iterations to do so, negating any energy gains due to operating in NTVC.

The ABFT presents solutions that are inherently application-specific and applied in an ad-hoc manner. For approximate computing to become a useful programming paradigm, there is the need for a systematic method of expressing possibilities for approximation during the implementation of an algorithm.

In this thesis, we design a programming model and runtime system that provides the required abstractions for the programmer to express approximation semantics for the developed algorithm. We opt for an OpenMP-like task parallel programming model because: a) parallelism is inevitable when performance is required. The NTVC system designs suggest that performance loss due to reduced frequency will be regained by massively parallel systems, and b) tasks present an intuitive abstraction level for describing approximation. A task is a well-defined block of computation to which the programmer can attribute approximation properties, such as *this task is a good/bad candidate for approximation*. In this way, we can decouple *what* is a candidate for the approximation from *how* it will be approximated.

We introduce the property of *task significance*, which, in our context, describes how much a task contributes to the quality of the results. Consequently, a significant task should always run accurately, whereas a non-significant or less-significant task is a candidate for approximation. Finally, the level of approximation is defined by user-defined constraints, which include the level of accuracy required from the computation and the available energy budget. We call this programming paradigm *significance-driven computing*.

2.2.2 Parallel Approximation Frameworks

Quickstep [56] is a tool that approximately parallelises sequential programs. The parallelised programs are subjected to statistical accuracy tests for correctness. Quickstep tolerates races that occur after removing synchronisation operations that would otherwise be necessary to preserve the semantics of the sequential program. Quickstep thus exposes additional parallelisation and optimisation opportunities via approximating the data and control dependencies in a program. On the other hand, Quickstep does not enable algorithmic and application-specific approximation, which is the focus of our work, and does not include energy-aware optimisations in the runtime system.

Variability-aware OpenMP [68] and variation tolerant OpenMP [67] 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 that 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 floating-point unit (FPU) operations, which execute on specialised FPUs with configurable accuracy. Our framework applies 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 does not require specialised hardware support and runs on commodity systems.

2.2.3 Other Approximation Frameworks

Several frameworks for approximate computing discard parts of code at runtime, while asserting that the quality of the result complies with the quality criteria provided by the programmer. Green [4] is an application programming interface (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 quality of service models for the outputs of the approximated blocks of code as well as recalibration functions for correcting unacceptable errors that may be incurred due to approximation. Sloan et al. [81] provided guidelines for manual control of approximate computation and error checking in software. These frameworks delegate the control of approximate code execution to the programmer. 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 [79] is a compiler technique that classifies loop iterations as critical and non-critical. The latter can be dropped, as long as the results of the loop are acceptable from a quality standpoint. Input sampling and code versioning [97] also use the compiler to selectively discard inputs to functions and substitute accurate function implementations with approximate ones. Similar 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 optimisations 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 for energy saving and performance gain. While these approaches demonstrate aggressive performance optimisation due to approximation, they do not consider parallelism in execution. Furthermore, these techniques operate at a granularity different from parallel tasks or specific runtime energy optimisation opportunities, which are exposed through approximation.

Several software and hardware schemes for approximate computing follow a domain-specific approach. ApproxIt [96] 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. Gschwandtner et al. used a similar iterative approach to execute error-tolerant solvers on processors that operate with NTVC and reduce energy consumption by replacing cores operating at nominal voltage with NTVC cores [24]. Schmoll et al. [77] presented 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 approximate computing framework, we provide sufficient abstractions for implementing the application-specific approximation methods.

Other tools automate the generation and execution of approximate computations. SAGE [74] is a compiler and runtime environment for automatic generation of approximate kernels in machine learning and image processing applications. Paraprox [73] implements transparent approximation for data-parallel programs by recognising common algorithmic kernels and then replacing them with approximate equivalents. Moreover, ASAC [70] provides a sensitivity analysis for automatically generated code annotations that quantify significance. We do not explore automatic generation of approximate code in this work. However, our techniques for quality-aware, selective execution of approximate code are directly applicable to scenarios in which the approximate code is derived from a compiler, instead of source code annotations.

2.3 DRAM Reliability and Fault Tolerance

Figure 2.3 shows the organisation of a DRAM-based memory subsystem. In modern architectures, there is one on-chip DRAM controller per CPU, attached to one or more memory channels, which can operate independently to maintain high bandwidth utilisation. One or more dual inline memory modules (DIMMs), which consist of one or more ranks, can be connected to each channel. Each rank is a set of DRAM chips that operate in unison to complete one memory operation. Note that each DRAM chip consists of one or more banks, each of which has a dedicated control logic, so that multiple banks can process memory requests in parallel. Finally, banks are organised in two-dimensional (2D) arrays of rows and columns of DRAM cells, which are composed of one access transistor and a capacitor that holds the information in the form of an electric charge.

Due to the ‘dynamic’DRAM nature, the charge of the capacitor of the DRAM cell leaks gradually and may lead to a complete loss of the stored information. The time interval during which the information stored in a DRAM cell can be retrieved correctly is called the *retention-time* T_{RET} of the cell. To maintain the integrity of the data stored in the cell, the charge on each capacitor must be periodically restored.

Auto-refresh: Modern memory chips employ the so-called auto-refresh (AF) operation, where the memory controller periodically issues a refresh command, during which the DRAM device refreshes one or more rows per bank depending on the DRAM size. During AF, a DRAM bank cannot serve any request and all open rows in a rank remain closed. This causes performance loss and consumes significant power since every opening of a row is expensive. According to the DDR3

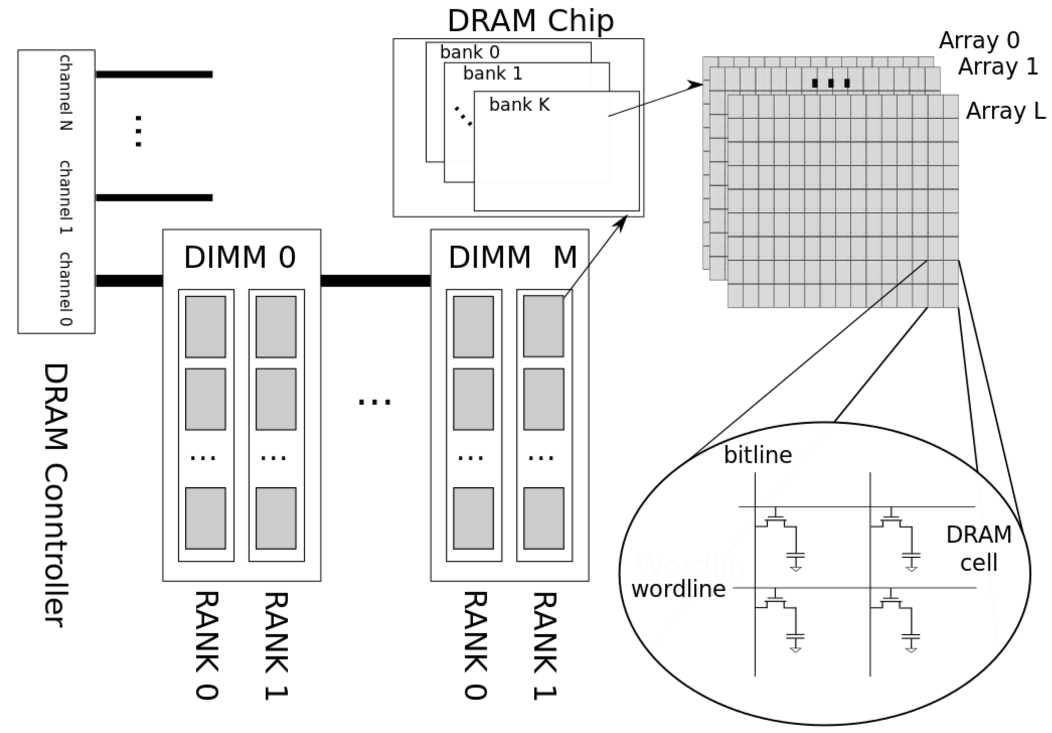
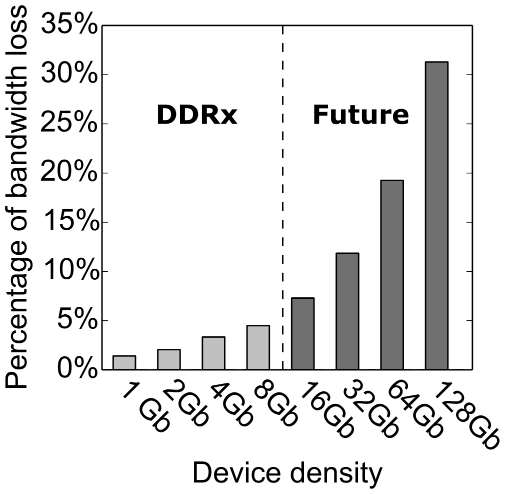
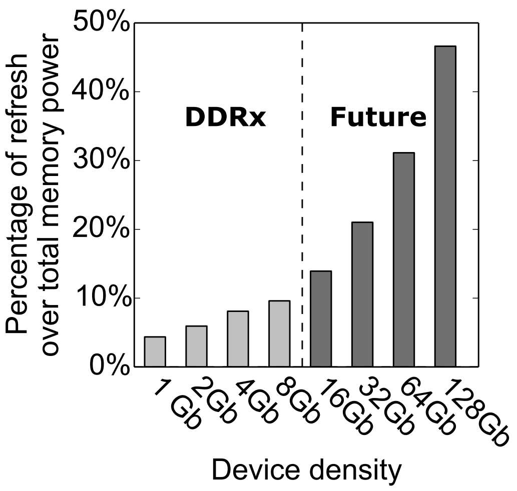


Fig. 2.3 DRAM Organisation



(a) Throughput loss



(b) Power overhead

Fig. 2.4 Power and performance overheads on current and future DRAM technologies [48]

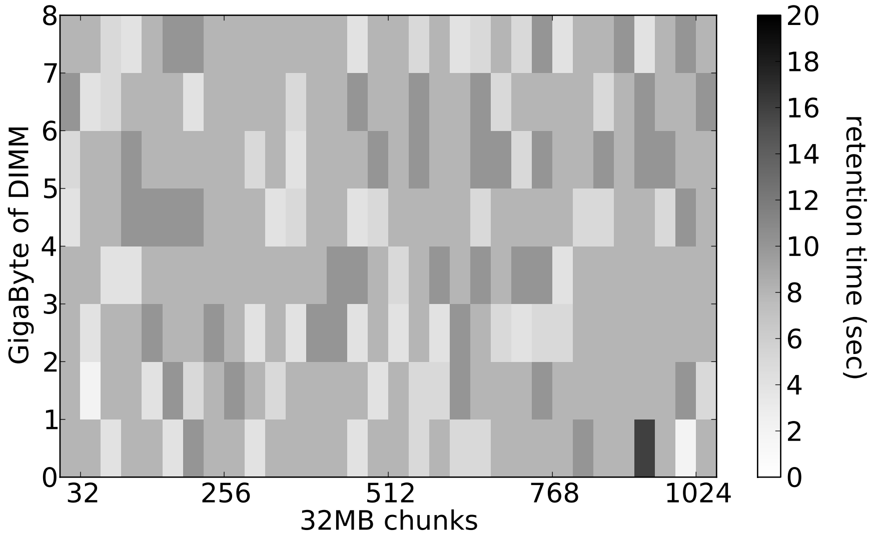
specification [33, 34] every row of the DRAM should be refreshed at least once in a period $T_{REFW} = 64$ *ms* and the DRAM controller needs to send, on average, one refresh command every $T_{REFI} = 7.8$ *us*, which means that in T_{REFW}, 8192 refresh commands must be issued. As the density of DRAM devices increases, the number of rows that need to be refreshed with every refresh command grows. Therefore, T_{REFC}

needs to increase proportionally. Consequently, the power consumption and the throughput loss incurred by the refresh operation is expected to increase considerably in future DRAM generations, as depicted in Figure 2.4.

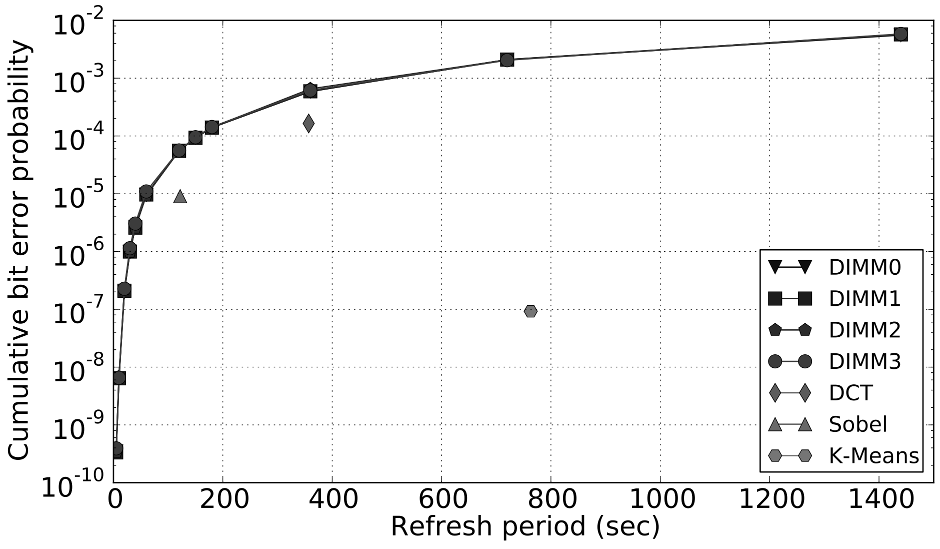
2.3.1 Pessimism of Auto-Refresh Operation

Recent studies on custom field-programmable gate array (FPGA) boards have shown that the cell retention time varies considerably across and within a DRAM chip. Typically, only a very small number of cells needs to be refreshed once every $T_{REFW} = 64\ ms$ [8, 37, 44, 47]. To verify this observation on typical server environments, we have performed experiments on various 8 GB DDR3 DIMMs of a premium vendor, where we used our experimental prototype and a memory tester with random and uniformly-distributed data designed to detect the retention time of DRAM bit cells. The details of our memory tester will be discussed in a following section. Figure 2.5a shows the spatial distribution of the retention time for one of the DIMMs, where it is evident that around 80% of the cells have a retention time of 5 s or more, far higher than the 64 ms minimum assumed by the DDRx specification. Going one step further, Figure 2.5b plots the cumulative distribution function (CDF) of bit errors on all 8-GB DIMMs of our system by aggressively relaxing the refresh period from the conservative 64 ms to 5 s and up to 60 s. Interestingly, even if, in our experiments, the refresh period is relaxed from 78$\times$ (5 s) to 937$\times$ (60 s), the cumulative bit error rate (BER) stays low, ranging from less than 10^{-9} to about 10^{-5}. Selecting the appropriate refresh period depends on the number of errors that is acceptable for applications running on the system. Furthermore, the system needs to guarantee that those errors do not affect any critical system data and lead to catastrophic failures, an issue that we address in this work.

Commercial DRAMs target a BER ranging from 10^{-12} to 10^{-9} when operating under conservative refresh [60]. Even under this common scenario, our results indicate that, within our server, the refresh rate can be relaxed at least by 78$\times$ (5 s) if we are prepared to adopt a BER of 10^{-9}. This indicates that insisting on a fixed refresh rate of 64 ms is extremely pessimistic and wastes power and throughput. Accepting a BER of 10^{-9} or even lower may allow aggressive relaxation of the refresh-rate, but this highly depends on the effect that these errors have on the correctness and output quality of applications.



(a) Spatial distribution of worst-case retention time



(b) Cumulative Distribution Function of bit-errors per DIMM with variable refresh intervals

Fig. 2.5 Retention Time Characterisation on a Commodity Server with Server-Grade DRAMs

2.3.2 Related Work on Relaxing the Refresh-Rate and Open Issues

Recent schemes have tried to exploit the non-uniform retention time of DRAM cells for relaxing the refresh rate, either by: a) targeting a very low memory BER by selectively applying retention-aware refresh on cells performing poorly or well, or b) by allowing a higher memory BER and permitting errors to be masked by the resilient data portions of the application.

Multi-rate Refresh: Techniques that preserve low BERs [9, 17, 36, 48, 58, 59, 91], which we refer to as *multi-rate refresh*, group rows into different bins based on an initial retention time profiling. This approach is similar to the one we apply in our server to select a higher refresh rate for rows belonging to the lower retention bin. However, such approaches are highly intrusive since they assume fine-grain control of the refresh rate (i.e., at the level of the row). Such schemes have been abandoned in modern DDRx technologies due their high cost. Furthermore, such schemes are deemed impractical since they neglect the fact that the retention time of each cell depends on the stored data and can change at runtime [26, 65]. Therefore, their essential profiling step cannot be accurate, and some cells may be refreshed at an inadequate refresh rate. In practice, this will lead to errors, which may affect the system data and result in catastrophic system failures. One can try to address these errors and minimise their effect on the system or the output-quality through error-correcting codes [46, 65], but this will essentially limit the gains realised out of the relaxed refresh.

Error-resilient Techniques: Another viable solution for addressing potential errors is to allow them and mitigate them at the application layer by exploiting inherent application-resilience properties [14, 75]. Recent studies have revealed error-resilient properties of numerous applications, including the processing of a probabilistic data, iterative structure, which allows averaging of the effect of any error, or user tolerance to small deviations in output quality. A very limited number of works have utilised such a paradigm in DRAMs [49, 66]. These researchers exclusively employ critical-aware data allocation (CADA) on different memory domains.

Flikker [49] is the most representative case of CADA in the literature. The Flikker framework assumes the separation of the memory into two domains, the reliability of which is controlled by the refresh rate. The CADA techniques ensure that critical portions of program data are stored on a reliable domain (using the conservative refresh rate), while the rest are stored on a less-reliable domain (with a relaxed refresh rate). Although such an approach may have offered an interesting programming model for exploiting application resilience and may help users accept a lower memory BER and lower the refresh rate compared to the multi-rate techniques, it has the following key limitations:

- The options to describe the criticality of data are effectively limited to a binary criticality measure (critical or non-critical), resulting in high-energy consumption for the refreshing of data, which cannot afford the highest error rates that may occur in the system. Supporting intermediate categories of criticality may present a significant opportunity for further reduction of refresh operations.
- It does not provide other error mitigation mechanisms for keeping the effect of errors under acceptable levels apart from moving data to the reliable but highly power-inefficient domain. As such, the scope for errors on non-critical data is limited. This limits the aggressiveness of the refresh-rate reduction.
- It has been implemented and evaluated using simulation, thus ignoring the implications of the full system stack and the time-dependencies that exist between data accesses and cell retention. This has an effect on the efficacy of the method and on the assumed application-specific error-resilient properties.

However, the most important limitation of CADA and of multi-rate-refresh schemes is that they use only spatial techniques for applying relaxed refresh to a few retention bins or a specific memory domain. They ignore an important property of applications, which is an implicit ability to refresh data by accessing it. Temporal properties of memory accesses in applications can be exploited to achieve natural refresh and error resilience, which in turn enables more aggressive refresh-rate relaxation than CADA.

2.3.3 Refresh-by-access

Every access to memory naturally opens every row and consequently restores the stored charge in the capacitor of each DRAM cell, thus incurring an implicit refresh operation that can help relax the pessimistic rate assumed by the conservative auto-refresh operation or even skip it completely. We call this property *refresh-by-access*, and as we demonstrate in this thesis, its exploitation can yield more aggressive DRAM refresh relaxation than CADA for a wide range of applications. Essentially, such a property can help to significantly reduce the number of manifested errors under relaxed refresh rates and improve application resilience and system reliability. Studying such a property requires the use of real DRAMs and cannot take place on simulators since, to the best of our knowledge, there are no models that jointly capture the time-dependent relation between access and retention time. The efficacy of *refresh-by-access* depends on the temporal properties of the application-specific access patterns. To better understand the potential of the technique, we used on an off-the-shelf server and server-grade DRAMs, the details of which are discussed in a later section, and executed three different applications (discrete cosine transform (DCT), Sobel, and K-means). The results are shown in Figure 2.5b. We observe that the inherently frequent memory accesses of each application result in a much lower number of errors than the number assumed at each relaxed refresh point. The effect of this outcome is two-fold: a) it shows that, in practice, *refresh-by-access* helps limit the number of manifested errors that need to be tolerated by the application and b) that the behaviour of an application on a real system will be completely different that one observed on simulators, as the latter do not capture the effect of refresh-by access. Therefore, solutions based on CADA may demonstrate completely different efficacy than the one shown in current simulation-based evaluations. The question that remains open is how to intelligently exploit such a temporal property in isolation or in combination with other methods that exploit spatial properties of retention and how to properly evaluate and capture all the time-dependent interactions on a real and complete system stack.

Chapter 3

Use Cases for Approximate Computing

3.1 On the Potential of Significance-Driven Execution for Energy-Aware High-performance Computing

3.1.1 Introduction

The interest in energy consumption in high-performance computing (HPC) has increased over the past decade. While high performance is still the main objective, many considerations have been raised recently that require including power and energy consumption of hardware and software in the evaluation of new methods and technologies. The motivational reasons are diverse, ranging from infrastructural limits, such as the 20 MW power limit proposed by the US Department of Energy, to financial or environmental concerns [86].

Thus, the hardware industry has shifted focus to include power and energy minimisation into their designs. The results of these efforts are evident by features such as DVFS or power and clock gating. Moreover, DVFS has been an efficient tool to reduce power consumption; however, increased voltage margins –resulting from shrinking transistors– put a limit on voltage scaling. Although an energy-optimal voltage setting would often be below the nominal transistor threshold voltage, this would give rise to increased variation, timing errors, and performance degradation that would be unacceptable for HPC applications. An alternative approach is to operate hardware slightly above the threshold voltage (also called NTV). The NTV would achieve substantial gains in power consumption but with acceptable performance degradation, which is caused by a rather modest frequency reduction and can be compensated for by parallelism [40].

Methods for tolerating errors in hardware have been studied in the past [21, 30]. This resulted in several solutions at different levels of the design stack, in both hardware and software. However, these solutions often come with non-negligible performance and energy penalties. As an alternative, the shift to an approximate computing –also known as significance-based computing– paradigm has recently been proposed [5, 40, 45, 75]. Approximate computing tries to trade reliability for energy consumption. It allows components to operate in an unreliable state by aggressive voltage scaling, assuming that software can cope with the timing errors that will occur with higher probability. The objective is to reduce energy consumption using NTV and avoid the cost of fault-tolerant mechanisms.

In this work, we are trying to utilise the potential for power and energy reduction that NTV computing promises combined with significant-based computing. We investigate the effects of operating hardware outside its standard reliability specifications on iterative HPC codes, incurring both computational errors as well as reductions in energy consumption. The HPC domain is a suitable candidate for applying our techniques. There is an increasing need for computational power in the HPC world, which we cannot deliver at the moment mainly due to power constraints. Moreover, a significant number of HPC applications are build on top of iterative algorithms which in many cases are inherently tolerant to errors. Our techniques could naturally be applied to other domains with power constraints e.g., embedded systems, but in this thesis we focus on the HPC domain. We show that codes can be analysed in terms of their significance, describing their susceptibility to faults with respect to the convergence behaviour. Using the Jacobi method as an example, we show that there are iterative HPC codes that can naturally deal with many computational errors, at the cost of increased iterations to reach convergence. We also investigate scenarios where we distinguish between significant and insignificant parts of Jacobi and execute them selectively on reliable or unreliable hardware, respectively. We consider parts of the algorithm that are more resilient to errors to be insignificant, whereas parts in which errors increase the execution time substantially are marked significant. This distinction helps us to minimise the performance overhead due to errors and utilise NTV optimally.

We show that, in our platform, we can achieve 65% energy gains for a parallel version of Jacobi running at NTV compared to a serial version at super-threshold voltage along with time savings of 43%, when we execute with 20% of the super-threshold frequency.

The notion of significance is introduced in Section 3.1.2. We will describe our methodology and experiment setup in Section 3.1.3 and analyse and illustrate our results in Section 3.1.4. Finally, Section 3.1.5 will conclude and provide an outlook for future research.

3.1.2 Significance

We propose the notion of code *significance* (i.e., that different parts of an application show different susceptibility to errors in terms of the change in the result). This applies to data as well as operations and gives rise to considering partial protection methods, employed only where and when necessary. This distinction, coupled with the prospect of NTVC, creates the opportunity for significant amounts of power savings by running non-significant parts of the computation on unreliable hardware.

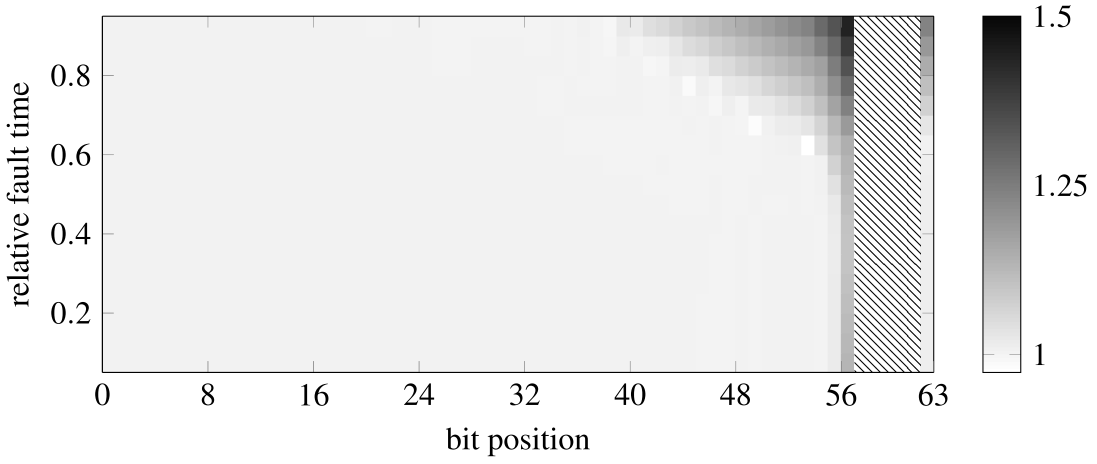


Fig. 3.1 Relative time overhead of Jacobi for faults in A at various iterations, averaged over all matrix positions, for all bit positions ($N = 1000$). The hatched bar denotes divergence.

We want to illustrate the applicability of significance classification on iterative solvers and their resilience in the presence of faults. Iterative solvers operate by repeatedly updating the solution of a system of equations until it reaches the required level of accuracy. Errors occurring in these algorithms can be gracefully mitigated at the cost of an increased number of iterations to reach convergence. Thus, these applications are suitable candidates for trading accuracy for lower energy consumption.

We selected the weighted Jacobi method as a representative use case to study the resilience to errors in the broader class of iterative numerical applications. Jacobi solves the system $Ax = b$ for a diagonally dominant matrix A. It starts with an initial approximation of the solution, x^0, and in each step updates the estimation for the solution, according to:

$$x_i^{(k+1)} = \omega \left(\frac{1}{a_{ii}} b_i - \sum_{j \neq i} a_{ij} x_j^k \right) + (1 - \omega) x_i^k. \tag{3.1}$$

The algorithm iterates until the convergence condition $||Ax - b|| \leq limit$ is satisfied and is guaranteed to converge if A is strictly diagonally dominant, that is:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{3.2}$$

To demonstrate the applicability of significance to Jacobi, Figure 3.1 presents the effect of a single bit flip fault happening in matrix A at various iterations of an otherwise fault-free Jacobi run. It shows the relative overhead of Jacobi (i.e. the number of additional iterations) required to reach the convergence limit compared to a fault-free run. Generally, Jacobi exhibits a logarithmic convergence rate and later iterations are more significant due to the overhead required to recover from a fault. Furthermore, because the achieved residual for later iterations is lower than for earlier iterations (due to the better x that has been computed), later iterations also show higher sensitivity to faults. Both these factors render late Jacobi iterations more significant than early iterations [20]. This motivates the potential application of protection or recovery mechanisms for later Jacobi iterations only.

In addition, Figure 3.1 illustrates that Jacobi can cope well with flips happening in lower bit positions, as they cause little to no overhead. This can be attributed to the high precision of double-precision floating-point numbers. Flips happening in the high bits of the exponent for elements of A, however, can have two possible outcomes, depending on their position and the error they introduce. If the flip causes a positive error in the floating-point number and happens aside the diagonal, there is a risk of violating Jacobi's convergence condition of strict diagonal dominance for A (Eq. (3.2)) (analogously for negative errors on the diagonal). These cases manifest themselves as the solid bar shape in Figure 3.1 for bits 57–62. For most cases that violate this condition, Jacobi does not converge and ends up with a residual of either *infinity* (*Inf*) or *not a number* (*NaN*), depending on the operations involved. Overall, Figure 3.1 shows that, for most bit positions, there is no protection or recovery necessary, except for a few high bits of the exponent that justify mitigation techniques.

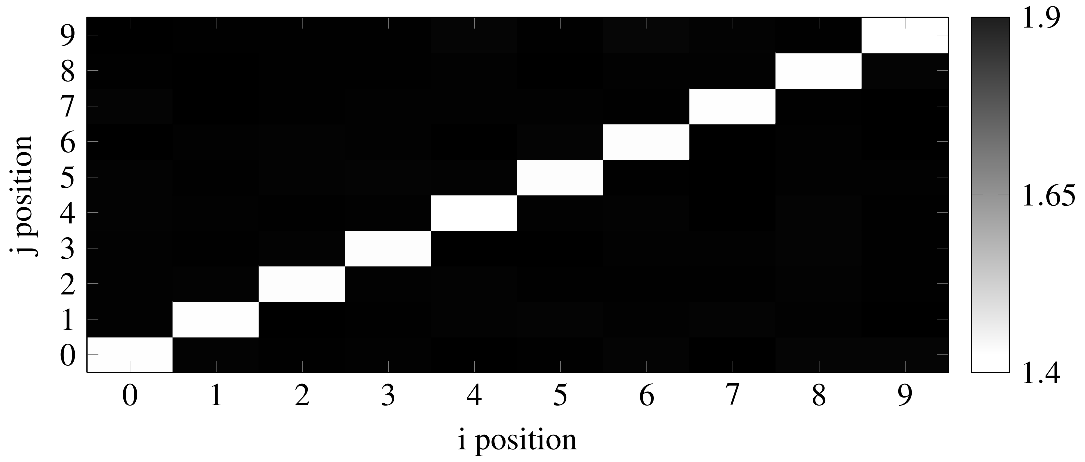


Fig. 3.2 Relative time overhead of Jacobi for faults in A at all matrix positions, averaged over all bit positions ($N = 10$).

In addition to Jacobi's varying significance depending on the progress of the algorithm, significance can also vary depending on the component that is exposed to a fault (A, b or x) as well as the position within a component. As an example, Figure 3.2 presents the relative overhead of injecting a fault on the diagonal and offside the diagonal of iteration matrix A (for illustrative clarity, we chose the problem size to be 10×10). The fact that elements on the diagonal have lower effects (and hence lower significance) can be attributed to two reasons. First, these elements are used in a division operation, whereas the others are used in a multiplication (see Eq. (3.1)). Second, we use a uniform distribution to randomly initialise the elements of A and b, that leads to most numbers being positive and greater than 1. Hence, a multiplication operation tends to increase any effect a fault might have, whereas a division generally reduces it. For this reason, our experiment results presented in Section 3.1.4 involving matrix A are based on sampling positions both on and aside the diagonal and computing a weighted average for the entire matrix. It should be noted that the weighting between diagonal vs aside elements naturally changes with the matrix dimensions. Consequently, the overall significance of A is also partially dependent on the input data size.

Detecting significance can be difficult for large applications. It is an algorithm-specific attribute; hence, input from programmers can be indispensable for a system that provides support for significance-based computing. Such a system would rely on the programmer's knowledge about the algorithm, who would denote the parts that can be executed unreliably. Large pieces of software are often composed from smaller mathematical kernels whose tolerance in faults has been extensively studied. Such studies could be used by programmers to mark code regions as significant or non-significant, without having to perform extensive profiling. Moreover, there is ongoing research for algorithmic detection of the significance of code based on profiling (e.g., automatic differentiation [89]). This approach studies the sensitivity of code blocks, monitoring the range of the output of a code block after perturbation applied to the input. A code block is then considered more sensitive to errors, when the range of possible output values is larger. Nevertheless, automatically and efficiently detecting code significance is still an open research area.

Furthermore, the design of a system for significance-based computing should provide contingencies for applications that cannot afford unreliability in their execution. For these applications to be able to benefit from NTVC, the system must employ software or hardware fault-recovery mechanisms.

3.1.3 Methodology

This section describes the fault model of our work. Moreover, it elaborates the power and energy effects that we expect from operating hardware unreliably and provides details about the hardware and our measurement methods.

Fault Model

Following common practice in related work [72], faults can be categorised as:

- No impact: The fault has no effect on the application.
- Data corruption:
    - Silent: only detectable with knowledge about the application,
    - Non-silent: detectable without knowledge about the application, and
    - Looping: faults that cause the application to loop.
- Other (e.g., illegal instructions or segmentation faults).

Of these fault classes, we consider silent data corruption (SDC) faults since they are the most insidious ones in high performance computing. Signalling errors, such as *Inf* or *NaN* or application crashes due to illegal instructions, are comparatively easily detectable. Additionally, looping could be identified by detecting fixed points in the iteration data of an application or constraints on the execution time of code regions. In contrast, SDCs can cause graceful exits with possibly wrong results, making them particularly important to handle.

Moreover, SDCs can be categorised further as persistent or non-persistent. Persistent faults occur at the source of the data in question (i.e., if the data is read multiple times, it will exhibit the same deviation each time). Non-persistent faults, on the other hand, are faults in temporary copies of data that are only used once (e.g., faults happening directly in execution units or registers).

We consider persistent faults, mappable to faults happening in CPU data caches that are read multiple times and might also be written back to the main memory. We do not account for errors in machine code in instruction caches because these can lead to non-recoverable errors. We assume that instruction caches are employed with protection mechanisms.

Energy Savings Through Unreliability

We explore execution schemes that deliberately compromise processor reliability, using NTV operation, for achieving power and energy savings in HPC codes [19]. Karpuzcu et al. [40] suggested that power savings between 10$\times$ and 50$\times$ are possible with NTV, albeit with a 5$\times$ to 10$\times$ reduction in clock frequency. Under these assumptions, a processor would consume 2$\times$ to 5$\times$ less energy per operation with NTV, compared to above-threshold voltage operation. Given a fixed power budget, a system design could replace a few cores operating in the above-threshold region with many cores operating in the NTV region. A similar strategy of trading each reliable core with many unreliable NTV cores could be applied to achieve a fixed performance target.

Experiment Setup

The experimental testbed used for testing our method consists of an HPC node equipped with four Intel Xeon E5-4650 Sandy Bridge EP processors. Each CPU offers eight cores with 32 KB and 256 KB of private caches each, and a processor-wide shared cache of 20 MB. The system runs a 3.5.0 Linux kernel and we used the GNU Compiler Collection (GCC) 4.8.2 for compilation.

Our workload —a C implementation of Jacobi— was parallelised using OpenMP. The problem size $N = 1000$ was chosen such that all data reside in the last-level cache to minimise main memory interaction not covered in the energy measurement scope (described below), while still being large enough to ensure reasonable runtimes with regard to our measurements. Time measurements were done via x86's *rdtsc* instruction and we used Intel's well-documented running average power limit (RAPL) interface to obtain energy consumption information [32]. Its PP0 domain provides readings encompassing all cores at a sampling rate of approximately 1 kHz and a resolution of 15.3 μJ, and recent work has shown it to be sufficiently accurate for our use case [25]. To achieve consistent energy readings, the target hardware was warmed up for an ample amount of time before taking measurements.

Since our target hardware system only allows reliable operation, we must simulate unreliable operation resulting in power and energy savings as well as faults. The former can be achieved by correcting the energy consumption data obtained via RAPL regarding the observations of near-threshold computing as discussed in Section 3.1.3. Moreover, to be able to simulate an arbitrary number of reliable or unreliable cores on non-configurable, commodity multi-core hardware, we need to take care when processing RAPL data, as it includes off-core entities that might be oversized or not necessarily present in some cases (i.e., ring bus for a single core). To that end, we profiled the target CPUs regarding their power consumption for all numbers of cores in a weak scaling experiment with Jacobi. Figure 3.3 shows the results of this endeavour. It indicates that the power consumption increases linearly with the number of cores with an offset for off-core entities of 7.1 W, which will be removed in subsequent data analysis to provide a fair comparison between arbitrary numbers of reliable or unreliable cores. The figure also shows a different maximum power consumption for the two CPU samples (44.3 vs 39.6 W), that can be explained by differences (e.g., in supply voltage or the temperature).

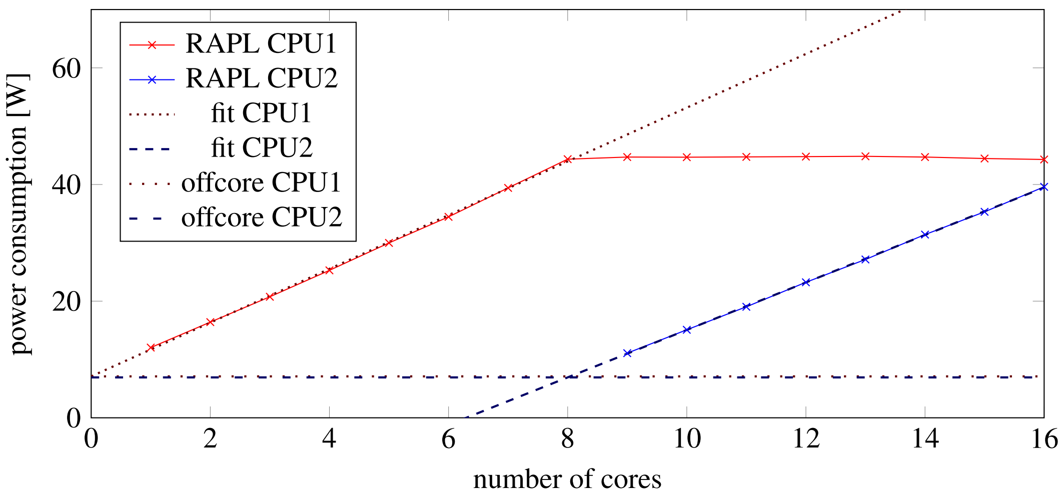


Fig. 3.3 Power consumption per number of cores on two Intel Xeon E5-4650 for a weakly scaling Jacobi run as measured by RAPL, and offcore amount as inferred via linear fitting.

As previously mentioned, our target hardware only supports reliable operation; therefore, we need to simulate faults in the software. We inject persistent faults represented by bit flips in the original data at a range of bit positions of double precision floating-point numbers prior to the computation of a Jacobi iteration. This simulates bit flips happening in the data caches of CPUs, which are accessed frequently during the computation. The implementation of the fault simulation is based on binary operators applied to the respective element in an inline function, causing only negligible performance overhead compared to the overall execution time of a Jacobi iteration. We assume a single bit flip per overall execution of Jacobi (i.e., over multiple iterations), since related work indicates that the effects of multiple faults will lead to similar observations [3] and because it reduces simulation complexity. Hence, an experiment is defined by:

- The data component in which the fault occurs (in the case of Jacobi matrix A, or vectors x or b);
- The bit position i at which a flip occurs;
- The Jacobi iteration k when the switch from unreliable to reliable mode occurs, with $1 < k < K$ and K denoting the total number of iterations, and
- The Jacobi iteration j that occurs before the execution of the fault, with $1 \leq j < k$.

To minimise simulation time, we do not inject faults at every possible element of the vectors and matrices of Jacobi but perform representative sampling (e.g., elements both on and aside the diagonal of a matrix) with respect to the algorithm. Furthermore, we employ a convergence limit of a factor of 10. This means that we consider a faulty Jacobi run as not converging if it takes more than 10 times the number of iterations of a correct Jacobi run for the same input dataset.

IEEE 754 Double-precision Floating-point Format

The data type in use for storing A, x, and b in our Jacobi implementation is the default double-precision floating-point type of C, *double*. The analysis of our results in Section 3.1.4 is based on the bit positions within the IEEE 754 binary representation of these numbers, the *binary64* format. Elliot et al. has already provided a detailed discussion regarding the effects of bit flips in this representation in [20]. Nevertheless, for clearness, we feel it is necessary to include a brief description of this representation and elaborate on the magnitude of errors introduced by bit flips, dependent on the position of the bit.

Figure 3.4 shows the binary layout of the widely-used *binary64* format. The first 52 bits (positions 0–51) correspond to the mantissa, and the following 11

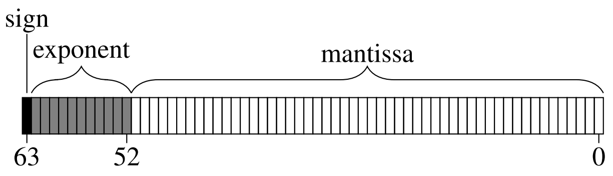


Fig. 3.4 IEEE 754 binary64 format.

bits (positions 52–62) are used for the exponent, while bit 63 denotes the sign. Furthermore, within the mantissa and the exponent, the lowest bits are the least significant ones. The decimal value v of a floating-point number is then computed using the formula:

$$v = (-1)^s (1 + \sum_{i=0}^{51} b_i 2^{i-52} \times 2^{e-1023}), \tag{3.3}$$

where s denotes the sign bit, b_i the i-th bit of the mantissa and $e-1023$ the exponent (stored with a bias of 1023). Hence, the altered floating-point number resulting from a single bit flip in position j can be expressed as:

$$a' = \begin{cases} a \pm 2^{j-52} \times 2^{e-1023} & \text{flip in mantissa,} \quad (3.4a) \\ a2^{\pm 2^j} & \text{flip in exponent,} \quad (3.4b) \\ -a & \text{flip in sign.} \quad (3.4c) \end{cases}$$

We will evaluate the effect of these perturbations on the energy consumption and execution time of Jacobi in Section 3.1.4.

3.1.4 Results

In this section, we compare executing Jacobi in parallel on unreliable hardware at NTV to sequential and parallel versions of Jacobi executed on reliable hardware at nominal voltage. The results present three cases.

First, we run Jacobi at NTV in parallel throughout its entire execution (i.e., all iterations) and analyse and discuss the energy savings that can be gained compared to a sequential run at nominal voltage. Since we are dealing with an HPC code, we will also investigate the effect of operating at NTV on performance.

Second, the same analysis is repeated when compared to a parallel execution of Jacobi.

Third, we explore the possibility of switching from NTV to nominal voltage for later iterations, motivated by our discussion in Section 3.1.2. We investigate whether later iterations of Jacobi are sufficiently significant to justify the energy and performance expense. This could create a potential trade-off, since executing late iterations at nominal voltage also prevents late faults and thereby saves convergence overhead.

Given the absence of documentation on the clock frequency impairment that results from near-threshold computing, we consider two extreme cases: a frequency reduction by a factor of 5 (i.e., 20% of the nominal frequency, denoted by $f = 0.2$, best case) and a reduction by a factor of 10 (i.e., 10% of the nominal frequency, denoted by $f = 0.1$, worst case), as discussed in Section 3.1.3. We inject exactly one error in matrix A of Jacobi under NTV execution using the process outlined in Section 3.1.2.

The results illustrate the significance of matrix A, since it is the biggest component of Jacobi in terms of memory consumption and therefore presumably more prone to faults than smaller components. Furthermore, we assume the worst case regarding the iteration before which a fault can happen (i.e., the last unreliably executed one). All results presented are averages over 50 random input datasets, with an overall variance of 10^{-5} for the relative overhead of the number of iterations for fault-injected Jacobi runs.

Sequential Reliable Jacobi

First, we investigate replacing a single, reliable core by multiple (in our case 16) unreliable cores to execute Jacobi under the same power envelope, as supported by NTVC (per-core power reductions of $10\times$–$50\times$). Hence, we assume their maximum power consumption to be equal. Figure 3.5 illustrates the results of such a series of experiments, where, in each experiment, a fault happens in a different bit position. It shows the relative energy and time savings over a sequential, reliable run of Jacobi for all possible bit positions where faults may happen.

The results show that the effects of bit flip faults on energy and time may be categorised as follows:

A: no observable loss in energy or time,

B: observable loss in energy or time, and

C: divergence.

Moreover, this classification coincides with the bit position that is flipped within an IEEE 754 double-precision floating-point number. Faults happening in bit positions 0–32 can be categorised as class A: since they show no effect on energy savings. This can be contributed to both Jacobi's resilience to faults with small magnitudes as well as the overall high precision of double-precision floating point numbers. Note that bits 0–32 are part of the mantissa and that a bit flip in these positions can affect normalised floating-point numbers by at most $2^{-20} \times 2^{e-1023}$ (see Eq. (3.4a)). As a result, energy savings of 31% and 65% are possible for a 10x (f=0.1) and 5x (f=0.2) frequency reduction respectively.

Bit positions 33–54 and 63 are classified as B:, with higher bit positions up to 54 showing a higher effect on energy and time. The maximum possible floating-point error for this class is $\pm 2^{-1}$ for the mantissa (c.f. Eq. (3.4a), bit 51) and a factor of $2^{\pm 2^{54}}$ for the exponent (c.f. Eq. (3.4b), bit 54). As such, energy savings are reduced to 48% for $f = 0.2$ in the worst case. Bit 63 is not part of the exponent but holds the sign, and as such, a flip in this position induces an absolute error of $2a$ (c.f. Eq. (3.4c)) for a floating-point number a, resulting in average energy savings of 52%. Class B: warrants protection mechanisms if the user wishes to control the performance penalty incurred by NTV execution.

The missing data points at bit positions 55–62 are a member of class C: since they are the highest significant bits of the exponent of a double-precision floating-point number (induced errors between $2^{\pm 2^{55}}$ and $2^{\pm 2^{62}}$). For our setup, flips in any of these positions aside the diagonal of matrix A cause violations in Jacobi's convergence criteria, lead to divergence, and have Jacobi break eventually with a non-silent *Inf* or *NaN* in most cases (see Section 3.1.2), which are easily detectable. Therefore, these bit positions should be protected in any case.

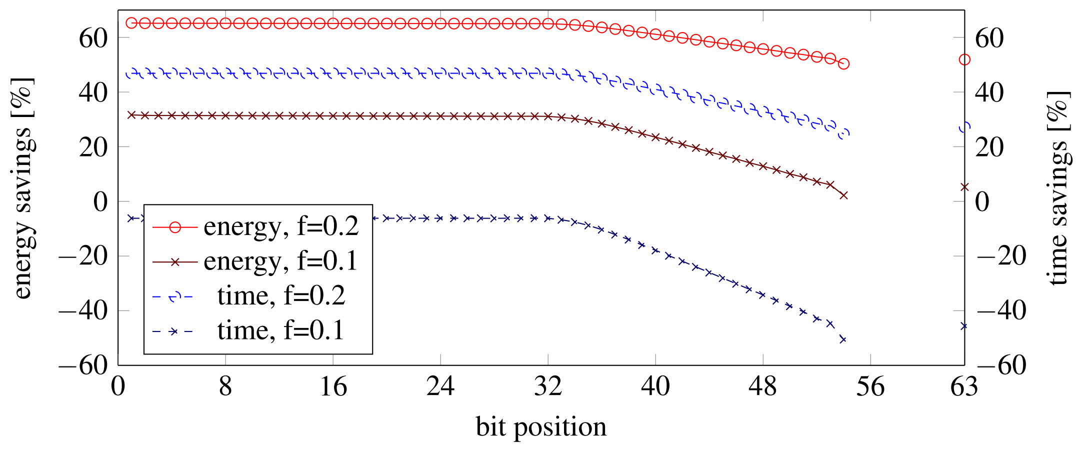


Fig. 3.5 Relative energy and time savings of an unreliable, parallel run of Jacobi on 16 cores compared to a reliable, sequential one. The missing data at bits 55–62 denote divergence.

The correlation of time and energy savings in our results is a direct consequence of both our constant workload (i.e. arguably leading to constant power consumption) and the fact that we assume the same power budget for the unreliable cores and the reliable one.

Parallel Reliable Jacobi

Our second experiment compares executing Jacobi in parallel at NTV against a parallel run at nominal voltage. The results of this comparison (Figure 3.6) show an identical classification of bit positions compared to our previous experiment. Nevertheless, one should note the lower performance compared to the previous scenario, attributed to the frequency reduction of near-threshold hardware by a factor of 5 to 10 as well as Jacobi's sub-linear scaling behaviour.

Therefore, for class A: faults, performance losses between 413% and 925% are visible. Second, energy savings increase slightly (up to 35% and 67%, respectively). This is expected due to the more energy-expensive nominal-voltage setup, since Jacobi does not scale perfectly with the number of cores. As a result, the increase in power consumption is not fully compensated for by a reduction in runtime, hence leading to a higher energy consumption. In turn, the relative energy reduction of the NTV execution increases.

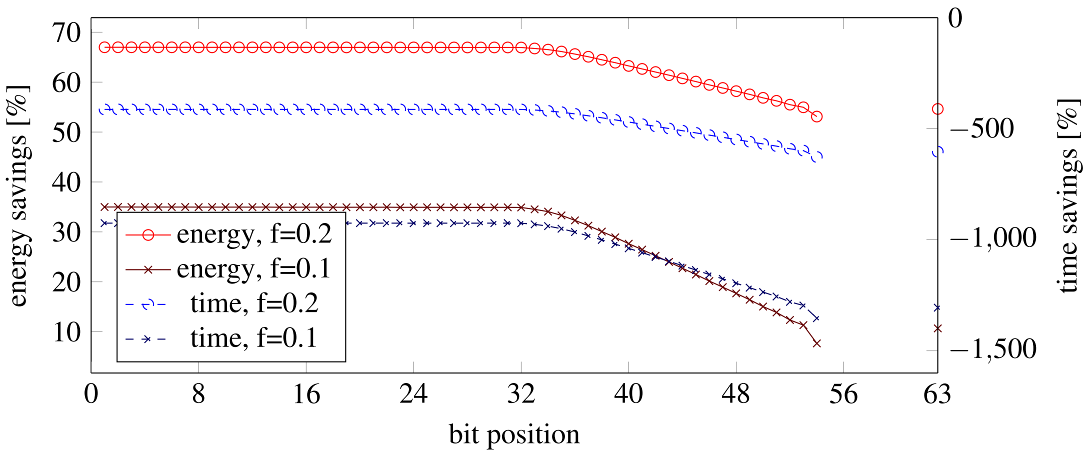


Fig. 3.6 Relative energy and time savings of an unreliable, parallel run of Jacobi compared to a reliable, parallel run on 16 cores. The missing data at bits 55–62 denote divergence.

Significance-dependent Reliability Switching

In our third scenario, we investigate whether a fraction of the last iterations of Jacobi are sufficiently significant to justify running them reliably at nominal voltage, and if so, when a switch from parallel execution at NTV to sequential execution at nominal voltage should occur. On one hand, switching to sequential execution increases run time and energy consumption. On the other hand, running at nominal voltage prevents faults and guarantees convergence, without necessitating recovery iterations.

To that end, we run experiments where we switch from NTV execution to execution with nominal voltage at three points during a Jacobi run: 75%, 85%, or 95% through completion. The intuition for this choice of switching points is Figure 3.1, where we observe that Jacobi experiences a significant slowdown when faults happen past the upper quartile of iterations. The energy consumption of these adaptive executions is depicted in Figure 3.7. For brevity, we only show results for $f = 0.2$. Switching later implies that faults in lower bit positions will have a higher effect since they happen in iterations with higher significance for convergence. Thus, switching at the 75% mark results in bit positions 0–47 that are categorised as class A:, while the same class includes only bits 0–35 when switching at the 95% mark. This naturally changes the lower bit boundary of class B: accordingly. However, it should be noted that it does not affect class C:. If a bit flip in matrix A leads to divergence of Jacobi, it will always do so, regardless of when it happens.

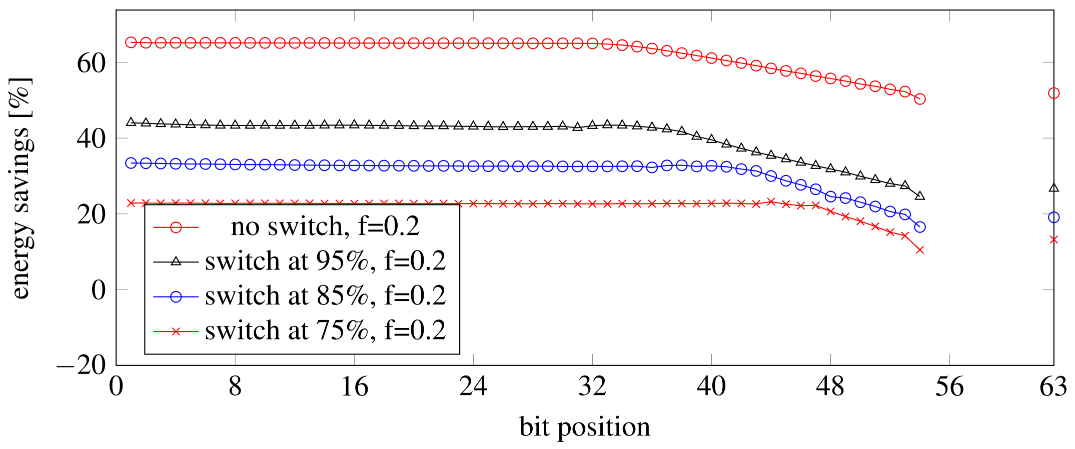


Fig. 3.7 Relative energy savings of an adaptive reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85%, and 95% runtime. The missing data at bits 55–62 denote divergence.

Overall, Figure 3.7 shows that switching at any of these three points in time does not provide a benefit if the objective is to minimise energy consumption. The best strategy is to switch as late as possible (in our case at 95%); however, all adaptive executions are outperformed by executing all iterations at NTV (65% energy savings vs 43% for switching at 95%).

Figure 3.8 shows execution time with adaptive execution. Similar observations can be made for classification of flipped bits and their effect on energy consumption. However, with NTV, the best strategy from an execution time perspective depends on the position of the bit flip, which affects the effect of late class B: faults. For example, when a flip happens at bit position 48, switching at the 75% mark yields a relative time loss of 44%, while the time loss is 60% when switching at the 85% mark and 78% when switching at the 95% mark. Furthermore, it is evident that switching at the 85% mark or earlier already yields performance losses due to the time spent in sequential execution. Overall, our results lead us to conclude that, while Jacobi does indeed show an increase in significance for later iterations, this increase is generally too small —within the boundaries of our setup— to justify switching from parallel execution at NTV to sequential execution at nominal voltage.

3.1.5 Conclusion

In this work, we explored the applicability and effect of NTV computation to a representative HPC code. We have shown that it can be a viable means of reducing

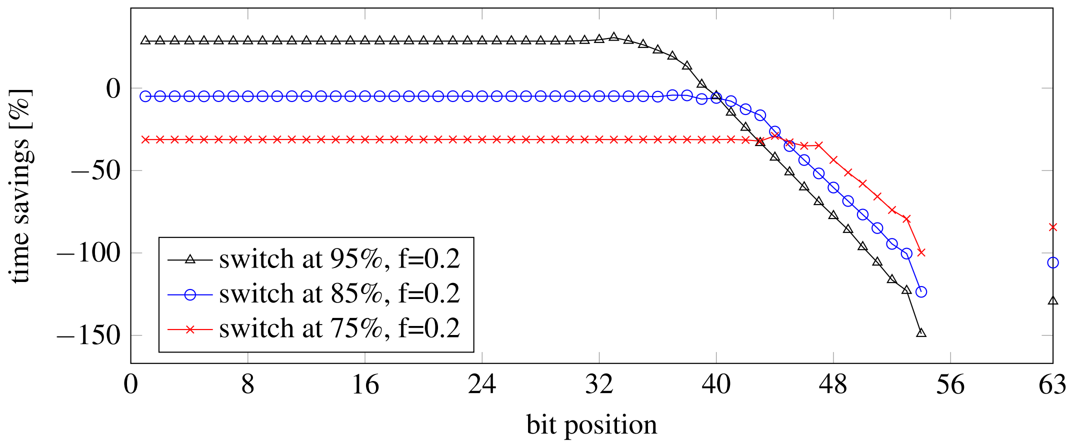


Fig. 3.8 Relative time savings of a hybrid reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85%, and 95% runtime. The missing data at bits 55–62 denote divergence.

the energy consumption, and that performance impairments caused by NTV can be mitigated via parallelism. We presented the notion of *significance-driven execution*, attributing varying significance to parts of a code or data and thereby determining whether they are candidates for NTV computation. Our results show potential energy savings between 35% and 67%, depending on the use case. As such, significance-driven execution and NTV are viable methods of reducing the energy consumption in HPC environments without compromising correctness or performance. Future research opportunities include a detailed analysis of the effect of different degrees of parallelism, protection mechanisms for intolerable faults as identified in Section 3.1.4, and investigating and comparing the significance of additional iterative HPC codes. Additionally, ways of automatically determining the significance of code regions within compiler frameworks could be explored.

3.2 Evaluating Fault Tolerance on Asymmetric Multicore Systems-on-Chip using iso-metrics

3.2.1 Introduction

The performance of today's computing systems is limited by the end of Dennard scaling [18] and the cooling capacity of CMOS technology [52]. To address these challenges, processor vendors turned towards multicore designs more than a decade ago. To further curb power consumption, power-saving techniques that were originally designed for battery-operated, embedded, and mobile appliances, such as DVFS and sleep states, were integrated into mainstream desktop and server systems. Unfortunately, these efforts seem unable to support higher performance under reasonable power budgets. Computing systems are threatened by the dark silicon phenomenon, wherein large portions of hardware real estate remain dormant to avoid power emergencies. Exploring alternative methods to better utilise hardware real estate is thus paramount.

The simple, low-power processors in smartphones and tablets as well as emerging low-voltage processors that operate near or even below threshold voltage are promising alternatives to tackle the power wall in computing systems. Broadly, these technologies attempt to modestly or drastically reduce supply voltage, while sustaining processor core clock frequencies to the greatest extent possible. However, these technologies may compromise performance and, in the case of NTVC, hardware reliability [40]. When these technologies are adopted, it is hoped that expected reductions in performance caused by frequency decay might be compensated for by including additional cores into the same power budget. In turn, this increase in hardware concurrency can be leveraged to improve performance and, in the case of NTVC, provide error tolerance by allocating cores to implement resilience techniques that address eventual data corruption.

This Section uses the principle of isometrics [11] to evaluate and compare conventional server class processors against low-power embedded architectures and NTVC processors that support ABFT. We compare these three classes of architectures under iso-power and iso-performance scenarios [82]. Specifically, we use the heterogeneous ARM cores on the big.LITTLE system-on-a-chip (SoC) as an actual embedded platform and an emulated NTVC platform. We use a high-performance Intel Xeon E5-2650 server as a baseline. To provide a fair comparison and in-depth insight, we use an important and well-studied algorithm, the CG method [71]. The CG method is a memory-bound algorithm for solving linear systems; it is gaining prevalence in HPC because it represents the type of operations and performance exhibited by many other scientific and engineering programs executing on supercomputers [2].

As an additional contribution, we describe the energy-saving potential of NTVC under a realistic application execution scenario. For this purpose, we leverage two fault-tolerant variants of CG, respectively enhanced with (i) an inner-outer detection and correction iteration [28] and (ii) a self-stabilising (SS) recovery mechanism [76] to assess the practical energy trade-offs between hardware concurrency, CPU frequency, and the hardware error rate. We again use the ARM big.LITTLE core architectures as an emulation platform.

Our central conclusion from both studies is that emerging low-power processor technologies can reliably sustain performance and significantly improve energy efficiency compared with the state of the art. For NTVC in particular, we demonstrate that highly optimised ABFT approaches effectively preserve the energy benefits of extreme low-power hardware.

The remainder of the Section is structured as follows: Section 3.2.2 discusses our experimental setup. Section 3.2.3 compares high performance and low-power processors using isometrics for performance and power. Section 3.2.4 investigates NTVC processors and the effect of NTVC on the CG method. We conclude in Section 3.2.5.

3.2.2 Experimental Setup

The CG method

The CG method is a key algorithm for the numerical solution of linear systems of the form $Ax = b$, where $A \in R^{nxn}$ is symmetric positive definite (SPD) and sparse, $b \in R^n$ contains the independent terms, and $x \in R^n$ is the sought-after solution [71]. The cost of this iterative method is dominated by a sparse matrix-vector multiplication involving A, which must be computed once per iteration. For a matrix A with nz nonzero entries, this operation requires roughly $2nz$ floating-point arithmetic operations (flops). Additionally, each iteration involves a few vector operations that cost $O(n)$ flops each.

For our evaluation, we employ IEEE 754 real double-precision arithmetic and terminate the CG loop after 2,000 iterations. Furthermore, for simplicity, we do not exploit the symmetric structure of the matrix. Under these conditions, we estimate the cost of CG to be $2nz$ flops per iteration (i.e., we disregard the lower cost of the vector operations). Moreover, for efficiency, we leverage multi-threaded implementations of the matrix-vector multiplication kernel in Intel MKL (version 11) for the Intel-based CPU. For the ARM-based cores, we rely on our own multi-threaded implementation of this operation, which is based on the CSR sparse matrix layout [28] and built upon the OpenMP parallel programming interface.

Target Architectures and Scenarios

The experiments in this Section were performed using three different multicore processors. The first processor, Xeon, is a high-performance Intel Xeon E5-2650 processor with eight cores, connected to 16 GB of DDR3-1333 MHz RAM. The alternative low-power architectures are two ARM quad-core clusters, based on Cortex-A15 and Cortex-A7 cores, in an Exynos5 SoC of an ODROID-XU board. These two clusters share 2 GB of DDR3-800 MHz RAM. Table 3.1 lists the most important features of the processor architectures considered in this Section. The row labelled 'stream memory bandwidth'reports the memory bandwidth measured using the triad test of the stream benchmark (http://www.cs.virginia.edu/stream), when executed with all cores available in the sockets. The row labelled 'Roofline GFLOPS'corresponds to the theoretical upper bound on computational performance (in terms of GFLOPS, or billions of flops per second) derived from the roofline model [94].

	Intel Xeon E5-2650	ARM Cortex-A15	ARM Cortex-A7
#Cores	8	4	4
Frequency range (GHz)	1.2 - 2.0	0.8 - 1.6	0.5 - 1.2
LLC: level, type, size (Mbytes)	L3,shared,20	L3,shared,2	L2,shared,0.5
TDP (W)	95	N/A	N/A
Peak mem. bandwidth (GBytes/s)	51.2	N/A	N/A
Stream mem. bandwidth (GBytes/s)	44	5.4	2.07
Roofline GFLOPS	11	1.35	0.51

Table 3.1 Hardware specifications of the target architectures

For our experimental analysis, we investigate scenarios that vary the number of cores, the core clock frequency, and the application benchmarks. For simplicity, we

XEON			A15			A7		
Name	Rows	#nonzeros	Name	Rows	#nonzeros	Name	Rows	#nonzeros
apache1	80800	542184	aft01	8205	125567	bcsstk21	3600	26600
cbuckle	13681	676515	bcsstk13	2003	83883	bcsstm12	1473	19659
denormal	89400	1156224	bloweybq	10001	49999	ex33	1733	22189
finan512	74752	596992	ex10hs	2548	57308	mhd4800b	4800	27520
G2_circuit	150102	726674	ex13	2568	75628	msc00726	726	34518
Pres_Poisson	14822	715804	nasa4704	4704	104756	nasa1824	1824	39208
thremal1	82654	574458	s1rmq4m1	5489	262411	plat1919	1919	32399
thermomech_TK	102158	711558	sts4098	4704	104756	plbuckle	1824	39208

Table 3.2 Sparse matrices from UFMC employed in the evaluation

consider the lowest and highest clock frequencies (disregarding Intel's turbo-mode) for each architecture. We also consider a collection of inputs that represent 'on-chip'execution, whereby the target working data fill the last-level cache (LLC) of each architecture. Our previous study in [11] exposed the poor scalability attained with CG when the key working sets reside off chip. We purposely omit off-chip working sets to avoid blurring the isometric comparison of processor architectures with memory effects. As part of future work, we intend to perform full system comparisons that also consider alternative memory technologies.

The performance of CG, as well as the fault-tolerant methods built upon it, strongly depend on the implementation of the kernel for the sparse matrix-vector multiplication. The throughput of this operation, in turn, is governed by the sparsity structure and storage layout of the matrix, which dictates the memory access pattern. To capture the behaviour of different scenarios, for each architecture, we use a set of eight sparse matrices as well as one dense matrix, all of which fit into the LLC. The sparse cases in Table 3.2 were obtained from the University of Florida Sparse Matrix Collection (UFMC). For each metric (GFLOPS, power, and GFLOPS/W), we present the average value obtained for the cases selected, for each architecture.

3.2.3 High Performance vs Low Power

In this section, we perform an experimental evaluation of the target CPU architectures, assuming they operate in nominal regions. We use the CG method, implemented on top of optimised multi-threaded versions of the sparse matrix-vector kernel from MKL and our ad-hoc OpenMP routine. The purpose of this analysis is to expose the trade-offs between performance, power dissipation, and energy efficiency for a memory-bound method, such as CG, with the goal of answering two key questions:

- **Q1 (Iso-performance):** Can we attain the performance of the Intel Xeon processor with the low-power ARM clusters while yielding a more power-efficient solution?

- **Q2 (Iso-power):** What level of performance can be attained using the low-power ARM clusters within the power budget of the Intel Xeon socket?

Trade-offs

Figure 3.9 illustrates the results from the evaluation of the multithreaded CG implementations in terms of performance (GFLOPS), power dissipation (W), and energy efficiency (GFLOPS/W). An evaluation in terms of GFLOPS and GFLOPS/W allows a comparison of these metrics for problems that vary in size and number of FLOPS performed. Table 3.3 quantifies the trends captured by the figure. Our analysis of these results is organised along three axes: number of cores (#cores), frequency, and architecture (configuration parameters) as well as three metrics. From the point of view of concurrency (#cores), increasing the number of these resources produces fair speed-ups for irregular memory-bound problems, which are slightly superior for Xeon processors and similar for both ARM architectures, independent of the frequency. For example, the use of four cores on A15 and A7 produces speedups between 2.0 and 2.4 for any of the two frequencies. The equivalent speedup on Xeon is 3.1 with four cores and both frequencies.

From the perspective of power, a linear regression fit to the data shows a high y-intercept for Xeon, which corresponds to static power, and can be explained by its large LLC, the complex pipeline, the large area dedicated to branch prediction, and other complex micro-architecture structures of the Xeon. By comparison, A15 and A7 exhibit much lower static power, reflecting the simpler design of these CPU clusters. This difference between the Intel and ARM-based architectures has a major effect on energy. Increasing the number of Xeon cores results in shorter execution time. Moreover, the large static power can be detrimental for energy efficiency (GFLOPS/W). This is a clear indicator of the potential benefits of a 'race-to-idle'policy, which can be applied to this architecture to amortise its high static power consumption. The effect of increasing the number of cores for A15 and A7 is the opposite, owing to their low static power.

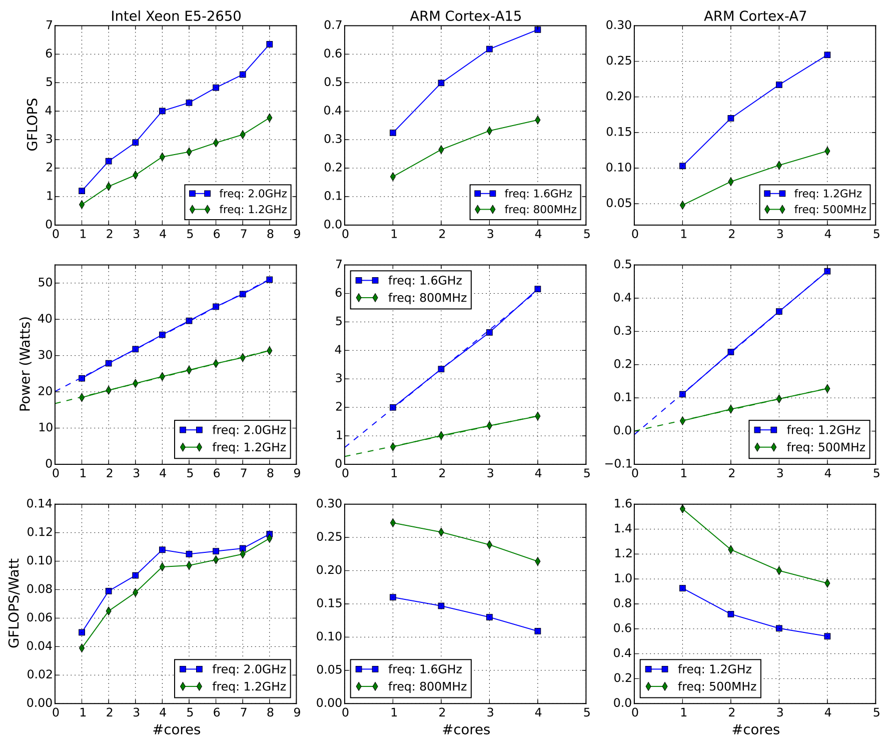


Fig. 3.9 Evaluation of performance, power, and energy on the target architectures using multi-threaded implementations of the CG method on both the on-chip and off-chip problems.

We proceed with the analysis of frequency. Independent of the number of cores, the effect of this parameter on performance is perfectly linear for Xeon but sub-linear for A15, where doubling the frequency only improves performance by a factor of approximately 1.8; the improvement is slightly higher for A7, where raising the frequency from 0.5 to 1.2 GHz (a factor of 2.4x) results in a performance increase of 2.1x. The effect of frequency on power is sub-linear for Xeon (a factor between 1.28 and 1.62x, depending on the number of cores) and is super-linear for both A15 (3.24 - 3.59x) and A7 (3.58 - 3.75x). The net effect of the variations of time and power with the frequency is that, on Xeon, increasing the frequency slightly improves energy efficiency (again pointing to a race-to-idle strategy for energy-aware execution), while on the ARM-based clusters, it reduces energy consumption by a factor close to 48% for A15 and 55% for A7

Finally, we observe some additional differences between the processor architectures. The eight-core Intel processor produces significantly higher performance

CPU	Freq. (GHz)	#cores	Time per iter. (ms)	GFLOPS	Speed-up	Power	Energy
XEON	1.2	1	3.11	0.72	1.0	18.4	0.039
		2	1.73	1.35	1.8	20.4	0.065
		4	1.02	2.39	3.1	24.2	0.096
		6	0.78	2.88	4.0	27.8	0.101
		8	0.66	3.76	4.8	31.3	0.116
	2.0	1	1.87	1.19	1.0	23.7	0.050
		2	1.05	2.24	1.8	27.9	0.079
		4	0.61	4.00	3.1	35.7	0.108
		6	0.47	4.82	4.0	43.4	0.107
		8	0.39	6.34	4.8	50.9	0.119
A15	0.8	1	1.20	0.17	1.0	0.61	0.272
		2	0.76	0.26	1.5	1.01	0.258
		4	0.55	0.36	2.1	1.69	0.214
	1.6	1	0.71	0.31	1.0	1.98	0.154
		2	0.46	0.47	1.5	3.32	0.140
		4	0.34	0.64	2.0	6.08	0.103
A7	0.5	1	1.38	0.048	1.0	0.031	1.563
		2	0.84	0.081	1.6	0.066	1.236
		4	0.58	0.124	2.4	0.128	0.966
	1.2	1	0.65	0.103	1.0	0.111	0.926
		2	0.40	0.170	1.6	0.238	0.718
		4	0.27	0.259	2.4	0.481	0.540

Table 3.3 Evaluation of performance, power, and energy on the target architectures using multi-threaded implementations of the CG method on the on-chip problems

rates and therefore shorter execution times than the ARM clusters at the expense of a much higher power dissipation rate and much lower energy efficiency (GFLOPS/W). The differences between the two types of ARM clusters also follow a similar pattern, offering higher performance with the A15 in exchange for higher power and lower energy efficiency.

Analysis of isometrics

We start the study of isometrics by noting that the questions Q1 (iso-performance) and Q2 (iso-power), formulated at the beginning of this section, can be explored in a different number of configurations/scenarios. Here, we select one that is relevant for design space exploration. Concretely, for Q1, we set the performance of one to eight Xeon cores, clocked at 2.0 GHz as the baseline. Then, we evaluate how many clusters (consisting of A15 or A7 cores, operating at either the lowest or highest frequencies) are necessary to match the reference Xeon performance. Question Q2 is the iso-power counterpart of Q1, with the baseline power budget fixed by the values obtained with one to eight Xeon cores operating at 2.0 GHz.

The left plot in Figure 3.10 reports the results from the iso-performance study. To achieve the performance of eight Xeon cores (2.0 GHz), it is necessary to use almost 10 A15 clusters (i.e., quad-cores) at 1.6 GHz or more than 51 A7 clusters at 0.5 GHz (note the different scales of the y-axis depending on the type of cluster). In this comparison, we implicitly introduce a simplification that favours the ARM processors. Specifically, on the multi-socket ARM platform, data and operations must be partitioned between the clusters, incurring an overhead associated with communication. For the CG method, we can expect that this additional cost comes primarily from the reduction vector operations (which are analogous to synchronisation). Furthermore, there is additional overhead caused by the relatively small problem sizes assigned to each core. These sources of overhead are disregarded in our study, due to the lack of access to a real platform that would allow us to measure these overheads. In that sense, our analysis presents a best-case scenario for the iso-metrics platform. In reality, for the iso-power scenario we would need to take into account the power consumed on the interconnects, which would result in a platform with less ARM platforms fitting in the power budget of a Xeon thus delivering less performance. Similarly, for the iso-performance scenario, we would need more ARM clusters to match the performance of the Xeon-based system, thus consuming more power.

The right-hand side plot in Figure 3.10 illustrates the ratio between the power rates dissipated by four configuration 'pairs' that attain the same performance. Each pair contains a Xeon core and either a A15 or A7 core, at the lowest or highest

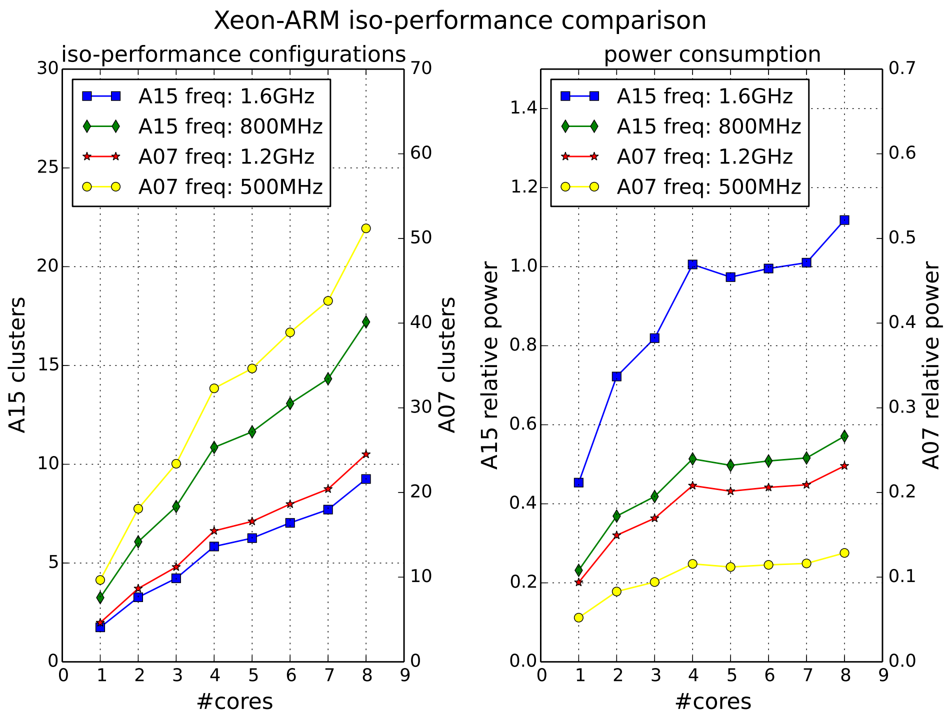


Fig. 3.10 Evaluation of iso-performance. Left: Number of A15 or A7 clusters required to match the performance of a given number of Xeon cores at 2.0 GHz. Right: Comparison of power rates dissipated for configurations delivering the same performance.

frequency. Using the previous examples, eight Xeon cores (2.0 GHz) deliver the same performance as 9.9 clusters consisting of A15 cores at 1.6 GHz; however, the A15 clusters consume 18% more power. On the other hand, using 51.1 clusters of A7 cores at 0.5 GHz only requires a fraction of the power rate dissipated by Xeon; concretely 12%.

Figure 3.11 displays the results from the complementary study on iso-power. The plot on the left illustrates that the power budget of one to eight Xeon cores can accommodate a moderate number of A15 clusters or a very large volume of A7 clusters. (Note again the different scales in the y-axis.) The performance ratio between these ARM-based clusters with respect to the Xeon, shown on the right-hand side plot, reveals decreasing gains with the increasing number of A15 clusters and equal performance with respect to seven or more Xeon cores. The ratio also decays for the A7 clusters; however, in this case, it stabilises around a factor of eight.

We note that not all ARM-based configurations considered in the iso-performance and iso-power study contain the same on-chip memory capacity (iso-capacity) as

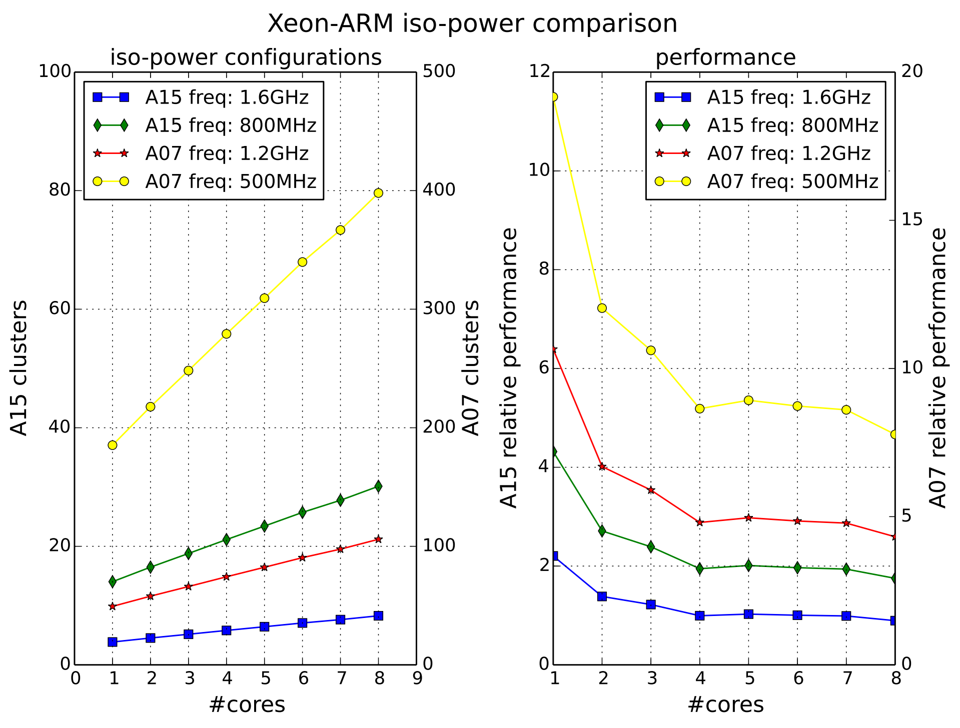


Fig. 3.11 Evaluation of iso-power. Left: Number of A15 or A7 clusters that match the power dissipated by a given number of Xeon cores at 2.0 GHz. Right: Comparison of performance rates attained for configurations dissipating the same power rate

Xeon. At least 10 A15 clusters and 40 A7 clusters would be required to achieve an iso-capacity scenario from the perspective of on-chip memory with respect to the LLC in Xeon.

3.2.4 Energy Cost of Reliability

The experiments and analysis in this section aim to explore the potential effect of NTVC on energy, a technique that trades lower processor voltage and frequency for higher concurrency but a higher failure rate. To perform this study, we make the following assumptions:

- To emulate a reliable/unreliable execution, we consider a big.LITTLE SoC consisting of several (N) quad-core A15 clusters plus the same number of A7 clusters. Here, the A15 clusters operate at the highest frequency, are reliable, and apply the fault-tolerance mechanism (i.e., the stabilising part in SS or the computations other than CG in the GMRES-based solver). On the other hand, the A7 clusters operate at the lowest frequency, operate in the NTV region, and are de facto less reliable than the A15 clusters. The A7 clusters are thus used to compute the CG iterations. We will refer to this SoC as NA15/A7, and we will use performance and energy-efficiency data for the corresponding on-chip problems for all experiments. Furthermore, we assume that idle clusters (e.g., the N A7 clusters during the execution of the stabilising operations) still contribute to the total consumption with an idle power. Safety measures integrated in the hardware prevent us from using these architectures below the minimum nominal frequency/voltage. The A7 consumes 0.128 W when used at the highest frequency. Given that these operating values are significantly lower than state-of-the-art high performance cores, we use the A7 power and performance regions as representative NTV cores.

- We employ a tuned variant of our multi-threaded implementations of the CG method equipped with an SS recovery mechanism [76] to cope with silent data corruption introduced by unreliable hardware. Following the experiments in [76], the SS part is activated once every 10 iterations in the CG method, and must be performed on reliable cores. From the computational point of view, the major difference between an SS iteration and a 'normal'CG iteration (baseline routine) is that the former performs two matrix-vector products instead of only one. However, these two operations can be performed simultaneously, as they both involve matrix A. Therefore, for a memory-bound operation, such as the matrix-vector product, we assume that, in practice, the two types of iterations share the same computational cost.

- Additionally, we consider an outer GMRES iteration that leverages CG as the inner solver [28] and integrates a conventional detection/correction mechanism for fault tolerance. The cost of this inner-outer iteration is dominated by the CG solver plus two additional matrix-vector products involving matrix A. The CG iteration can be performed on unreliable cores, but the remaining computations should be executed on reliable cores. Following the experiments in [28], we perform 20 iterations of the outer method, with the inner solver executed for 50 iterations each time it is invoked. With these numbers, 96.2% of the FLOPS are performed on unreliable cores and only 3.8% are performed on reliable cores.

- The convergence rate of the CG iteration depends on the condition number of matrix A [71]. The convergence of the fault-tolerant variants degrades logarithmically with the error rate [76]. The SDC occurs during the matrix-vector product, introducing bit flips into any of its results and propagates from there to the remainder of the computations. The convergence rate of the fault-tolerant variants also depends, to some extent, on whether the bit flips are bounded to the sign/mantissa or can also affect the exponent.

Under the conditions, we perform an experimental analysis of the energy gains that a hybrid reliable/unreliable big.LITTLE SoC can achieve, against a reliable single quad-core A15 cluster operating at the highest frequency. We compare the architectures again under iso-performance and iso-power conditions. Note that, for the latter, we still consider the power of a single idle quad-core A7 cluster.

Under iso-performance assumptions, we aim to determine how many NA15/A7 clusters must be involved during the execution of the fault-tolerant CG iterations so that, when combined, the hybrid clusters match the performance of the baseline CG solver running on a single A15 cluster operating at the highest frequency. Table 3.4 lists these values, which were determined experimentally for each problem (matrix) and fault-tolerant solver. For example, for the dense case, 7.06 NA15/A7 clusters are required to run the SS variant at the same GFLOPS rate as the baseline CG solver executed on a single A15 cluster. For simplicity, we will round this number to seven NA15/A7s. Next, we can compare the power dissipation rate of the two configurations (for the dense case and SS): 4.28 W for one A15 cluster (plus one idle A7 cluster) and 2.1 W for seven NA15/A7s. This implies that, for this benchmark case and fault-tolerant solver, it is possible to accommodate an increase in the number of iterations (decay of convergence) that is close to a factor of 2 and still attain the same ETS. Figure 3.12 reports the percentage of increase in the number of fault-tolerant solver iterations (executed on unreliable clusters) that would produce the same ETS as the baseline CG executed on the reliable platform. These results demonstrate the energy gains that can be expected from operating with simpler low-power cores at low frequencies for these applications. Concretely, depending on the benchmark case, NA15/A7 outperforms a single A15 in terms of ETS when the degradation occurs in up to 7% to 129% more iterations for SS and 178% to 359% for the fault-tolerant version of GMRES furnished with an inner-outer iteration.

benchmarks	iso-performance CG-SS	iso-performance GMRES
dense	7.06	5.02
nasa1824	9.24	7.42
bcsstk21	9.78	8.23
bcsstm12	13.1	9.35
plat1919	8.53	6.74
msc00726	13.09	9.02
ex33	11.67	8.33
plbuckle	11.94	8.71

Table 3.4 Iso-performance of reliable A15 cluster vs unreliable NA15/A7.

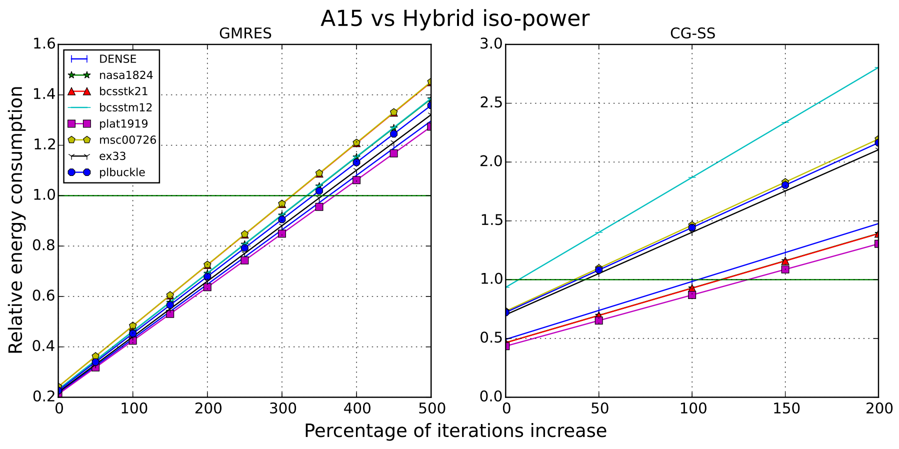


Fig. 3.12 Iso-performance ETS for the original CG (right) and GMRES (left) methods executed by A15 at the highest frequency (reliable mode) and the SS variant of CG executed by NA15/A7 under unreliable conditions that degrade convergence.

We also conducted an iso-power study. We set the power dissipated by the A15 cluster operating at the highest frequency as the baseline. We derive how many NA15/A7 clusters operating at the lowest frequency fit within the same power budget. This exercise produces the same ETS as the iso-performance analysis. This is to be expected because any increase in NA15/A7 clusters yields a proportional increase in its GFLOPS rate or, equivalently, an inversely proportional decrease in execution time. Simultaneously, the power dissipation will be increased in the same proportion, yielding the same ETS.

To conclude this section, we focus on the iso-capacity problem. For this case study, we require the aggregated LLC of the A7 clusters in NA15/A7 to equal the capacity of A15. Now, the A15 includes a 2 MB LLC cache and four A7 clusters, to match the LLC capacity of a single A15 cluster (see Table 3.1). In conclusion, we can build an NA15/A7 system that can solve problems that are the same in size to those tackled by a single A15.

3.2.5 Conclusions

The computation and data processing requirements of future systems demand more energy-efficient processors. In this study, we utilise processors designed for the mobile computing market and future processors that operate outside the nominal supply voltage regions to investigate whether they can be used to build HPC systems and data centres with better energy-to-performance ratios. We concretely show that it is possible to use power-efficient ARM clusters to match the performance of a high-end Intel Xeon processor while operating, in a worst-case scenario, within the same power budget. Conversely, it is possible to use a rather large number of ARM clusters, fit within the power budget of one Intel Xeon processor, and attain higher performance. As a further contribution, we tested a reliable CG execution in an A15 cluster and compared it with an execution of fault-tolerant variants of this method using a hybrid configuration of A15 and A7 clusters to emulate an unreliable processor that operates in the NTVC region. From this study, we find that it is possible to improve ETS, even when errors significantly slow the convergence of CG.

Because the cornerstone of the CG method is the sparse matrix-vector product, we believe that the significance of this study carries over to many other numerical methods for scientific and engineering applications. On the other hand, the study has certain limitations. For example, we did not consider factors such as the cache hierarchy, interconnection networks, memory buses, and bandwidth, which can be relevant in large-scale designs and affect both performance and power consumption. We made this choice to be able to extract some first-order conclusions about the potential of employing NTVC; however, we intend to investigate those matters in more depth in the future.

Chapter 4

Exploiting the Significance of Computations for Energy-constrained Approximate Computing

4.1 Introduction

Energy consumption is a fundamental challenge for the entire computing ecosystem, from the tetherless devices that must operate in severely energy-constrained environments to the data centres that must tame the data deluge. Large-scale computational experiments that underpin big science are hampered because the inordinate power draw of high-performance computing hardware makes the implementation of exascale systems impractical. Likewise, current technologies are too energy inefficient to realise smaller and more intelligent wearable devices for a range of ubiquitous computing applications that can benefit society, such as personalised health care.

Computing systems execute programs under the assumption that every instruction in a program is equally significant for the accuracy of the program output. This conservative approach to program execution may unnecessarily increase the energy footprint of software. Earlier work on approximate computing [4, 68, 75] shows that in several application domains, a program may produce virtually unaffected output if some parts of the program generate incorrect results or even fail completely.

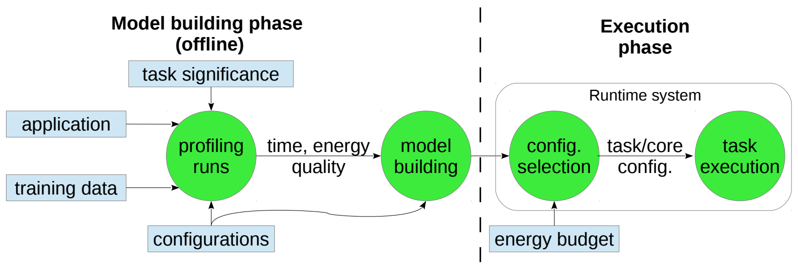


Fig. 4.1 Our methodology to maximise the output quality of applications that approximate computations while respecting a user-specified energy constraint to gracefully trade off output quality for energy reduction.

Many data-intensive applications and kernels from multimedia, data mining, and visualisation algorithms can tolerate a certain degree of imprecision.

As an example, the DCT is a module of popular video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. The DCT can be partitioned into different layers of significance because the human eye is more sensitive to lower spatial frequencies rather than higher ones. Then, 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.

Approximate computing is particularly interesting for programs that execute in energy-constrained environments. Consider, for example, an embedded system running on batteries, such as a mobile phone or an autonomous robot; when the battery is low, it may be preferable to run certain computations with a limited energy budget to prolong system lifetime, even if this comes at reduced output quality or an acceptable compromise in user experience. As another example, cloud providers contemplate billing their clients based on the energy consumption of the hosted client applications. Clients would like to make their applications energy-aware and flexible, so that the energy cost of each application fits the owner's available budget. Furthermore, the willingness of a specific client to pay for energy may vary over time.

In this Chapter, we introduce the first *significance-driven* programming framework for *energy-constrained approximate computing*. The framework comprises a programming model, a compilation-profiling-modelling tool chain and a runtime system. The programming model allows the developer to express the significance of computational tasks, depending on how strongly these tasks contribute to output quality. The developer can also provide approximate versions of selected tasks with lower complexity than that of their accurate counterparts. Approximate tasks may return inaccurate results or just a meaningful default value.

Our framework compiles and subjects each program to an offline profiling phase that uses different input datasets to measure the energy footprint of the program under different levels of concurrency, different processor frequency steps, and different degrees of approximation. This information is used to train a model, which is then employed at runtime to pick the proper configuration that achieves the highest output quality under a user-defined energy budget for a new dataset as shown in Figure 4.1.

This Chapter makes four contributions: (a) We introduce a programming model that allows the developer to structure the computation in terms of distinct tasks with different levels of significance and to supply approximate versions of non-significant tasks; (b) We introduce a profiling and model-training process to predict the energy footprint of programs as a function of the input size, the number and frequency configuration of the cores used to run the program, and the ratio of tasks that are executed accurately; (c) We introduce a runtime system that employs our model to pick the configuration that achieves the highest possible output quality within a user-defined energy budget; (d) We experimentally evaluate our approach for several application benchmarks, showing that our framework model performs very well in most cases, and achieves better output quality compared to loop perforation [79] (a well-known compiler-based approximation technique) for the same energy budget.

Specifically, our system can predict energy consumption accurately for all but three out of nine benchmarks. This prediction is used effectively by the runtime system to degrade output quality in a graceful way, even when operating under severe energy constraints (down to 20% of the energy footprint of the most efficient accurate execution). In one of the three benchmarks where our model fails to make good predictions, application behaviour depends not only on the size but also on the structure of the input data. The other two benchmarks have widely varying locality patterns, which in turn lead to additional data transfers between the last-level non-shared caches of the cores. Such inherently unpredictable programs are not amenable to profile-driven modelling and optimisation.

The rest of the Chapter is structured as follows. Section 4.2 introduces the programming model. Section 4.3 describes the runtime system that exploits our model to allow graceful quality degradation of applications. Section 4.4 evaluates the runtime system in terms of its performance and ability to enforce the policies defined at the programming model level. Section 4.5 discusses the energy modelling and prediction methodology, that our framework depends on to optimise execution based on the significance information defined by the programmer and the energy constraints defined at execution time. Section 4.6 presents the experimental evaluation of our framework on a multi-core server, using nine benchmarks that we ported to our programming model. Section 4.7 concludes the Chapter and presents directions for future work.

4.2 Programming Model

Part of the problem of energy inefficiency in computing systems is that all parts in a program are treated as equally important, although only a subset of these parts may be critical to produce acceptable program output. Tagged computations of the program, which must be executed accurately from those that are of less importance, thus can be executed approximately. Our vision is to elevate significance characterisation as a first-class concern in software development, similar to parallelism and other algorithmic properties on which programmers traditionally focus. To this end, the main objectives of the proposed programming model are to enable programmers to: (i) express the significance of computations in terms of their contribution to the quality of the output, (ii) specify approximate alternatives for selected computations, (iii) express parallelism beyond significance, and (iv) optimise and explore trade-offs via offline and online methods.

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

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

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

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

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

Listing 4.1 Programming model use case: Sobel filter

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

Listing 4.2 #pragma omp task

We adopt a task-based paradigm where the programmer expresses both parallelism and significance using `#pragma` directives; this facilitates non-invasive and incremental code transformations without extensive code refactoring and rewriting. Task scheduling decisions are taken by the runtime system, which considers resource availability and the data dependencies between tasks. The directives proposed by our model are extensions to those in the latest version of OpenMP [63]. Listing 4.1 illustrates the Sobel filter, which we use as a running example, implemented with our programming model.

Tasks are specified using the `#pragma omp task` directive (Listing 4.2), followed by the task body function. Task input and output is explicitly specified via the `in()` and `out()` clauses. This information is exploited by the runtime to detect task dependencies. Task significance is given by the `significant()` clause. It takes values in the range [0.0, 1.0], indicating the relative importance of the task for the quality of the output. Depending on their significance, tasks may be approximated or dropped at runtime. The special values 1.0 and 0.0 are reserved for unconditional accurate and approximate execution, respectively.

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

Finally, `label()` can be used to group tasks under a common identifier (name), which is used as a reference to implement synchronisation at the granularity of task groups (discussed later in this section).

As an example, lines 41-46 of Listing 4.1 create a separate task to compute each row of the output image. The significance of the tasks gradually ranges between 0.1 and 0.9 (line 45), so that there are no extreme quality fluctuations across the output image. The approximate function `row_appr` implements a lightweight version of the computation. All tasks created in the specific loop belong to the `Sobel` task group, using `img` as input and `res` as output (lines 43-44).

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

Listing 4.3 #pragma omp taskwait

Explicit barrier-like synchronisation is supported via the `#pragma omp taskwait` directive (Listing 4.3). If the `label()` clause is missing, this serves as a global barrier, instructing the runtime to wait for all tasks spawned up to that point. Otherwise, it becomes a barrier for the task group that is specified via `label()`, in which case, the runtime system waits for the termination of all tasks of that group.

Importantly, `taskwait` can also be used to control the quality of application results. Using the `ratio()` clause, the programmer can instruct the runtime to execute in an accurate way (at least) the specified percentage of tasks (globally or within a group, depending on the scope of the barrier) while *respecting* task significance. A more significant task should not be executed approximately while a less significant task is executed accurately. 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 optimisation space. Smaller ratios give the runtime more energy reduction opportunities with a potential penalty in terms of output quality. As an example, line 48 of Listing 4.1 specifies a barrier for the tasks of the `Sobel` task group. In this case, the runtime is instructed to ensure that, at a minimum, the most significant 35% of the tasks of the group are executed accurately. Note that the runtime may opt for a higher ratio (e.g., if this is feasible with the energy budget of the program).

The programming model is implemented by a source-to-source compiler, based on the SCOOP [95] infrastructure. It recognises the pragmas of the programming model, and lowers them to corresponding calls of the runtime system (discussed in Section 4.3). The resulting code is then compiled by the standard *GCC* tool chain to produce the final executable.

4.3 Runtime

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 [88].

Our runtime system is organised 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 queue runs empty, the worker may steal tasks from other worker 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 not affected by our approximate computing programming model. As such, we provide no further details on this feature.

4.3.1 Runtime API Extension

The runtime exposes an API that matches the pragma-based programming model. Every pragma in the program is translated in one or more runtime calls. The runtime API is extended to convey the new information in the programming model. Task creation is extended to indicate the *task group* and *significance* of the task as well as an alternative (approximate) task function. On the first use of a task group, the compiler inserts a call to `tpc_init_group()` to create support data structures in the runtime for the task group. This API call also conveys the per-group ratio of tasks that must be executed accurately.

An additional waiting API call is created. Next to the API call `tpc_wait_all()`, which waits for all tasks to finish, we create the API call `tpc_wait_group()` to synchronise on the completion of a task group.

4.3.2 Runtime Support for Approximate Computing

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 to which a task belongs, and (iii) the fraction of tasks that may be executed approximately for each task group. Moreover, preference should be given to approximating tasks with lower significance values as opposed to tasks with high significance values.

The runtime system has no a priori information on how many tasks will be issued in a task group, nor the distribution 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 approximately execute those tasks in each task group with the lowest significance . The policies we design must however work without this information and estimate it at runtime. We define two policies, one globally controlled policy based on buffering issued tasks and analysing their properties, and a policy that estimates the distribution of significance levels using per-worker local information.

4.3.3 Global Task Buffering (GTB)

```c
TaskDesc buffer[BUFFER_SIZE]; // to analyze tasks
size_t task_count = 0;        // buffer occupation
float group_ratio;            // set by #pragma

void buffer_task(TaskDesc t) { // called by master thread
   buffer[task_count] = t;
   task_count++;
   if (task_count == BUFFER_SIZE)
      flush_buffer();
}

void flush_buffer() { // when tasks need to execute
   sort(buffer); // sort by increasing significance
   for (i=0; i<task_count; i++) {
      if (i < group_ratio * task_count)
         issue_accurate_task(buffer[i]);
      else
         issue_approximate_task(buffer[i]);
   }
   task_count = 0;
}
```

Listing 4.4 Global task buffering policy to choose the accuracy of a task

In the first policy, the master thread buffers several tasks as it creates them, postponing the issue of the tasks in the worker queues. When the buffer is full or when the call to `tpc_wait_all()` or `tpc_wait_group()` is made, the tasks in the buffer are analysed and sorted by significance. Given a per-group ratio of accurate tasks R_g, and several B tasks in the buffer, the $R_g \cdot B$ tasks with the highest significance level are executed accurately. The tasks are subsequently issued to the worker queues. The policy is described in Listing 4.4 for a single task group. The variables described (buffer, task count, and per-group accuracy ratio) are replicated over all task groups introduced by the programmer.

The task buffering policy is parameterised by the task buffer size. A larger buffer size allows the runtime to take more informed decisions. Notably, if the buffer size is sufficiently large, the runtime can end up buffering all tasks until the corresponding synchronisation barrier is encountered and thus take a fully correct decision as to which tasks to run accurately/approximately. In our implementation, the buffer size is a configurable parameter passed to the runtime system at compile time.

The global buffer policy has the potential disadvantage that it slows the program by postponing task execution until the buffer is full and sorted. In the extreme case, the runtime system needs to wait for all tasks to be issued and sorted in the buffer before starting their execution. This overhead can be mitigated using a smaller window size and tasks of sufficiently coarse granularity, so that the runtime system can overlap task issues with task execution. Using smaller window sizes will incur the cost of not making fully correct decisions for approximate execution. Section 4.4 demonstrates that the global task buffering (GTB) policy sustains low overhead in practice.

4.3.4 Local Queue History (LQH)

The local queue history (LQH) policy avoids the step of task buffering. Tasks are issued to worker queues immediately as they are created. The worker decides whether to approximate a task right before it starts its execution, based on the distribution of significance levels of the tasks executed so far, and the target ratio of accurate tasks (supplied by the programmer). Hence, the workers track the number of tasks at each significance level as they are executed.

Formally, the LQH policy operates as follows. Let $t_g(s)$ indicate the number of tasks in task group g observed by a worker with significance s or less. These statistics are updated for every executed task. Note that the significance levels s are constrained to the range 0.0 to 1.0. In the runtime system, we implement 101 discrete (integer) levels to simplify the implementation, ranging from 0.0 to 1.0 (inclusive) in steps of 0.01. By construction, $t_g(1.0)$ equals the total number of tasks executed so far. Let R_g be the target ratio of tasks that should be executed accurately in task group g, as set by the programmer. Then, assuming a task has significance level s, it is executed accurately if $t_g(s) > (1 - R_g)t_g(1.0)$, otherwise it is executed approximately.

This policy attempts to achieve a ratio of accurately executed tasks that converges to R_g and approximates those tasks with the lowest significance level, as stipulated by the programming model.

The LQH algorithm is performed independently by each worker using only local information from the tasks that appear in their work queue. Tasks of one group are distributed among the workers via pushing of tasks to different local queues by the master and work stealing. Thus, each worker has only partial information about each group.

The overhead of the LQH algorithm is the bookkeeping of the statistics that form the execution history of a group. This happens every time a task is executed. Updating statistics includes accessing an array of size equal to the number of distinct significance levels (101 in the runtime), which is negligible compared to the granularity of the task.

The LQH algorithm requires no global snapshot of all tasks in the program and no synchronisation between workers and the master. It is thus more realistic and scalable than GTB. However, given that each worker has only a localised view of the tasks issued, the runtime system can only approximately enforce the quality requirements set by the programmer.

4.4 Experimental Evaluation of the Runtime System

We performed a set of experiments to investigate the performance of the proposed programming model and runtime policies using different benchmark codes that were rewritten using the task-based pragma directives. We evaluate our approach in terms of: (a) the potential for performance and energy reduction, (b) the potential to allow graceful quality degradation, and (c) the overhead incurred by the runtime mechanisms. In the sequel, we introduce the benchmarks and the overall evaluation approach and discuss the results achieved for various degrees of approximation under different runtime policies.

4.4.1 Approach

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%	Rel. Err.
Kmeans	A	80%	60%	40%	Rel. Err.
Jacobi	D, A	10^{-4}	10^{-3}	10^{-2}	Rel. Err.
Fluidanimate	A	50%	25%	12.5%	Rel. Err.

Table 4.1 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).

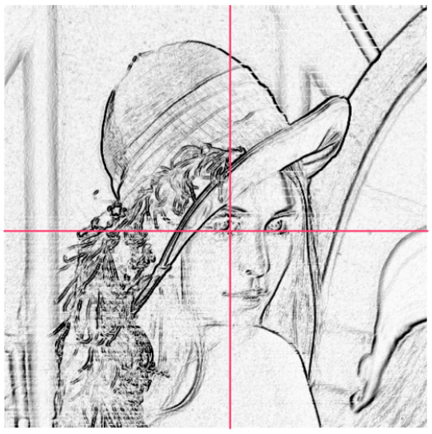


Fig. 4.2 Different levels of approximation for the Sobel benchmark.

We use a set of six benchmarks, outlined in Table 4.1, where we apply different approximation approaches, subject to the nature/characteristics of the respective computation.

Sobel is a 2D filter used for edge detection in images. The approximate version of the tasks uses a lightweight Sobel stencil with just two-thirds 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 method of assigning significance to tasks ensures that the approximated pixels are uniformly spread throughout the output image.

The DCT is a module of the JPEG compression and decompression [80] algorithm. We assign higher significance to tasks that compute lower frequency coefficients.

A Monte Carlo (MC) [90] approach is applied to estimate the boundary of a subdomain within a larger partial differential equation (PDE) domain by performing random walks from points of the subdomain boundary to the boundary of the initial domain. Approximate configurations drop a percentage of the random walks and the corresponding computations. A modified, more lightweight methodology is used to decide how far from the current location the next step of a random walk should be.

The *K-means* clustering method aims to partition n observations in a multi-dimensional space into k clusters by minimising the distance of cluster members to a cluster representative. In each iteration, the algorithm spawns several tasks, each being responsible for a subset of the entire problem. All tasks are assigned the same significance value. The degree of approximation is controlled by the `ratio` used at `taskwait` pragmas. Approximated 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.

Jacobi is an iterative solver of diagonally dominant systems of linear equations. We execute the first five iterations approximately by dropping the tasks (and computations) corresponding to the upper right and lower left areas of the matrix. This is not catastrophic, since 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, at a higher target error tolerance than the native execution (see Table 4.1).

Fluidanimate, a code from the PARSEC benchmark suite [10], applies the smoothed particle hydrodynamics (SPH) method to compute the movement of a fluid in consecutive time steps. The fluid is represented as several particles embedded in a grid. 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 4.1). 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 the programming model. As an example, Figure 4.2 visualises 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 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 result differs according to the computation. For benchmarks involving image processing (*DCT and Sobel*), we use the peak signal to noise ratio (*PSNR*) metric, whereas for *MC, K means, Jacobi* and *Fluidanimate*, we use the relative error.

In the experiments, we measure the performance of our approach for the different benchmarks and approximation degrees for the two different runtime policies GTB and LQH. For GTB, we investigate two cases. The buffer size is set so that tasks are buffered until the synchronisation barrier (referred to as the 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 [79], 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 two *Intel(R) Xeon(R) CPU E5-2650* processors clocked at 2.00 GHz, with 64 GB shared memory. Each CPU consists of eight cores. Although cores support SMT execution (hyper-threading), we deactivated this feature during our experiments. We use a Centos 6.5 Linux operating system with a 2.6.32 Linux kernel. Each execution pinned 16 threads on all 16 cores. Finally, the energy and power are measured using likwid [87] to access the RAPL registers of the processors.

4.4.2 Experimental Results

Figure 4.3 depicts the results of the experimental evaluation of the 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 4.2 and the '*Quality*' column of Figure 4.3.

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

In the case of *Sobel*, the perforated version seems to significantly outperform our approach in terms of both energy consumption and execution time. However, the cost of doing so is unacceptable output quality, even for the mild approximation level as shown in Figure 4.4. Our programming model and runtime policies achieve graceful quality degradation, resulting in acceptable output even with aggressive approximation, as illustrated in Figure 4.2.

The DCT is friendly to approximations; it produces visually acceptable results even if a large percentage of the computations are dropped. Our policies, except for

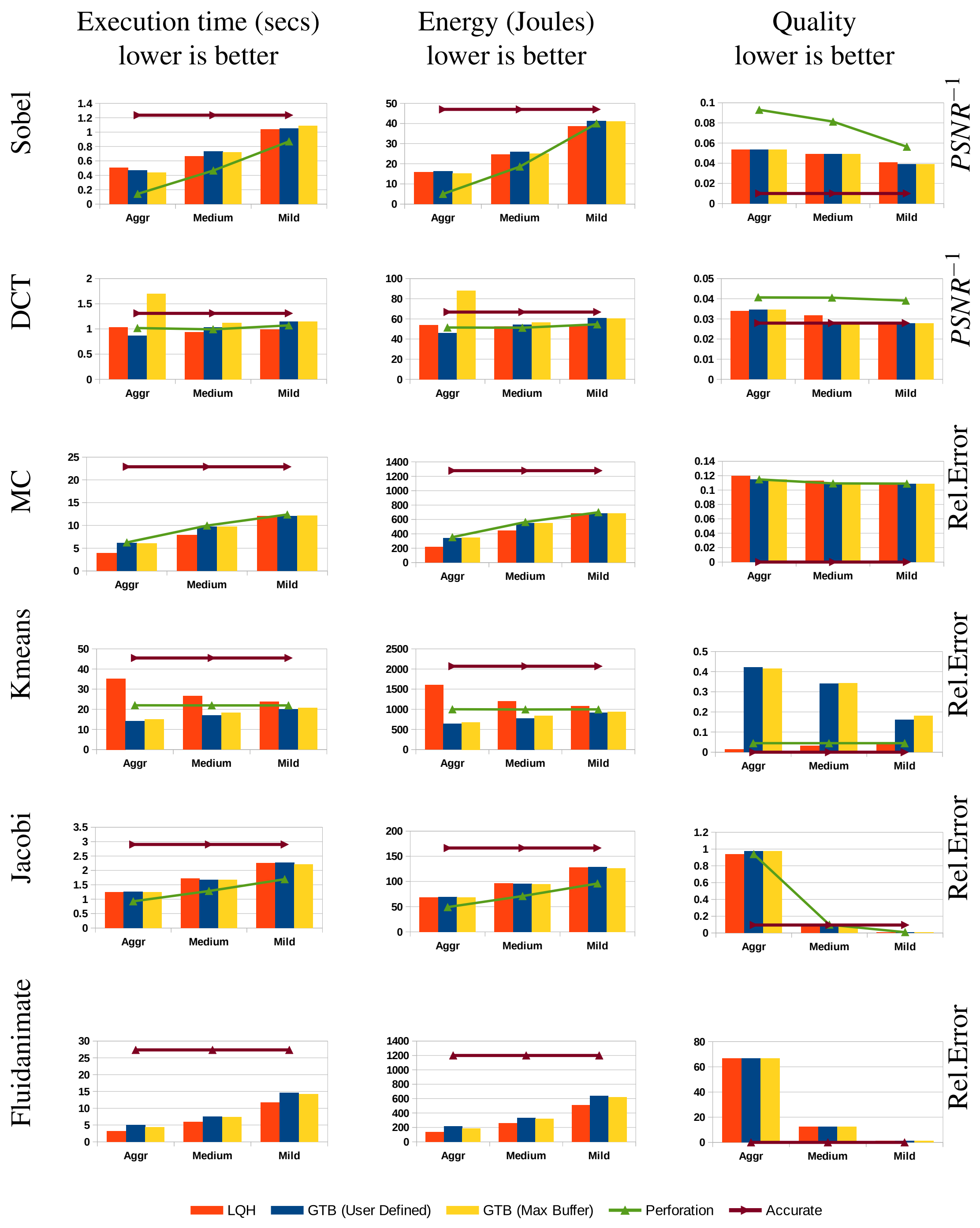


Fig. 4.3 Execution time, energy, and quality of results for the benchmarks used in the experimental evaluation under different runtime policies and degrees of approximation. In all cases, lower is better. Quality is depicted as PSNR $^{-1}$ for Sobel and DCT, relative error (%) is used in all other benchmarks. The accurate execution and the approximate execution using perforation are visualised as lines. Note that perforation was not applicable for Fluidanimate.

the max buffer version of GTB, perform comparably to loop perforation in terms of performance and energy consumption, yet result in higher quality results[1]. This is

[1]Note that PSNR is a logarithmic metric

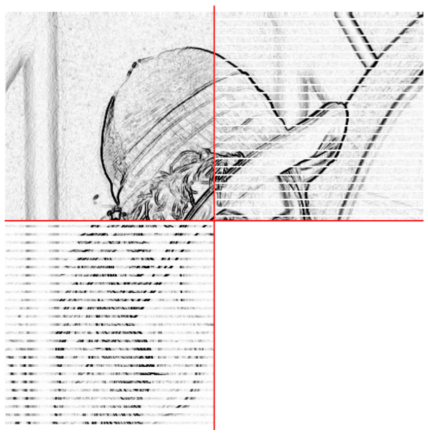


Fig. 4.4 Different levels of perforation for the Sobel benchmark. Accurate execution, perforation of 20%, 70%, and 100% of loop iterations on the upper left, upper right, lower left, and lower right quadrants respectively.

because 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. Randomised algorithms are inherently susceptible to approximations without requiring much sophistication. It is characteristic that the performance of our approach is almost identical to that of blind loop perforation. We observe that the LQH policy attains slightly better results. In this case, we found that the LQH policy undershoots the requested ratio, evidently executing fewer tasks [2]. This affects quality, which is lower than that achieved by the rest of the policies.

The K-means method 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 that move to another cluster is less than 1/1000 of the total object population. As mentioned in the Section 4.4.1, objects that are computed approximately do not participate in the termination criteria. The GTB policies

[2]Note: 4.6% and 5.1% more than requested tasks are approximated for the aggressive and the medium case, respectively.

behave deterministically, therefore always selecting tasks corresponding to specific objects for accurate executions. On the other hand, due to the effects of dynamic load balancing in the runtime and its localised perspective, LQH tends to evaluate different objects in each iteration accurately. 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, in the sense that approximations can affect its rate of convergence in deterministic, yet hard to predict and analyse ways. The blind perforation version requires fewer iterations to converge, thus resulting in lower energy consumption than our policies. It also results in 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, to ensure stability, it is necessary to alternate accurate and approximate time steps. In the 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 a 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 two versions of the GTB policy being almost as efficient.

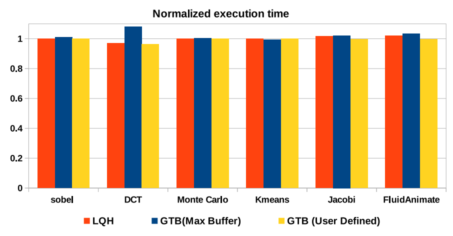


Fig. 4.5 Normalised execution time of benchmarks under different task categorisation policies with respect to that over the significance-agnostic runtime system.

Following this, we evaluate the overhead of the runtime policies and mechanisms discussed in Sections 4.3.3 and 4.3.4. We measure the performance of each benchmark when executed with a significance-agnostic version of the runtime system, which does not include the execution paths for classifying and executing tasks according to significance. We then compare it with the performance attained when executing the benchmarks with the significance-aware version of the runtime. All tasks are created with the same significance, and the ratio of tasks executed accurately is set to 100%, eliminating any benefits of approximate execution.

Figure 4.5 summarises the results. It is evident that the significance-aware runtime system typically incurs negligible overhead. The overhead reaches in the order of 7% in the worst case (DCT under the GTB max buffer policy). The DCT creates many lightweight tasks, stressing the runtime. Simultaneously, given that task creation is a non-negligible percentage of the total execution time for DCT, the latency between task creation and task issues introduced by the max buffer version of the GTB policy results in a measurable overhead.

Benchmark	(%) Inversed Significance Tasks			Average Ratio Diff		
	LQH	GTB(UD)	GTB (MB)	LQH	GTB(UD)	GTB (MB)
Sobel	2.7	0	0	0.07	0	0
DCT	2.7	0	0	0.18	0	0
MC	4.8	0	0	0.17	0	0
KMeans	0	0	0	0.9	0	0
Jacobi	0	0	0	0.12	0	0
FluidAnimate	0	0	0	0	0	0

Table 4.2 Degree of accuracy of the proposed policies.

The last step of our evaluation focuses on the accuracy of the policies in terms of respecting the significance of tasks and the user-supplied ratio of accurate tasks to be executed. Table 4.2 summarises the results. The average offset in the ratio of accurate tasks executed is calculated by the following formula:

$$ratio\_diff = \frac{\sum_{i=1}^{Groups} |requestedratio_i - providedratio_i|}{TotalGroups}$$

The two versions of GTB perfectly respect task significance and the user-specified ratio. This is expected for the max window version of GTB. The version of GTB using a limited window benefits from the relatively small task groups created by the applications and the smoothly distributed significance values in the tasks of each group. The LQH, in turn, is inherently more inaccurate due to its localised perspective. It manages to avoid significance inversion only in cases where all tasks within each task group have the same significance (K-means, Jacobi, and Fluidanimate). Even in these cases, LQH may slightly deviate from the specified ratio due to the loose collaboration of policy modules active on different workers.

4.5 Modelling and Prediction of Application Energy Footprint

We introduce 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 reason for introducing the processor frequency as one of the parameters that are explored by our model is that we have experimentally observed that the energy footprint of a computation may correlate to the combination of frequency and the task approximation ratio in a non-trivial way.

For example, Figure 4.6 depicts the energy consumption of the *fisheye* benchmark (discussed in more detail in Section 4.6.1) for 16 cores, different CPU frequencies, and different ratios of accurate/approximate tasks. The plot shows that the most energy-efficient executions when low quality can be tolerated (ratios 0.0-0.3) are at either 1.2 or 2.8 GHz, while the best frequency when targeting higher quality (ratios 0.6-0.8) is 2.4 GHz. In addition, note that 1.6 GHz is a bad choice, independent of the desired quality of the result.

Next, we describe our modelling 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 LLC and operate at the same frequency (as is the case with the popular Intel processors). We start by presenting the analytical model for the execution time of a multi-tasking computation on top of such a platform, and the energy that is expected to be consumed by it. We then discuss the process that is followed to train the model through an offline profiling and fitting phase.

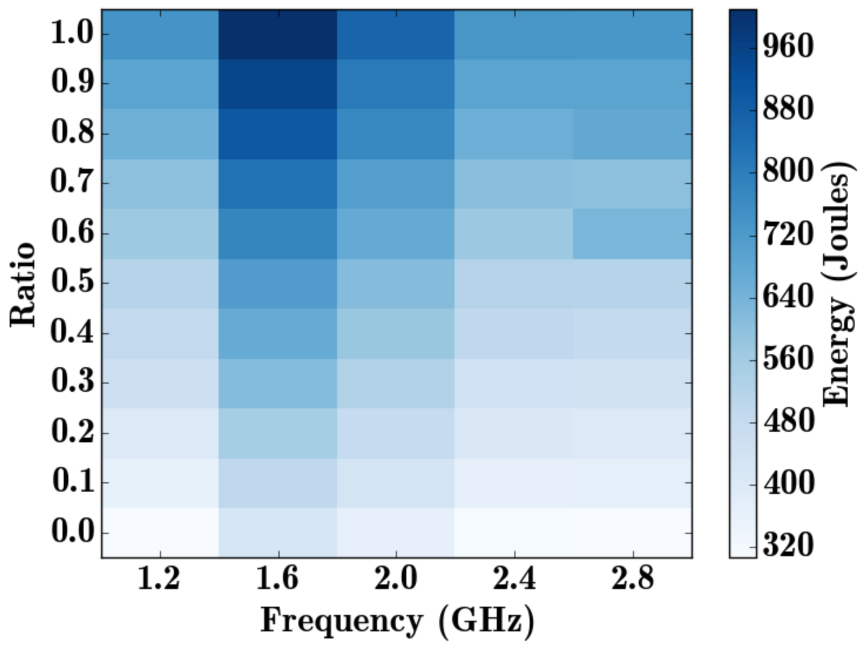


Fig. 4.6 Energy footprint of the *fisheye* benchmark under different $(CPUFrequency, TaskRatio)$ configurations.

4.5.1 Analytical Model of Execution Time

Let a computation employ m task-groups, with each group i consisting of n_i tasks. Let the accurate task ratio for group i be r_i. Additionally, 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 roughly take 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 in a purely sequential way is given by Eq. 4.1, 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.

$$T_{seq}(f,\vec{r},s,\vec{n}) = \sum_{i=1}^{m} \Bigg(n_i \cdot (r_i \cdot \overline{T_{accurate_i}(f,s,n_i)} + (1-r_i) \cdot \overline{T_{approx_i}(f,s,n_i)}) \Bigg) \quad (4.1)$$

Note that larger problem sizes s may also require a larger 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 and intermediate data footprints associated with different problem sizes

Equation 4.2 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 most off-the-shelf platforms, including the one we use in our evaluation.

$$T_{par}(f,\vec{r},s,\vec{n},c) = \frac{T_{seq}(f,\vec{r},s,\vec{n})}{c \cdot scaling(f,s,c)} \tag{4.2}$$

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 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.

4.5.2 Analytical Model of Power and Energy Consumption

The power consumption of the processing elements is given in Eq. 4.3.

$$P(f,c,s,\vec{r}) = P_{background}(f,c) + P_{dynamic}(f,c,s,\vec{r}) \tag{4.3}$$

where $P_{background}$ captures the 'background'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 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 ratio values. 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 thoroughly modelled we can now properly define the *Energy* model:

$$Energy(f,\vec{r},s,\vec{n},c) = T_{par}(f,\vec{r},s,\vec{n},c) * P(f,c,s,\vec{r}) \tag{4.4}$$

4.5.3 Offline Profiling and Model Fitting

In a profiling phase, the computation is executed with three different representative input datasets, 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 LLC of a single processor, a large working set that exceeds the total LLC capacity of all processors in the system[3] 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.

Then, a step-wise model fitting phase follows, 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 different terms of the analytical models presented above, so that they predict execution time and energy consumption of a given computation for the different configurations.

The first step is to produce estimation functions for $\overline{T_{accurate_i}}$ and $\overline{T_{approx_i}}$ in Eq. 4.1. We perform regression to map the average execution time of tasks in a 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 attempted, and we use one that minimises the prediction error with respect to profiling data.

[3]We skip problem sizes that are unrealistic. This is done for the large dataset in the Monte Carlo and MD benchmarks.

Next, we produce the function for the *scaling* term in Eq. 4.2, 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 Eq. 4.3. Again, a separate function is produced for each frequency, which returns an estimation based on the problem size, task ratio, and number of cores used. Note that $P_{background}$ 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.

The whole profiling and model-fitting process is repeated for each application, yielding different functions for each case. This application-specific information is then made available to the runtime system to pick the best configuration for a given energy budget.

4.6 Experimental Evaluation of Approximate Computing Framework

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-driven programming model. We compare our framework against loop perforation [79] in terms of quality of results under the same energy constraints.

4.6.1 Benchmarks

We apply different approximation approaches to each benchmark, subject to algorithmic characteristics of the underlying computation.

Sobel is a 2D filter for edge detection in images. The approximate version of the tasks uses a lightweight Sobel stencil with just two-thirds of the filter taps. Additionally, it substitutes the costly formula $\sqrt{sbl_x{}^2 + sbl_y^2}$ with its approximate counterpart $|sbl_x| + |sbl_y|$. Significance is assigned to tasks in a round-robin manner, which ensures that approximated pixels are uniformly distributed throughout the output.

The DCT is a module of the JPEG compression and decompression algorithm [80]. 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.

Fisheye lens distortion correction [6] 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 of the 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 neighbouring pixels of the corresponding point in the distorted image. The approximate task also performs the inverse mapping procedure; however, instead of calculating each output pixel by interpolating around the corresponding point in the input, it simply uses the value of the nearest neighbouring pixel.

The *K-means* approach 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 centres by considering the position of the points that have moved. The first phase is characterised as non-significant because errors in the assignment of individual points to clusters can be tolerated. Approximate tasks 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 centre.

A Monte Carlo (MC) [90] approach to estimate the boundary of a sub-domain within a larger PDE domain, by performing random walks from points of the sub-domain boundary to the boundary of the initial domain. Approximate configurations 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.

Canneal, a code from the Parsec benchmark suite [10], applies an annealing methodology to optimise the routing cost of a chip design. This optimisation method pseudo-randomly swaps net-list elements. If the swap results in a better routing cost, it is accepted immediately. Local minima are avoided by rarely accepting swaps that increase the routing cost of the net-list. 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.

The *MD* (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 [35]. The potential is defined as a function of distance (r) and two material specific constants (σ and ε):

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{4.5}$$

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 *MD*, the 3D container of the particles is partitioned into regions that 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. The task that performs the calculation for the region that contains the atom in question, is tagged as fully significant. The significance of tasks that are responsible for other regions drops with increasing distance to the atom home region.

Black Scholes is a benchmark of the Parsec suite [10]. It implements a mathematical model for a market of derivatives, which calculates the buying and selling of assets to reduce the financial risk. The computation of a stock price can be broken down to four blocks of code A, B, C, D, with $sig(A) > sig(B) \gg sig(C) > sig(D)$. The least important parts (C and D) are approximated using less accurate but faster implementations of mathematical functions, such as *exp* and *sqrt* [55].

There is a wide variety of applications that model the behaviour of materials when colliding or being subjected to forces. Lulesh [39] 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 it partitions the problem into a collection of elements where hydrodynamic equations are applied. We introduce an approximate version of the hourglass force calculation. Similar to *MD*, 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.

4.6.2 Experimental Methodology

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

The profiling phase uses a pool of representative input sets for each benchmark, discussed in Section 4.5. At the end of the profiling and model fitting process, each benchmark is associated with a model estimating its energy consumption per 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 unseen input sets (and input set sizes) that have not been used during the training phase for all benchmarks. 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 several 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 runtime, the ratio and concurrency level with which it can achieve execution within the requested energy budget, while minimising the effect on output quality by maximising 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 and ratio) for each case and compare it to the one selected by our system.

4.6.3 Experimental Evaluation and Discussion

Figure 4.7 summarises 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 consumed by approximate executions as a percentage of the energy consumed by the accurate execution. 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. Loop perforation is a compiler technique that drops loop iterations deemed less significant for the output quality while keeping critical loop iterations that must always be executed [79].

In practice, the oracles 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 of the configurations that consume less energy than the user-specified energy gap. The only difference between the approximate oracle and the loop-perforation oracle is that the first one accesses the approximated configurations, whereas the second one reads the loop-perforation configurations.

For the first three applications (DCT, Sobel, and fisheye) output quality is quantified using PSNR (higher is better). The PSNR is a logarithmic metric. For K-means, the metric of the quality of output is the relative difference of the average distance between data points and the centre of the cluster to which they are assigned, 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 whose energy consumptions are very similar to the optimal ones. Even in cases in which 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, K-means, and Lulesh, which are discussed in more detail later in this section. Both our approach and the optimal tend to adapt

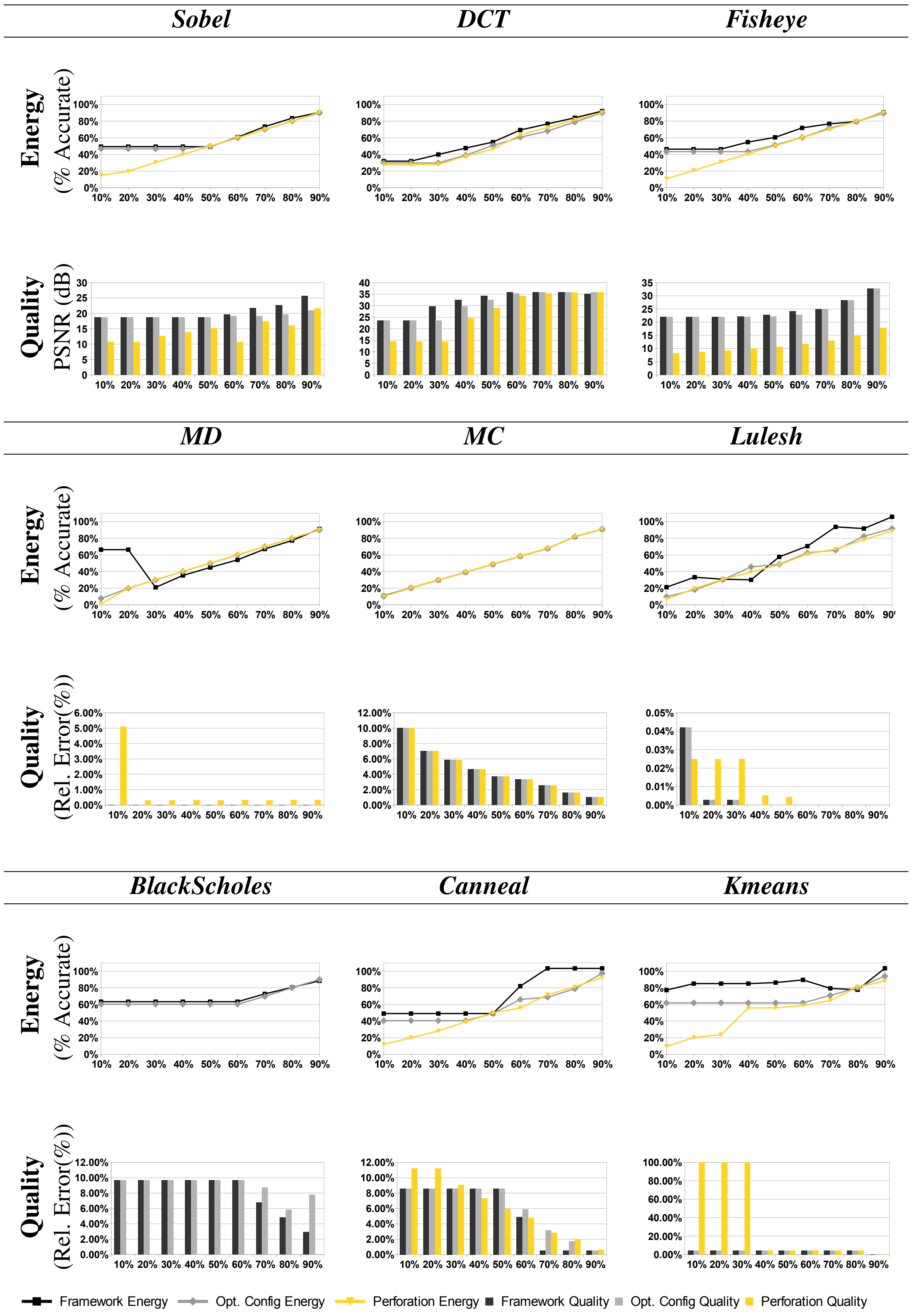


Fig. 4.7 Quality and energy metrics for different energy targets (as a percentage of the most energy-efficient accurate execution). Energy and quality plots show the results achieved by our system, an oracle selecting the optimal configuration, and loop perforation.

concurrency to utilise all cores of both CPUs. This is expected, as the dominant term in power estimation is due to the activation of additional CPUs.

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 behaviour 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.

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 that are higher than the optimal. In the case of DCT, we overspend, on average, by 6.2%. For Sobel, this number is 2.1% and finally *Fisheye* overspends by 6.4%. This leads to a pitfall in Figure 4.7 where our framework seems to outperform the oracle, which is clearly impossible. Figure 4.8 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.

The *MD* is another well-behaved application for our framework. In most cases, we choose configurations that 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 that differs by 4.5% from the user-specified energy budget. Moreover, we always achieve better quality results than the perforated version of the benchmark. With just 30% of the energy budget of the most energy efficient accurate execution, *MD* computes results with a relative error on the order of 0.0006%.

For MC, we observe that our framework makes optimal choices in almost every case. Approximation in MC drops random walks, similarly to perforation; therefore,

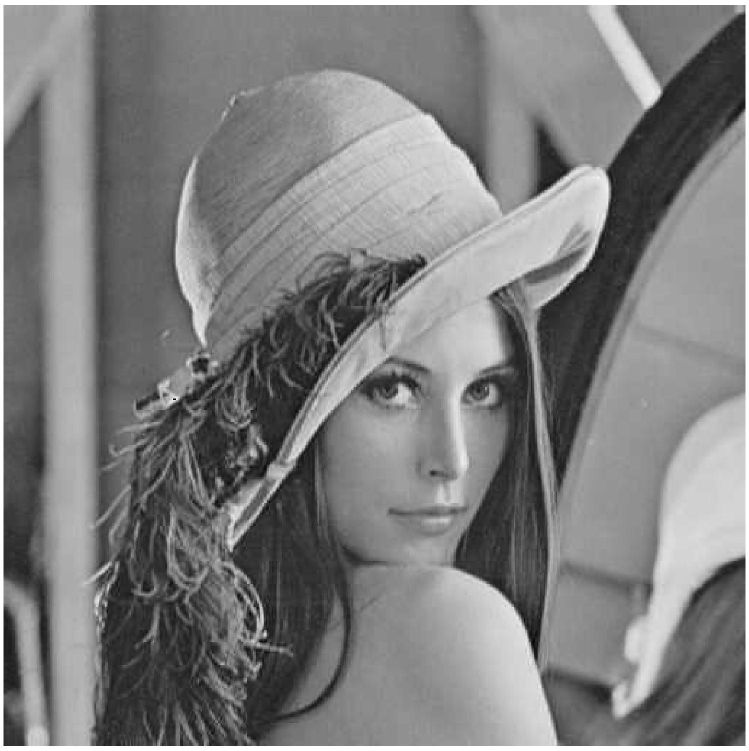


Fig. 4.8 Lena portrait compressed and decompressed using DCT with a ratio of 0.3.

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 effect 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 framework version always produces 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 case is particularly peculiar; the perforated version is better in terms of quality and energy than both our framework and the oracle. This is because the approximated executions must spend some of their energy budget to compute the significance of tasks. Furthermore, approximated 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 affects memory locality in a detrimental fashion. 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 the 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 estimations. Figure 4.9 depicts the positions of particles calculated by a small-scale approximate execution with ratio set to 0.2. Particles are coloured according to the relative error of their final position with respect to the fully accurate execution. The maximum relative error is negligible (on the order of 10^{-8}).

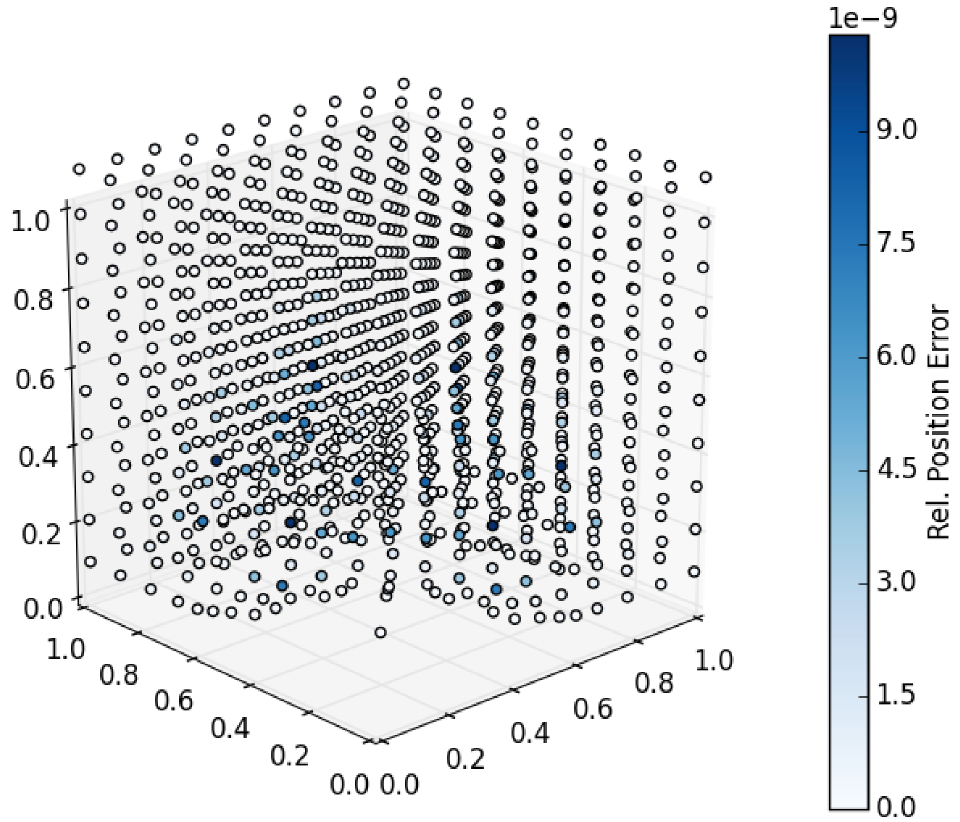


Fig. 4.9 Final positions of particles for an approximate execution with ratio 0.2. Particles have been coloured according to the relative error of their position with respect to an accurate execution.

Black Scholes calculates prices for several assets. The main loop iterates across different assets; however, there is no loop involved in the calculation of each particular asset [79]. Thus, 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; the 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 of 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 behaviour cannot be modelled accurately by our framework. Thus, we underestimate the execution time of the application and often select configurations that do not satisfy the energy constraints.

The K-means approach reveals the limitations of our approach. It cannot be modelled effectively, as it is iterative, with the number of iterations being heavily dependent on the characteristics of the input set (and not just the size of the input set). Moreover, wrong decisions in the approximate tasks (point classification) tend to increase point movement between clusters, and thus, the workload of accurate tasks (cluster centre 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 that are as good as our framework using less energy.

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

4.7 Conclusion

This Chapter introduced a directive-based programming model that allows developers to specify computational significance at the granularity of tasks. This information is used to achieve energy-constrained execution with graceful quality degradation. An offline, profile-based, training process produces a model that predicts the energy footprint of a given application as a function of its input size, the number of cores used, the processor frequency, and the ratio of accurate tasks to the total number of tasks. This model is exploited by the runtime system of an energy-constrained multi-core platform to steer execution towards a configuration that maximises quality of output while complying with energy constraints.

The experimental evaluation across several benchmark codes shows that the exploitation of programmer wisdom on the significance of computations is necessary to achieve energy constrained execution without excessive quality loss. This is particularly evident when comparing our approach against loop perforation [79], a blind approximation technique applied at the compiler level. In this work, we consider programmer wisdom to be the cornerstone of significance-driven computing. However, our intuition indicates that an automatic or at least semi-automatic significance analysis of computations may be realistic and would extend the applicability of the proposed framework.

In the future, we plan to investigate automatic significance analysis methods. We also intend to explore alternative optimisation scenarios by combining profile-based methodologies with dynamic heuristics in the runtime system. Moreover, we will investigate effective domain-specific ways to express quality constraints and use the framework to achieve automated energy-efficient execution within quality limitations. Finally, we plan to work on cost-effective ways to evaluate the intermediate quality of results at runtime.

Chapter 5

DARE: Data-Access Aware Refresh via Spatial-Temporal Application-Resilience on Commodity Servers

5.1 Introduction

Power consumption and system resilience are two of the most important challenges towards exascale computing. Projections ([7, 23]) show that the memory subsystem will contribute up to 30% of the power budget in exascale systems because of technology scaling of DRAM devices. Besides increasing power, scaling the memory capacity will increase the likelihood of errors, challenging the reliability of system operation. Clearly, solutions must address both the need for reduced power consumption and system resilient operation to be acceptable.

The ever-increasing need for memory capacity is driving the aggressive scaling of DRAM, which will continue to play a key role by offering higher density than SRAM and lower latency than non-volatile memory. However, aggressive DRAM scaling is hampered by the need for periodic refresh cycles to retain the stored data, the frequency of which is conventionally being determined by the worst-case retention time of the leakiest cell. Such an approach might guarantee error free storage, but its viability as the parametric variations worsen and resultant spreads in retention time increase are in doubt for future designs. In fact, it is becoming apparent that most of the cells do not require frequent refresh, and designing for the worst case leads to a large waste of power and throughput that may reach up to 25 to 50% and 30 to 50%, respectively, in future 32 Gb to 64 Gb densities ([48]).

Recent approaches ([48, 91]) have exploited the spatial characteristics of non-uniform retention of DRAM cells to relax DRAM refresh rates. These methods aim at enabling error-free DRAM storage by grouping rows into different retention bins and applying a high refresh rate only for rows of low retention times. However, such multi-rate-refresh techniques require intrusive hardware modifications, which are costly and hence hinder their use. Equally important is that error-free storage by these approaches may be impossible to achieve in practice since the retention time of cells changes over time ([65]). Therefore, methods that relax the refresh rates of DRAM should be aware that errors are unavoidable and must be mitigated.

While it is possible to deal with errors in DRAM with error correcting schemes ([65]), these may incur significant power consumption, negating the gains from using relaxed refresh rates. A more viable alternative is to mask errors, acknowledging the inherent error-resilient properties of many real-world applications. Recent studies ([14, 75]) have revealed numerous error-resilience properties of applications, such as probabilistic input data processing, iterative execution patterns that resolve or mitigate errors, algorithmic smoothing of the effect of errors, or user tolerance to small deviations in output quality. These error-resilience properties provide opportunities to relax the DRAM refresh cycles without concern about the resulting faults, at least on some application data.

A limited number of studies ([49, 66]) have attempted to leverage application error-resilience to relax DRAM refresh rates. Invariably, these studies build on the concept of data criticality to allocate them on either of two different memory domains: one being refreshed at the conventional worst-case rate and the other at a relaxed one. Such methods, referred to as CADA, have exhibited that application resilience can relax refresh rates, but they are still limited in three key aspects:

(i) They fail to identify and make use of the inherent ability of applications to refresh memory by accessing their own data, a property that we call *refresh-by-access*. (ii) They have not employed systematic ways to modulate the classification of data into criticality domains. (iii) They were implemented and evaluated in simulation, thus ignoring the implications of the full system stack, the time-dependencies that exist between data accesses, and cell retention.

All those limitations result in missing additional opportunities for aggressive refresh relaxation while reducing the effect of potential errors.

Interestingly, [22] predicted that integrating DRAMs in 3D die stacks will exacerbate the refresh-related overheads and propose a hardware solution that accounts for the implicit refreshes due to regular memory accesses. By tracking memory accesses, refresh skips those memory rows that have been recently accessed. Nevertheless, this approach requires intrusive and costly modifications to the DRAM memory controller, with space overhead increasing linearly with the size of DRAM memory.

In this thesis, we overcome the limitations of CADA by elevating refresh-by-access as a first-class property of application resilience to develop non-intrusive software methods for *data-access aware refresh (DARE)*, which facilitates refresh relaxation on commodity servers and software stacks without intervention in existing hardware.

We present an experimental prototype where we quantify for the first time the potential benefits of DARE standalone or in combination with other schemes, while comparing it to conventional approaches on a real server-grade system stack. Our contributions are summarised as follows:

- We identify and systematically exploit refresh-by-access to improve application resilience and enable aggressive relaxation of the DRAM refresh rate beyond what is achievable by existing techniques that rely on spatial access properties. Importantly, such an approach helps us reduce errors and enable applications that frequently access memory to operate with acceptable output quality, even when they allocate their data in memory domains without refresh.
- We introduce DARE, a novel and non-intrusive technique that facilitates implicit refresh-by-access, thus improving application resilience. Furthermore, DARE minimises the number of errors under aggressively relaxed refresh rates. It does so by reordering the execution of application tasks, based on their data read and write access patterns, while considering the retention characteristics of the underlying variably reliable memory domains. In addition, the task

reordering of DARE improves upon existing CADA-only methods by systematically moderating and increasing the amount of data that can be stored in memory domains with relaxed refresh rates.

- We realise a non-intrusive and complete system stack that integrates DARE and captures for the first time the time-dependent system and application data interactions in a system with relaxed DRAM refresh. We present a system that achieves non-disruptive operation of the whole system. The stack achieves non-disruptive operation of the whole system under relaxed DRAM refresh rates, enabling to compare and combine application-resilience techniques, including CADA, DARE, and application-level error mitigation, to evaluate their efficacy for the first time on a complete system.
- We evaluate the proposed method using a variety of multimedia, signal processing, and graph applications on a commodity system with server-grade DDR3 DRAM chips. Our findings demonstrate that it is possible to extend refresh cycles by orders of magnitudes compared to the DDRx standard specifications of auto refresh or completely disable refresh, while achieving better output quality compared to existing CADA schemes. Moreover, DARE achieves this without hardware modifications and with imperceptible performance effects.

The rest of the Chapter is organised as follows. Section 5.2 presents the proposed approach, while Section 5.3 presents an implementation of a DARE system using off-the-shelf hardware. Section 5.4 describes the executed benchmarks and presents the evaluation results. Finally, conclusions are drawn in Section 5.5.

5.2 The DARE Approach

In this section, we present the DARE approach by formalising the refresh-by-access property and developing the access-aware scheduling scheme. The objective of the proposed method is to facilitate the refresh-by-access effect, thus minimising the number of manifested errors by reordering the execution of application tasks and hence memory access patterns. This way, DARE improves the resilience of applications even under aggressively relaxed refresh rates.

5.2.1 Refresh-by-access Memory Model

The probability of error of a program variable is analogous to the time data in its memory location staying non-refreshed and the type of access to that variable: a read or a write. Reading a variable consumes its data and hence obsoletes that memory location, while writing a variable updates the data in the memory location to be consumed later, either in the program or as the output. Consuming the data propagates the error to the application, and this is the key distinction between a read and a write access. Reading a variable means its memory location is vulnerable to errors due to non-refresh periods *before* it is accessed. By contrast, writing a variable means it is vulnerable to errors *after* its memory location is updated, when this variable is actually consumed. We illustrate the time-dependent manifestation of errors and type of access interactions through a simple example.

Figure 5.1 shows timelines of memory access events of program variables. R_x denotes a *read* access to variable x and W_x denotes a *write access*. Time is quantised as an ordered set of access intervals for illustration. In this example, an application accesses two variables, namely, a and b, and those variables are stored in variably reliable memory within the time interval T_{start} to T_{end}. During this time, errors may appear due to lack of refresh. However, reordering memory accesses, while respecting data dependencies, reduces the probability of error. We show this by presenting two different, but both valid, schedules of memory accesses.

In Schedule A, the application accesses and implicitly refreshes variables a and b, at times T_3, T_5, and T_8. The vulnerability to errors for read accesses is determined by the duration between the start of non-refreshed operation up to the time the read operation happens. By contrast, for write accesses, the vulnerability to errors is determined by the time the write occurs and up to the time the non-refreshed operations end. For modelling vulnerability, we calculate the error-prone time interval for each read and write memory access. For read access, this interval is equal to the time in which data are consumed, that is, $T_{read} - T_{<start>}$, whereas for write accesses, it is equal to $T_{<end>} - T_{write}$, which is the non-refresh interval after which written data may be consumed. These vulnerability intervals by memory access characterise Schedule A in terms of how likely it is for an error to propagate to the application.

However, the vulnerability to errors can be reduced by scheduling the operations of the program differently, reordering memory accesses at different times. Schedule B shows such a reordering, by swapping R_b with W_a so that reading variable b occurs earlier, whereas writing variable a is pushed later in time. Comparing the vulnerability intervals between those schedules, shows that Schedule B reduces the time that data stay non-refreshed before they are consumed, thus it reduces the probability an error propagates to the application.

Notably, rescheduling the operations in the program must observe data dependencies and program semantics. Instruction-level parallelism and task-level parallelism are well-known ([27]) mechanisms for reordering operations. However, they have been so far used for increasing performance by improving memory locality. In addition, DARE proposes reordering at the level of tasks and, although optimising for memory locality may appear contradictory to reordering for minimising errors, we describe further in the Chapter how we address this issue and demonstrate minimal performance effects.

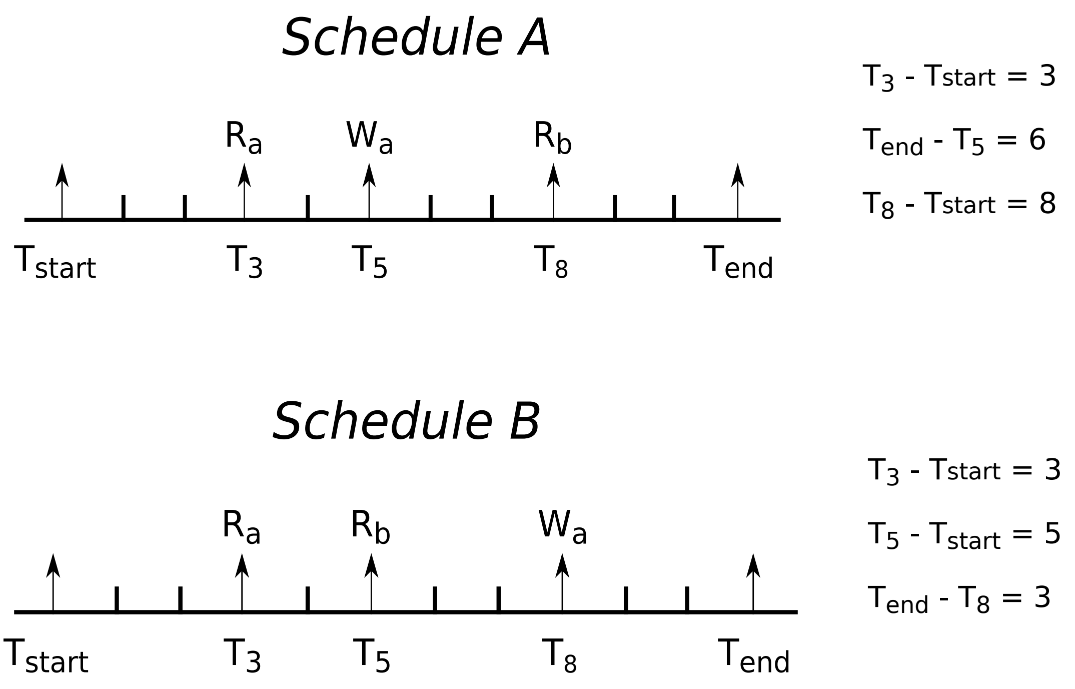


Fig. 5.1 Example of harnessing *refresh-by-access* in order to reduce the expected number of errors.

5.2.2 Access-aware Scheduling

We formulate an optimisation criterion and an associated scheduling policy to minimise the time that data remain in memory without refresh. We purposely derive a scheduling policy that has low complexity, rather than one that exhaustively covers all possible scenarios, to demonstrate the key contribution of this work, namely, that refresh-by-access can be tuned on a relevant set of applications.

We assume a *bag-of-tasks* model to describe task-level parallelism in applications. Note that the model is general enough to capture both, parallel DO-ALL loops and data-level parallelism. Moreover, an application can potentially contain multiple consecutive regions, each one matching the bag-of-tasks model independently.

Going into more detail, let an application consists of n tasks $t_1, \ldots, t_n$. Each task reads a set of memory locations denoted by R_i for task t_i. It writes to a set of memory locations denoted by W_i. Note that the bag-of-tasks model implies that there are no overlaps between any two write sets and between a pair of read and write sets. Otherwise, the tasks would incur data races. For this work, we consider only the read and write sets stored in variably reliable memory. Accesses to data stored in maximum-reliable memory have no effect on the scheduling problem.

Furthermore, we assume that the average retention time is higher than the execution time of a task, such that all the read and write sets remain valid by the end of the task. Such a minimum retention time is realistic, typically even for the nominal refresh rate. We now formulate the scheduling criterion:

$$\min_{T_1,\ldots,T_n} \left(\sum_{i:R_i \neq \{\}} E(T_i - T_{start}) + \sum_{i:W_i \neq \{\}} E(T_{end} - T_i) \right) \tag{5.1}$$

where T_i is the time when task t_i executes, $\{\}$ is the empty set and $E(T)$ is the probability of bit errors when data is residing in variably reliable memory for a period T. The $E(\cdot)$ function is characterised in Figure 2.5b. For the purposes of the scheduling algorithm, it suffices to know that it is a monotonically increasing function, which implies that $E(T)$ is minimised by minimising T.

We propose a scheduling policy that optimises Eq. 5.1 by distinguishing three classes of tasks, depending on whether the read or write sets are empty or non-empty. Tasks that only read data in variably reliable memory ($W_i = \{\}$) should be executed as soon possible to minimise $T_i - T_{start}$. Likewise, tasks that only write data in variably reliable memory ($R_i = \{\}$) should be executed as late as possible to minimise $T_{end} - T_i$. The remaining tasks either read and write variably-reliable memory, or do not access it all. These tasks are executed after the read-only tasks and before the write-only tasks.

It is common that all tasks are similar in applications matching the bag of tasks model, as they correspond to iterations of the same loop. This is also the case in our applications, including the data-dependent graph analytics case. 'In' tasks have similar execution times and equally large read and write sets. Under these assumptions, interchanging tasks within each of the three groups does not further improve the optimisation criterion. For example, Figure 5.2 shows how DARE will reorder tasks to minimise errors of Sobel, based on the optimisation criterion. Sobel tasks transform an image to emphasise its edges. With default schedule $T_i - T_{start} = 2$ for the read set of the third task, and $T_{end} - T_i = 1$ for the write set of the second task, DARE scheduling reduces both those values to zero time units, minimising the T.

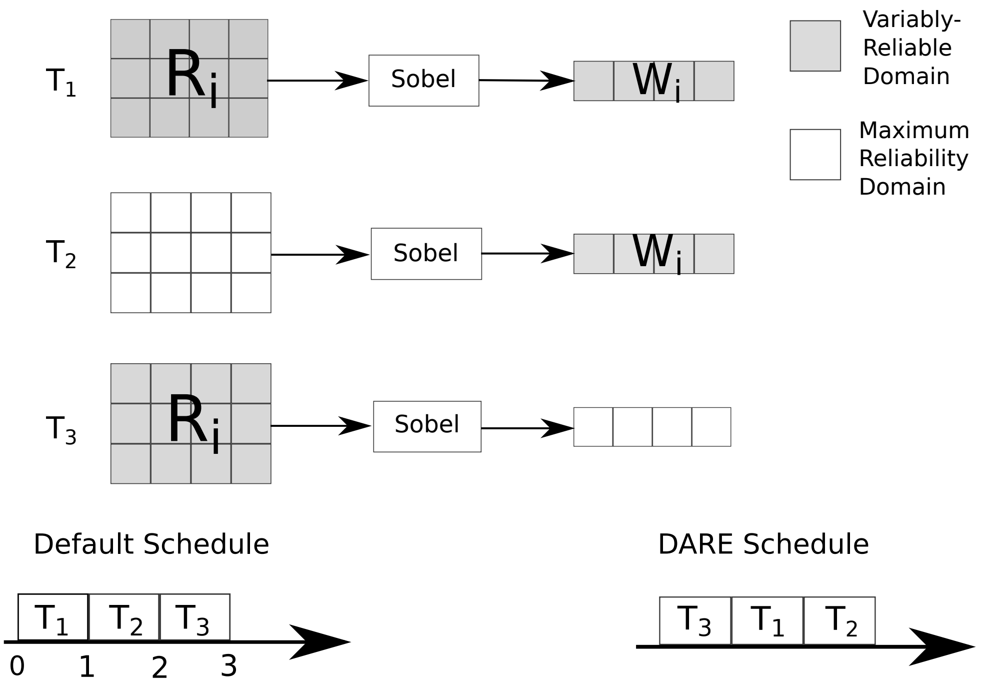


Fig. 5.2 Default and DARE scheduling

Note that an access to a single bit in memory refreshes the whole row of the DRAM bank where that bit is located. As such, accesses to one bit, word, or variable may cause others to be refreshed as well. It is possible to model these effects. All interactions between variables can be captured given precise knowledge of the memory locations of variables and knowledge of the mapping of virtual addresses to ranks, banks, and rows in DRAM. However, we need not model the memory system in this level of detail, as it is irrelevant for our applications, and we believe for many others as well. The reason is that tasks typically access data that are laid out sequentially. Moreover, those data are sufficiently voluminous to span multiple DRAM rows. Under these conditions, accidental refresh of DRAM through 'false sharing'has negligible potential.

The access-aware schedule can be extended to parallel execution by prioritising the execution order of tasks following the three groups of tasks. Whether those tasks are executed sequentially, as explained above, or in parallel has no effect. Note, however, that parallel execution reduces execution time. This way, it reduces the probability of errors even further.

5.3 Realizing DARE on a Commodity Server

As we discussed, to capture the time-dependent interactions between memory accesses and retention time, which have been neglected by existing approaches, and to evaluate DARE, it is essential to implement it on a real, complete system stack. This section discusses the proposed implementation of DARE on a commodity server with the required modifications on the Linux OS, for ensuring seamless operation under relaxed refresh, while enabling the allocation of data on variably reliable memory domains. Note that the detailed modifications that we present depend on the available hardware (e.g., the granularity at which memory refresh is controlled). However, in any case, all steps and especially the changes on the software stack can be followed to realise DARE on any commercial system.

5.3.1 Hardware Platform

Our platform is based on a dual-socket commodity server. Each socket hosts an Intel® Xeon E5-2650 (Sandy Bridge) processor, featuring an integrated memory controller (iMC) to control the DRAM device attached to the socket. The iMC exposes a set of configuration registers ([31]) to control DRAM refresh to: (i) Define the period T_{REFI} for sending refresh commands per DRAM channel and (ii) Enable or disable refresh for the entire DRAM.

```
software_refresh(refresh_period)
{
  while (true) {
    enable_refresh()
    // wait 64ms to refresh once the whole memory
    msleep(64ms)
    disable_refresh()
    // sleep for the remaining target refresh period
    msleep(refresh_period - 64ms)
  }
}
```

Fig. 5.3 Software refresh controller

The register controlling T_{REFI} has a 15-bit field for setting the period. This limits the maximum period to 336 ms up from the nominal value of 64 ms. Even though this is a 5.2× increase, it is still conservative, considering that memory cells have retention times of several seconds. Instead, we opt for a software refresh mechanism that we devise and implement to set the refresh-rate at any arbitrary value. The software refresh controller, shown in pseudocode in Figure 5.3, uses the iMC register interface to enable or disable refresh periodically for emulating a hardware refresh. The software refresh runs as a kernel thread with high priority, independently for each memory domain in the system.

In our hardware configuration, each iMC controls a single memory domain attached to the CPU socket, thus reliability domains are aligned to sockets and non-uniform memory access (NUMA) domains (Figure 5.4). The DRAMs attached to different CPU sockets may have different refresh periods to implement memory domains with a variable degree of reliability. In our dual-socket system, the first memory domain is deemed reliable with the nominal refresh applied, while the second one is the variably reliable one with a configurable refresh. We elaborate in the next section on this organisation, driven by the fact that critical data must be stored and accessed reliably.

Hardware error detection and correction codes (ECC) are a common mechanism employed in DRAMs to improve the bit error probability of the memory. In our

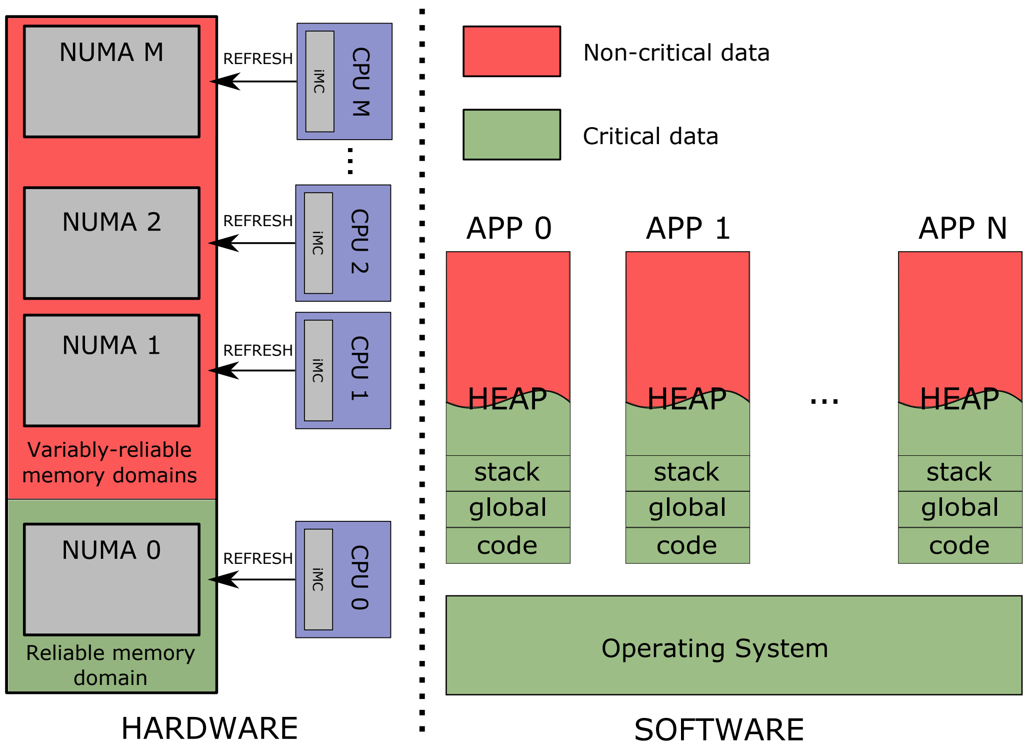


Fig. 5.4 Hardware and software setup for variably-reliable memory.

testing platform, it is necessary to disable ECC, since detection of an uncorrectable error triggers the system management mode (SMM) of the CPU and interferes with our experimentation. The SMM is a special, privileged CPU mode that suspends execution of other modes, including the operating system, to execute firmware code that typically reboots the machine to avoid catastrophic failures.

However, disabling ECC has the advantage of enabling us to study the worst-case scenario effects of reduced refresh rate operation. Using ECC would capture and correct most of the errors that occur when relaxing the refresh requirements. Evaluating the potential power and performance overhead of the ECC is out of the scope of this thesis and it is left for future work.

Before going into the details of the changes on the software stack, we would like also to propose some useful extensions to the hardware interfaces, motivated by our implementation. Although, memory controllers already have an interface to control refresh per-channel, the range of values of T_{REFI} is very conservative, up to 336 ms. As we have shown experimentally, most memory cells have retention times of seconds. A straightforward extension is to allow larger values for refresh, which in the case of the iMC, translate to larger bit-width for the configuration register, possibly with no other extensions in hardware logic. Furthermore, providing an interface to control refresh per-DIMM or even per-rank can unlock more opportunities for fine-grain, selective refreshing.

5.3.2 Software Stack

In this section, we discuss the software platform architecture for enabling variably reliable memory operation. This includes modifications to the Linux operating system to ensure crash-free system operation, and a system API to control refresh and data allocation on variably reliable memory. The hardware view of the OS is the one shown in Figure 5.4. In this setup, the physical address space is divided into two reliability domains. The *reliable memory* domain contains memory locations that are always refreshed with the nominal refresh period. On the other hand, the *variably reliable memory* domain contains memory locations for which the refresh rate can be relaxed at runtime.

In a similar fashion as reliability domains, software characterises data as critical or non-critical. Operating system kernel data are critical since even a single error in them may result in a catastrophic failure at the system level.

At the application level, data pertaining to the code and stack sections of applications are extremely vulnerable to errors too (i.e., errors in them can lead to an illegal instruction exception or a segmentation fault, which is catastrophic for the application). Thus, those data are critical too. Moreover, static data, allocated at load time, are also deemed critical. This is to avoid changing the system loader and to simplify programming abstractions for allocating data on the variably reliable memory. Our software platform design provides an API to the programmer for selecting the criticality of dynamic memory allocations on the heap, similar to other approaches ([49]). Critical data are always allocated from the reliable memory domain, whereas non-critical data allocate from the variably reliable memory. Figure 5.4 also depicts our software platform architecture.

Linux Kernel Modifications and Interface to Variably Reliable Memory

We modify the Linux kernel to create the software abstractions for the two reliability domains. This means rendering the kernel reliability aware for the memory domains and exporting a user-space interface to applications for allocating memory either from the reliable or the variably reliable domain.

For our implementation, we build on the pre-existing concept of *memory zones* ([51]) that the Linux kernel uses to group pages of physical memory for different allocation purposes. Existing Linux systems typically include a DMA zone for data operations of DMA devices, a Normal zone for virtual addresses directly mapped to physical ones, and in 32-bit systems, a High memory zone for pages that go through memory translation to map to physical pages. At the kernel level, the page allocation interface includes zone modifier flags to indicate the zone from which the kernel allocates pages. For example, when a kernel component needs memory from the DMA zone, it requests an allocation using the `GFP_DMA` flag.

Moreover, allocation from memory zones follows some priority orderings. For example, when pages from the Normal zone are requested and the request cannot be satisfied by the free pages of the Normal zone, the allocator can fall back to free pages of the DMA zone. However, allocations from the DMA zone cannot fall back to the Normal zone.

We define a new zone, namely, the *Variably reliable zone*, additionally to the DMA and Normal zones used on our platform. The size of the Variably reliable zone is defined at boot time, by kernel boot arguments denoting the start and end address of the physical address space of variably reliable memory. In our testing platform, the Variably reliable zone pools pages from the second NUMA domain, which has a relaxed refresh rate to be the variably reliable memory. Nonetheless, the zone extension we propose is generic enough to allow any subset of the physical address space to be part of the Variably reliable zone. Moreover, allocations from the Variably reliable zone may fall back to the reliable zone in case variably reliable memory is depleted, but not vice versa. The in-kernel page allocator requests a page from the Variably reliable zone using the `GFP_VREL` modifier flag, and any other allocation will be exclusively served by the rest of the reliable memory zones.

For the user-space interface, we extend the *mmap* system call for dynamic memory allocation with a new flag, namely, `VM_VREL`, which instructs the kernel that the requested memory can be allocated to physical pages of the Variably reliable zone. For the actual allocation, we extend the kernel page fault handler, so that virtual pages flagged as `VM_VREL` map to physical page frames of the Variably reliable zone. Note that the `VM_VREL` is valid only for heap allocations and is otherwise ignored to ensure that code, stack, and static data during application loading are always stored reliably.

Lastly, we expose a system interface to set the refresh period on the variably reliable memory domain. For our setup, we extend the `sysfs` kernel interface to have a `refresh_period` entry in milliseconds for each NUMA domain under its respective `/sys/devices/system/node/nodeX` device tree. Setting the `refresh_period` entry of a NUMA domain changes the refresh period of the software refresher kernel thread for this domain – the special value 0 means no refresh at all. A similar interface is usable even if reliability domains are decoupled from the NUMA architecture, by adding memory domain proxies in the device tree. Notably, the `sysfs` interface is at the system level; thus, it needs administrator privileges to be set. We do not envision this interface to be used by application programmers but rather by system administrators. The presented modifications allow plugging in DARE on any Linux-based system.

5.4 Evaluation

In this Section, we evaluate the efficacy of the DARE approach on minimizing the manifestation of errors and compare it with CADA only techniques. For performing the evaluation, we use the hardware and software platform presented previously and carefully select a range of applications from various domains with different error-resilient properties. Note that both DARE and CADA are evaluated for the first time on a real system. Therefore, our evaluation campaign reveals interesting results and shows how the time dependencies affect application error-resilience and the true performance of DARE and CADA schemes. For each benchmark, we use CADA as the baseline technique to customize data allocation and augment it with DARE, amenable to the specific characteristics of each application. For experimentation, we vary the percentage (PR) of data stored in reliable memory, using conventional refresh, to create different vulnerability scenarios. Moreover, the variably-reliable memory domain operates with no refresh at all, to test the efficacy of error-resilience techniques at the extreme. For each experiment configuration we quantify the output quality in terms of the related metric for each benchmark and measure the time overhead of the applied technique. Each experiment is repeated 10 times and present the average of the results across runs. The compiler used is GCC version 4.8.5. Finally, we discuss the impact of our approach on power and performance for current and future DRAM densities and to compare the gains between approaches under fixed, iso-quality comparisons.

5.4.1 JPEG and Discrete Cosine Transform (DCT)

JPEG is a widely used application for image compression based on DCT, composed of two parts: (i) compression, during which DCT is applied on the input image and its output is then quantized and stored on memory, and (ii) decompression, during which the stored compressed image in the form of quantized coefficients is dequantized and reconstructed using inverse DCT (IDCT). In the experiments, the input of DCT is a grid of 64×64 image tiles, while each image is 512×512 pixels. The memory consumption of the DCT output, to which we focus, amounts to 8GB.

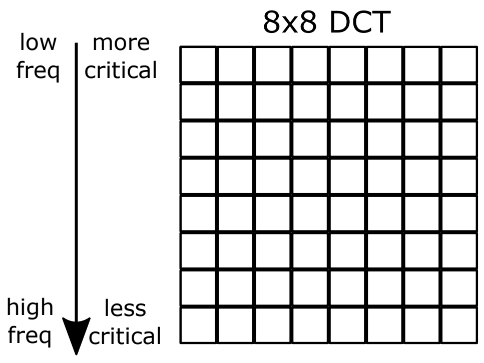
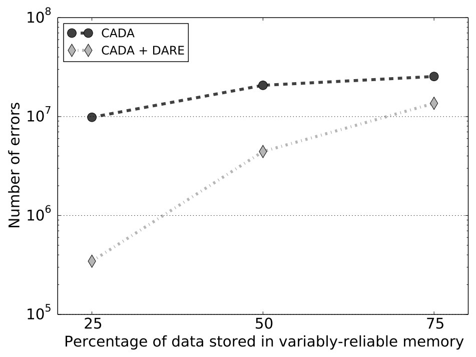
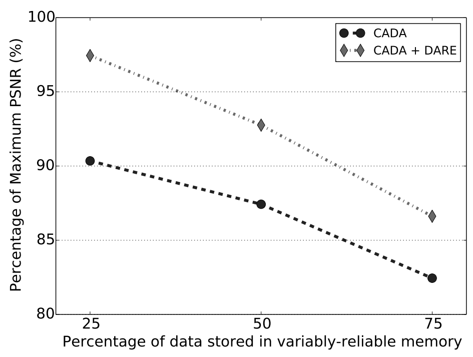


Fig. 5.5 Data criticality within an 8×8 DCT block

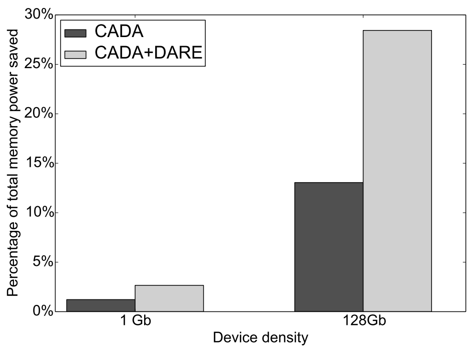
CADA It was shown ([38]) that in DCT some coefficients, i.e., low-frequency ones, are more significant for determining the output quality than the rest, high-frequency ones. This means DCT is resilient when errors manifest in high-frequency coefficients, since their impact on output quality is limited. The application of CADA to DCT is straightforward in this case. The input image is set as critical data, thus stored in the reliable memory, together with the low-frequency coefficients of each



(a) Reduction on the number of errors



(b) Quality improvements on DCT



(c) Power savings setting PSNR to 90% of best

Fig. 5.6 Comparison of application resilience techniques for DCT using variably-reliable memory without refresh.

of the 8×8 DCT computation blocks that constitute our kernel. Figure 5.5 shows the location of the critical coefficients within a DCT block. The lowest frequency, top left coefficient is the most critical and criticality reduces significantly, moving towards the bottom right part of the block. Each task in our experiments reads a 8×8 block of the reliable stored input image, computes the DCT coefficients for a row and stores them to an output array. There is not an exact threshold to how many of the top rows need to be stored reliably to decisively affect output quality. For experimentation, we vary this threshold, translated as the percentage of the output data stored reliably, to investigate its impact on quality. For a given output quality, the more data stored in the variably-reliable memory, the less the power consumption due to relaxing refresh on larger parts of memory.

DARE Following our task model, all tasks read from the reliable memory but depending on the threshold for storing rows reliably, some of them write their output on the variably-reliable memory. Note that DARE reorders the execution of tasks schedule to ensure that tasks writing on variably-reliable memory execute as late as possible, to minimize manifested errors from relaxed refresh execution.

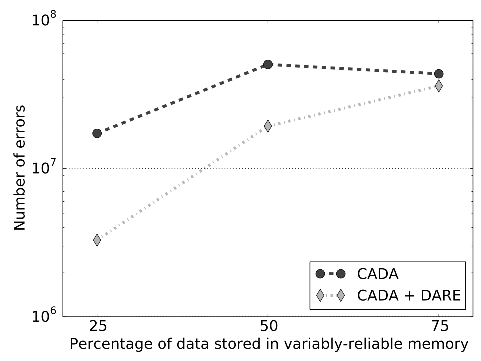
Quality Metric The quality metric for DCT is the Peak Signal-to-Noise Ratio (PSNR), calculated by comparing the original to the reconstructed image. For the tiled images input, we take the minimum PSNR among all tiles in the grid to perform a worst-case analysis.

Results Figure 5.6 shows results on the number of errors and output quality, varying PR ratios to 25%, 50% and 75% of the data store in the variably-reliable memory. Note that a PR of 0%, meaning all data are stored in the reliable memory, does not utilize the inherent error resilient properties; whereas a PR of 100%, meaning all data are stored in the variably-reliable memory, cannot clearly show the effects of DARE since all tasks read and write variably-reliable data, hence no reordering is meaningful.

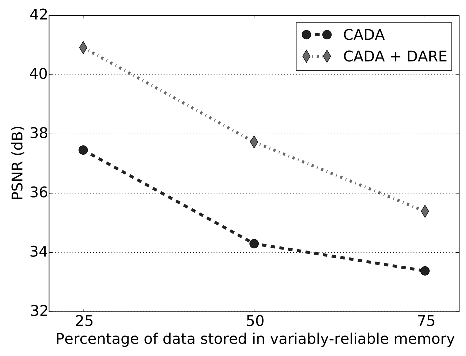
Figure 5.6a shows the number of manifested errors in case CADA-only applies versus augmenting CADA with DARE. DARE reduces the number of errors, up to an order of magnitude when 25% of the data are stored in the variably-reliable memory. Figure 5.6b depicts the quality as a percentage of the maximum PSNR achievable when refresh is enabled. Notably, the combination of CADA and DARE improves quality consistently, up to 8% in case of PR=25%. From another point of view, CADA+DARE achieves the same quality as CADA, but by storing a larger portion of data in variably-reliable memory, hence enabling relaxed refresh operation on a larger part of memory. An iso-quality comparison fixing the target PSNR to 90% of the best possible shows that CADA achieves this target by having at most 25% of the data versus 50% of the data for CADA+DARE stored in variably-reliable memory. Importantly, CADA+DARE saves more power compared to CADA-only, by enabling to store more data to the variably-reliable domain. Indicatively, Figure 5.6c shows power savings for the PSNR 90% iso-quality target for current, 1Gb, memory technology and future, 128Gb, technology. Power savings project to be up to 28% by combining CADA+DARE techniques compared to about 14% of CADA-only.

5.4.2 Sobel filter

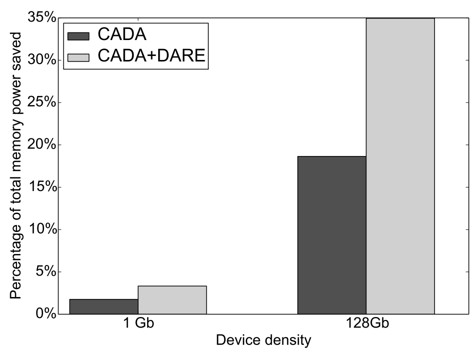
Sobel is an image processing filter that is used frequently in edge detection applications. Similar to DCT, the input consists of a grid of 200×200 images, each 512×512 pixels, while the output is a grid of the same dimensions, resulting in a total memory consumption of 20GB.



(a) Reduction on the number of errors



(b) PSNR from standard output



(c) Power savings setting PSNR=35dB

Fig. 5.7 Comparison of application resilience techniques for Sobel using variably-reliable memory without refresh

CADA For Sobel, we relax criticality for both the input and the output data, allocating parts from both data sets to variably-reliable memory. In contrast to DCT, there is no a priori algorithmic property to indicate which parts of the input or output are more significant in determining the output quality. For this reason, we randomly select data to store in the variably-reliable domain using a uniform distribution. Moreover, experiments vary the amount of data stored in the variably-reliable memory, to show the impact on output quality in relation to the application resilience techniques.

DARE In order to compute one pixel (i, j) of the output image, a task reads the (i, j) and all the neighboring pixels of the input image. As a result, to compute and write out the i-th row of the output image, a task reads rows $i-1$, i and $i+1$ from the input. Depending on the randomly selected data allocation, a task may only read or only write or both read and write data in the variably-reliable domain. Following the principle of DARE scheduling, tasks that read from variably-reliable memory execute first, followed by tasks that both read and write on it, while write-only tasks execute last, to minimize the expected number of manifested errors.

Quality Metric The output quality is quantified in terms of the PSNR between the output image from relaxed refresh operation and the reference output produced with full reliability enabled. In our evaluation we report the minimum PSNR across image tiles for performing a worst-case analysis.

Results Figure 5.7 shows the results for Sobel. The PR ratios allocated on the variably-reliable memory are 25%, 50% and 75% of the aggregated input and output data. Applying CADA+DARE reduces consistently errors and improves the PSNR of the output compare to CADA-only. Under an iso-quality comparison, setting the target PSNR to 35dB, CADA achieves this target by having at most 25% of the data stored in the variably-reliable memory, whereas CADA+DARE achieves that even when 75% of the data stored in the variably-reliable memory. As with DCT, CADA+DARE significantly improves application resilience, enabling to relax data criticality to a greater extent than CADA alone, thus allowing to minimize the data that need to be stored reliably. Figure 5.7c shows the power savings for the existing and future memory technologies under an iso-quality comparison of CADA versus CADA+DARE, setting PSNR equal to a minimum 35dB target. Savings reach up to 35% in future DIMMs, leveraging the ability of CADA+DARE to place more data on the variably-reliable domain, thus requiring less reliable memory, compared to 18% of CADA-only techniques.

5.4.3 Pochoir Stencil Algorithms

We demonstrate DARE resilience techniques on stencil algorithms extending the Pochoir stencil compiler ([85]). Pochoir optimizes the computation of stencil iterations using a cache-oblivious divide-and-conquer strategy. The algorithm decomposes the space-time iteration domains using trapezoidal shapes, to improve memory locality. In terms of memory consumption, Pochoir allocates two large array data structures to store the previous and current values of physical quantities to compute each grid point.

For experimentation, Pochoir simulates a 2-dimensional heat dissipation problem with mirroring boundary conditions. Moreover, we vary the grid size (spatial dimensions) from 14K to 30K.

CADA In stencil algorithms no errors are tolerable. This is because errors propagate to neighboring grid points, violating the algorithm's correctness and convergence criteria. Thus, criticality-aware data allocation techniques are unable to enable error resilience from data placement. However, refresh-by-access resilience techniques can relax data criticality, as we discuss next.

Refresh-by-Access The data values stored at the grid points are refreshed regularly due to the iterative execution of the algorithm, reading and writing those points. In case the problem size is small, less than the retention time of memory, this iterative read and write refresh is sufficient to execute error-free.

However, for larger problem sizes, the effectiveness of the refresh-by-data access diminishes because the default Pochoir decomposition favors memory locality, updating the same set of grid points at later time steps before computing another set. This computation strategy delays accessing grid points in the space domain, hence reduces the implicit refresh-by-access. Next, we discuss how the DARE technique applies to Pochoir to facilitate refresh-by-access.

DARE Our modified version of Pochoir performs time-cuts crossing the full spatial domain at regular intervals to enable refresh-by-access. Figure 5.8 shows this computation strategy. All trapezoids below Δt_i are visited before the algorithm proceeds to later time steps to trade cache locality for error-resilient operation. Nevertheless, the time-cut interval Δt_i is tunable to incur the least possible performance loss while ensuring correctness, we demonstrate this in our results.

Quality Metric Due to the fact that stencil algorithms are not resilient to errors, the quality metric is measured as the percentage of correct runs over the total number of runs of the program.

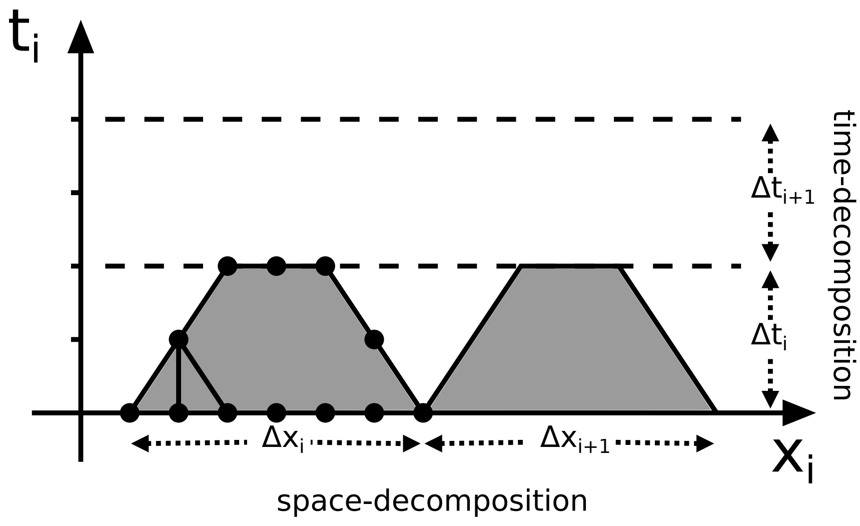


Fig. 5.8 DARE trapezoidal decomposition applied in the Pochoir compiler

Results Figure 5.9 shows the results by contrasting the original to DARE computation decompositions for various problem sizes. The original version of the Pochoir stencil compiler is intolerable to errors. Scaling the problem size to more than 18K results in almost none correct runs. By contrast, the DARE version of Pochoir tolerates more errors and has a smooth degradation as the problem size grows. Notably, the overhead in execution time for the DARE decomposition is less than 4%, averaged across grid sizes.

5.5 Conclusions

The work presented in this Chapter exposed and systematically exploited refresh-by-access as another key property of application resilience to enable aggressive relaxation of DRAM refresh. We proposed DARE, a novel non-intrusive method that facilitates refresh-by-access to reduce the number of manifested errors under

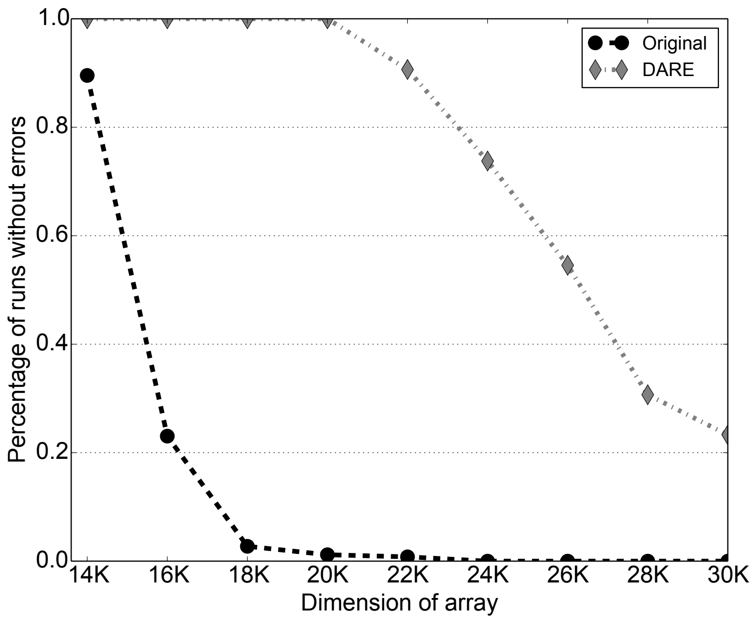


Fig. 5.9 Comparison of DARE Pochoir versus the original

aggressively relaxed or even completely turned-off refresh. We realised DARE on a complete system stack of an off-the-shelf server to capture for the first time the time-dependent system and data interactions, which is infeasible on existing simulators. The developed system stack is a key instrument to compare and combine multiple techniques for DRAM refresh relaxation, including DARE and CADA techniques. Experimentation results, for a variety of applications with different resilience characteristics, show that is possible to eliminate refresh completely while avoiding avoid catastrophic failures and having acceptable output quality because of our DARE techniques. Such benefits come with imperceptible performance overhead and with minor output quality degradation that ranges from 2% to 18%, which is, in all cases, much less than CADA-only schemes.

Chapter 6

Conclusions and Future Work

In this thesis, we have studied approximate computing from the perspective of the system design. We strongly believe that, for approximate computing to become a useful system and programming paradigm, all layers of the computing system need to be aware and cooperate in the context of approximation, whether this is materialized through hardware unreliability or is software defined. As a result, our study extends from the application-level, to the system software and the hardware. We started by investigating algorithmic properties for iterative solvers that enable the application of approximate computing to this class of algorithms and a methodology that allow us to evaluate its efficacy in terms of power consumption and performance. Next, we focused on the design and implementation of an approximate computing programming model which builds on top of pre-existing models for parallel execution and enhances them with semantics for approximate computing. Finally, we focused on memory unreliability and the design of a full system stack which enables the adoption of unreliable memory within a robust computing environment, which utilizes programmer domain-specific knowledge from the application field and the programming model which developed for approximate computing. Additionally, we explore the potential to minimize at the system software level the adverse effects of memory unreliability to the application in a transparent fashion.

6.1 Summary of Contributions

6.1.1 Evaluating Approximate Computing Using iso-metrics

We began our work investigating whether approximate computing can prove to be a way to overcome the power wall. A large body of previous research has adopted that claim without, in reality, considering the effects of unreliability to applications. In order to keep those effects in control, several techniques (e.g., checkpoint and restart, increased number of iterations to convergence, in the case of iterative solvers, etc.) need to be applied. These techniques often incur performance penalties that we need to consider when designing approximate computing systems. In Chapter 3, we proposed a method based on iso-metrics to evaluate the real efficiency of such systems showing case studies for three algorithms where approximation is indeed a viable method to reduce energy consumption.

Our methodology provides a systematic way to evaluate the performance and energy benefits of applying approximate computing as a programming and system design paradigm. We consider the effects of approximation on application as well as the hardware. Approximation on iterative solvers can potentially increase the execution time of the application by slowing down the convergence of the algorithm. At the same time, operating a hardware platform at NTV significantly reduces its power consumption, however at the same time deteriorates its performance. The performance penalty induced from NTV operation is expected to be recovered through parallelism. As a result we consider the NTV configuration of our platform that achieves the same performance (iso-performance), or power (iso-power) as the nominal configuration and evaluate the potential benefits in terms of power consumption, performance and Energy-to-Solution for the evaluated benchmarks.

It is important to mention that our methodology neglects overheads related to migrating from a single-socket ARM platform operating at nominal voltage to a multi-socket NTV ARM platform. Such a transition will incur additional communication overheads that our approach fails to capture. However, our methodology can easily be adapted to reflect the changes once we have access to a real platform that will allow us to measure these overheads. In that sense our current setup provides a best-case scenario estimation for approximating the benchmarks of our evaluation in the platform we experimented with. We expect that applications which can take advantage of parallelism the additional cost will be minimal, whereas applications which are harder to parallelize e.g., applications with irregular access patterns and inter-thread communication will benefit less.

Finally, we focused on the use-case of iterative solvers, however our methodology does not restrict itself to such computations. We can apply our method to any application-domain if we have an estimation of the cost of approximation, e.g., cost of checkpoint and restart recovery time, etc.

6.1.2 Programming Models for Approximate Computing

The second contribution of this thesis is the design and implementation of a programming model and runtime system for approximate computing. The goal of this endeavor is to give the mechanisms to programmers to define what part of the code and data is suitable for approximation and manage the degree of approximation. Our intuition is that what can be approximated is tightly coupled with the algorithm and, hence, should be delegated to the developer. Our programming model provides these mechanisms to the developers, while at the same time under the hood the runtime system implements approximation, respecting the requirements, e.g. power/energy consumption, quality of result, of users as defined at runtime. As a result, we introduced the concept of significance for computation, which is meant to express a high-level algorithmic property that expresses the contribution of a computation to the quality of the final result. The more significant a computation is, errors or approximations during its execution will lead to higher quality degradation.

For example, in Chapter 4 we show that using domain-specific knowledge about the application we approximate, we can achieve quality results that degrade smoothly when the level of approximation increases. In contrast, system level techniques, such as loop perforation, which disregard such knowledge can hurt quality abruptly, even when mild approximation is applied.

The main conclusion of this part of the thesis is the importance of approximation-related knowledge for the application. For example, the loop perforation technique is not to be disregarded as an approximation technique. Rather, it should be coupled with domain-specific knowledge regarding the application it is being applied. However, in order to allow programmers to focus in the challenging task of understanding and harnessing approximation for their applications, it is important to have available the programming tools necessary to express approximation, rather than using ad-hoc techniques. Our programming model is an effort in doing that for task-parallel applications.

6.1.3 Memory Reliability and System Level Management of Approximate Computing

In the final part of the thesis we deal with memory approximation and a full-stack design and implementation of an approximate platform, based on an off-the-shelf hardware platform. We study memory approximation that originates from relaxing the refresh rate of DRAM memory chips. We profile the behavior of DRAM chips when operating with relaxed refresh rate, i.e., and temporal distribution of errors of the DRAM, as well as the behavior of applications when they are executed on top of unreliable memory. In order to test applications, we expanded our task-based parallel programming model with semantics for applying significance properties to memory allocation.

Of course, a system with unreliable memory is not useful unless we avoid errors occurring in critical system data, e.g., kernel data structures. As a result, we worked and implemented a design which a) ensures the non-disruptive execution of the platform, avoiding errors to crash the system b) configure the reliability of the memory and c) expose to the user-level an API which selectively allows programmers to allocate unreliable memory when this is suitable for their applications. Having this platform in place, we were able to extend our programming model to use the memory allocation API and experiment with a number of computational kernels. This helped us to get insight about the behavior of an application when executing on top of unreliable memory, using an off-the-shelf server. As a result, we were able to identify a system-wide DRAM property, *Refresh-by-access*, i.e., applications refresh DRAM data solely by accessing them.

We used this property to build a system-level scheduling technique, DARE, which is able to reduce the amount of DRAM errors an application experiences. Despite the fact that this is a system-level approximation technique, DARE is an optimization technique, since it does require input from the application designer, since it cannot shield the applications from errors happening in critical data structures.

6.2 Future Work

In conclusion, we believe that approximate computing, being in its infancy, requires a significant amount of research to be conducted towards establishing it as a robust and reliable computing paradigm. In our opinion, the future work should focus on two broad areas: a) algorithm-focused research to identify application properties that make the application a suitable target of approximation and b) research around the system-level issues of approximate computing to design systems that are robust to approximation and platforms that allow expressing approximation in a systematic rather than ad-hoc manner, through programming models and APIs that expose approximation-related information from the hardware to the application developer.

In our experience, research on the former region is rather restricted, particularly due to the lack of significant research of the latter area. Algorithm designers are reluctant to invest time in yet another design point during the development cycle of their applications. This is largely due to the lack of support for approximation from mainstream programming models and frameworks. Researchers need to apply ad-hoc tricks to develop and test approximate applications, which is, naturally, counter-intuitive. Moreover, the unavailability of approximate hardware or system software that supports such hardware is an additional roadblock.

In this thesis, we took a step forward towards designing systematic ways of expressing and applying approximate computing, and we hope that more research in the same direction will emerge in the future.

References

[1] Ahmad A Al-Yamani, Nahmsuk Oh, and Edward J McCluskey. Performance evaluation of checksum-based abft. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 461–466. IEEE, 2001.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, pages 265–276. Springer, 2013.

[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, pages 198–209, New York, NY, USA, 2010. ACM.

[5] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[6] Nikolaos Bellas, Sek M Chai, Malcolm Dwyer, and Dan Linzmeier. Real-time fisheye lens distortion correction using automatically generated streaming accelerators. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 149–156. IEEE, 2009.

[7] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[8] Ishwar Bhati, Mu Tien Chang, Zeshan Chishti, Shih Lien Lu, and Bruce Jacob. DRAM Refresh Mechanisms, Penalties, and Trade-Offs. *IEEE Transactions on Computers*, 65(1):108–121, jan 2016.

[9] Ishwar Bhati, Zeshan Chishti, Shih-Lien Lu, Bruce Jacob, Ishwar Bhati, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. Flexible auto-refresh. *ACM SIGARCH Computer Architecture News*, 43(3):235–246, jun 2015.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[11] Charalampos Chalios, Dimitrios S Nikolopoulos, and Enrique S Quintana-Ortí. Evaluating asymmetric multicore systems-on-chip using iso-metrics. *arXiv preprint arXiv:1503.08104*, 2015.

[12] Gregory Chen, Dennis Sylvester, David Blaauw, and Trevor Mudge. Yield-driven near-threshold sram design. *IEEE transactions on very large scale integration (VLSI) systems*, 18(11):1590–1598, 2010.

[13] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, volume 48, pages 167–176. ACM, 2013.

[14] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 113:1–113:9, New York, NY, USA, 2013. ACM.

[15] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, apr 2012.

[16] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[17] Zehan Cui, Sally A. McKee, Zhongbin Zha, Yungang Bao, and Mingyu Chen. DTail. In *Proceedings of the 28th ACM international conference on Supercomputing - ICS '14*, pages 43–52, New York, New York, USA, 2014. ACM Press.

[18] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[19] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[20] J Elliot, F Müller, Miroslav Stoyanov, and Clayton Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Tech. Rep. ORNL/TM-2013/282, Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, 2013. 6, 9, 2013.

[21] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for

large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[22] M. Ghosh and H. H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 134–145, Dec 2007.

[23] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor Mudge, and David Blaauw. Exploring dram organizations for energy-efficient and resilient exascale memories. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 23:1–23:12, New York, NY, USA, 2013. ACM.

[24] Philipp Gschwandtner, Charalampos Chalios, DimitriosS. Nikolopoulos, Hans Vandierendonck, and Thomas Fahringer. On the potential of significance-driven execution for energy-aware hpc. *Computer Science - Research and Development*, pages 1–10, 2014.

[25] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, January 2012.

[26] Yinhe Han, Ying Wang, Huawei Li, and Xiaowei Li. Data-aware dram refresh to squeeze the margin of retention time in hybrid memory cube. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 295–300, Piscataway, NJ, USA, 2014. IEEE Press.

[27] J. L. Hennessy and D. A. Patterson. *Computer architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[28] Mark Hoemmen and Michael Heroux. Fault-tolerant iterative methods via selective reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, volume 3, page 9, 2011.

[29] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[30] Joshua Hursey, Jeffrey Squyres, Timothy Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[31] Intel. Intel xeon processor e5-1600/2400/2600/4600 (e5-product family) product families datasheet, volume two. 2012.

[32] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B Part 2*, Jun 2013.

[33] JEDEC. DDR3 sdram specification. 2010.

[34] JEDEC. DDR4 sdram specification. 2013.

[35] J. E. Jones. On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 106(738):441–462, 1924.

[36] Matthias Jung, Deepak M. Mathew, Christian Weis, and Norbert Wehn. Efficient reliability management in SoCs - an approximate DRAM perspective. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 390–394. IEEE, jan 2016.

[37] Matthias Jung, Deepak M. Mathew, Christian Weis, and Norbert Wehn. Invited - approximate computing with partially unreliable dynamic random access memory - approximate dram. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 100:1–100:4, New York, NY, USA, 2016. ACM.

[38] Georgios Karakonstantis, Nilanjan Banerjee, and Kaushik Roy. Process-Variation Resilient and Voltage-Scalable DCT Architecture for Robust Low-Power Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(10):1461–1470, oct 2010.

[39] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[40] Ulya Karpuzcu, Nam Sung Kim, and Josep Torrellas. Coping with parametric variation at near-threshold voltages. *Micro, IEEE*, 33(4):6–14, July 2013.

[41] Ulya R Karpuzcu, Abhishek Sinkar, Nam Sung Kim, and Josep Torrellas. Energysmart: Toward energy-efficient manycores for near-threshold computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 542–553. IEEE, 2013.

[42] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design—opportunities and challenges. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1149–1154. IEEE, 2012.

[43] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134. IEEE, 2008.

[44] Kinam Kinam Kim and Jooyoung Jooyoung Lee. A New Investigation of Data Retention Time in Truly Nanoscaled DRAMs. *IEEE Electron Device Letters*, 30(8):846–848, aug 2009.

[45] L Leem, H Cho, J Bau, Q Jacobson, and S Mitra. ERSA: Error-resilient system architecture for probabilistic applications. In *IEEE/ACM Design Automation and Test in Europe*, pages 1560–1565. IEEE, mar 2010.

[46] Chung-Hsiang Lin, De-Yu Shen, Yi-Jung Chen, Chia-Lin Yang, and Cheng-Yuan Michael Wang. SECRET. *ACM Transactions on Architecture and Code Optimization*, 12(2):19:1–19:24, jun 2015.

[47] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, Onur Mutlu, Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern DRAM devices. In *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, volume 41, page 60, New York, New York, USA, 2013. ACM Press.

[48] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-aware intelligent dram refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[49] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker. *ACM SIGPLAN Notices*, 46(3):213, mar 2011.

[50] Yongpan Liu, Huazhong Yang, Robert P Dick, Hui Wang, and Li Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*, pages 204–209. IEEE, 2007.

[51] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[52] Robert Lucas, J Ang, K Bergman, S Borkar, W Carlson, L Carrington, G Chiu, R Colwell, W Dally, J Dongarra, et al. Top ten exascale research challenges. *DOE ASCAC Subcommittee Report*, 2014.

[53] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[54] Dejan Markovic, Cheng C Wang, Louis P Alarcon, Tsung-Te Liu, and Jan M Rabaey. Ultralow-power design in near-threshold region. *Proceedings of the IEEE*, 98(2):237–252, 2010.

[55] Paul Mineiro. fastapprox. http://code.google.com/p/fastapprox/, 2012.

[56] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embed. Comput. Syst.*, 12(2s):88:1–88:26, May 2013.

[57] 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*, pages 195–200. ACM, 2009.

[58] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, José F. Martínez, Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, volume 41, page 48, New York, New York, USA, 2013. ACM Press.

[59] Prashant J. Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. Refresh pausing in DRAM memory systems. *ACM Transactions on Architecture and Code Optimization*, 11(1):1–26, feb 2014.

[60] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 72–83, New York, NY, USA, 2013. ACM.

[61] Amir Moosavie Nia and Karim Mohammadi. A generalized abft technique using a fault tolerant neural network. *Journal of Circuits, Systems, and Computers*, 16(03):337–356, 2007.

[62] Koji Nii, M Yabuuchi, Y Tsukamoto, S Ohbayashi, Y Oda, K Usui, T Kawamura, N Tsuboi, T Iwasaki, K Hashimoto, et al. A 45-nm single-port and dual-port sram family with robust read/write stabilizing circuitry under dvfs environment. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 212–213. IEEE, 2008.

[63] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 4.0). Technical report, July 2013.

[64] Nathaniel Pinckney, Korey Sewell, Ronald G Dreslinski, David Fick, Trevor Mudge, Dennis Sylvester, and David Blaauw. Assessing the performance limits of parallelized near-threshold computing. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1143–1148. IEEE, 2012.

[65] Moinuddin K. Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J. Nair, and Onur Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 427–437, Washington, DC, USA, 2015. IEEE Computer Society.

[66] Arnab Raha, Hrishikesh Jayakumar, Soubhagya Sutar, and Vijay Raghunathan. Quality-aware data allocation in approximate DRAM? In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 89–98. IEEE, oct 2015.

[67] Abbas Rahimi, Andrea Marongiu, Paolo Burgio, Rajesh K. Gupta, and Luca Benini. Variation-tolerant openmp tasking on tightly-coupled processor clusters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 541–546, San Jose, CA, USA, 2013. EDA Consortium.

[68] Abbas Rahimi, Andrea Marongiu, Rajesh K. Gupta, and Luca Benini. 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, pages 35:1–35:10, Piscataway, NJ, USA, 2013. IEEE Press.

[69] Barry Rountree, Dong H Ahn, Bronis R de Supinski, David K Lowenthal, and Martin Schulz. Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 947–953. IEEE, 2012.

[70] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 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, pages 95–104, New York, NY, USA, 2014. ACM.

[71] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[72] Giacinto P Saggese, Nicholas J Wang, Zbigniew T Kalbarczyk, Sanjay J Patel, and Ravishankar K Iyer. An experimental study of soft errors in microprocessors. *IEEE micro*, 25(6):30–39, 2005.

[73] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 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, pages 35–50, New York, NY, USA, 2014. ACM.

[74] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[75] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 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, pages 164–174, New York, NY, USA, 2011. ACM.

[76] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, page 4. ACM, 2013.

[77] Florian Schmoll, Andreas Heinig, Peter Marwedel, and Michael Engel. Improving the fault resilience of an h.264 decoder using static analysis methods. *ACM Trans. Embed. Comput. Syst.*, 13(1s):31:1–31:27, December 2013.

[78] Mingoo Seok, Gregory Chen, Scott Hanson, Michael Wieckowski, David Blaauw, and Dennis Sylvester. Cas-fest 2010: Mitigating variability in near-

threshold computing. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(1):42–49, 2011.

[79] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[80] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, September 2001.

[81] Joseph Sloan, John Sartori, and Rakesh Kumar. On software design for stochastic processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 918–923, New York, NY, USA, 2012. ACM.

[82] Shuaiwen Song, Chun-Yi Su, Rong Ge, Abhinav Vishnu, and Kirk W Cameron. Iso-energy-efficiency: An approach to power-constrained parallel computation. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 128–139. IEEE, 2011.

[83] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. Accessed: 2017-01-01.

[84] Richard M Swanson and James D Meindl. Ion-implanted complementary mos transistors in low-voltage circuits. *IEEE Journal of Solid-State Circuits*, 7(2):146–153, 1972.

[85] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11*, page 117, 2011.

[86] Matthew Tolentino and Kirk W Cameron. The optimist, the pessimist, and the global race to exascale in 20 megawatts. *Computer*, 45(1):95–97, 2012.

[87] 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*, pages 207–216. IEEE, September 2010.

[88] George Tzenakis, Angelos Papatriantafyllou, Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for task-based parallelism. In *Advanced Parallel Processing Technologies*, pages 17–31, 2013.

[89] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Trans. Math. Softw.*, 34(4):18:1–18:36, July 2008.

[90] Manolis Vavalis and George Sarailidis. Hybrid-numerical-PDE-solvers: Hybrid Elliptic PDE Solvers. http://dx.doi.org/10.5281/zenodo.11691, Sep 2014.

[91] R.K. Venkatesan, S. Herr, and E. Rotenberg. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 157–167. IEEE, 2006.

[92] Gregor Von Laszewski, Lizhe Wang, Andrew J Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[93] Lizhe Wang, Gregor Von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE Computer Society, 2010.

[94] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[95] Foivos S. Zakkak, Dimitrios Chasapis, Polyvios Pratikakis, Angelos Bilas, and Dimitrios S. Nikolopoulos. Inference and declaration of independence: Impact on deterministic task parallelism. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 453–454, New York, NY, USA, 2012. ACM.

[96] Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu. Approxit: An approximate computing framework for iterative methods. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 97:1–97:6, New York, NY, USA, 2014. ACM.

[97] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 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, pages 441–454, New York, NY, USA, 2012. ACM.