Scheduling and Performance Characterization on Heterogeneous Computing Systems

Giorgis Georgakoudis

A thesis presented for the degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering
University of Thessaly
Greece
May 2016

Dissertation Committee:
Associate Prof. Spyros Lalis
Prof. Dimitrios S. Nikolopoulos
Assistant Prof. Christos Antonopoulos

I would like to dedicate this thesis first to my family and then to my true teachers

Acknowledgements

This thesis includes research work conducted while working in two institutions. These are the Department of Electrical and Computer Engineering of University of Thessaly in Greece, and the School of Electronics, Electrical Engineering and Computer Science of Queen's University of Belfast in United Kingdom. I have only good memories, warm feelings and gratitude for the universities I have been working in. I have also to acknowledge the funding agencies which made this research possible through financial means. These include the European Commission through EU projects and the Engineering, Physics and Research Council (EPSRC) through UK national projects.

First and foremost I would like to thank my mentors and supervisors. I would like to express my gratitude to Professor Spyros Lalis from the University of Thessaly who trusted in me at the beginning of my research career. His guidance during my initial steps and throughout my PhD molded me as a researcher. Also, I would like to especially thank Professor Dimitrios Nikolopoulos in Queen's University of Belfast who was instrumental in enabling this research. Without his guidance and mentoring none of this work would be possible. I would like to express my gratitude to Dr. Hans Vandierendonck from Queen's University of Belfast too. He was always available to discuss and provide constructive criticism. His criticism helped tremendously in presenting effectively this work.

I would like also to acknowledge and thank my past and present colleagues who provided me with help and enjoyable moments to ease the burden of research. Particularly, I would like to thank Dr. Dimitris Syrivelis, Manos Koutsoubelias and Dr. Nikos Tziritas with whom I shared my early years as a researcher. Also, from my current colleagues I would like to especially thank Charalambos Chalios. Sharing research and technical experience and having stimulating, and often heated, discussions, was very educating and relaxing.

Last but certainly not least, I owe great many thanks to my family. Especially I owe to my parents, Thanasis and Kaiti, for their unconditional love and support all along my academic pursuits. Also, I would like to express my gratitude and appreciation to my soulmate Konstantia Georgouli, who I consider part of my family. Konstantia has been always by my side during good times and bad times to support me with patience and love. The least I can do is dedicate this thesis to them.

Abstract

Scheduling and Performance Characterization on Heterogeneous Computing Systems

by
Giorgis Georgakoudis

Doctor of Philosophy, Graduate Program in Electrical and Computer Engineering,
University of Thessaly, Greece.
May 2016
Prof. Spyros Lalis, Chairperson

The era of performance and power efficiency scaling through transistor shrinking and frequency scaling, characterized by Dennard's law, has come to an end. Nevertheless, transistor doubling still withstanding, computing architectures have moved to the multicore and manycore era for improving performance through parallelism. However, this scaling of computing architectures exposed power consumption to be an important limiting factor and a key concern to sustain further growth. Heterogeneous computing architectures are the next evolution step and promise to improve the performance, power and cost efficiency of systems. They promise to do so through hardware specialization, by including components which exhibit different performance and power consumption characteristics. The potential for improved performance and power efficiency from heterogeneous architectures has rendered them ubiquitous in every aspect of computing, from mobile devices such as smartphones and tablets to large-scale computing installations such as supercomputers and datacenters. However, the efficient allocation of heterogeneous resources to match workload requirements with their capabilities, thus achieve their performance and power efficiency potential, is an open and challenging problem.

In this dissertation we contribute novel solutions, techniques and insight on efficient resource allocation for heterogeneous systems. At the level of heterogeneous processor architectures, we present new techniques for managing instruction-based heterogeneity on

state-of-the-art, shared-ISA architectures. These include binary adaptation techniques to enable software portability and thread migration across heterogeneous cores despite their binary incompatibilities. Following, we present a novel, heterogeneous-aware scheduler for maximizing the speedup of a workload through dynamic re-allocation of heterogeneous cores between co-executing programs. The evaluation of our methods on a hardware prototype platform shows that the dynamic scheduler we propose outperforms significantly previous state-of-the-art solutions. Moreover, we scope resource allocation for performance and energy efficiency on heterogeneous datacenter architectures. We present new methodologies and define iso-comparison metrics to fairly and accurately characterize the performance and energy efficiency across diverse, heterogeneous server platforms in the context of online data analytic services. We evaluate several heterogeneous types of servers, including standard servers, micro-servers and manycore accelerators. Our methods provide new insight by assessing resource allocations in heterogeneous datacenter architectures under the prism of meeting Quality-of-Service requirements of online data analytics while characterizing their energy efficiency. The application of our methodologies can effectively guide efficient heterogeneous resource allocation and provisioning in the datacenter.

Περίληψη

Χρονοπρογραμματισμός και Χαρακτηρισμός Απόδοσης Σε Ετερογενή Συστήματα Υπολογιστών

από τον
Γιώργη Γεωργακούδη

Υποψήφιος Διδάκτωρ, Μεταπτυχιακό Πρόγραμμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ
Πανεπιστήμιο Θεσσαλίας, Ελλάδα.
Μάιος 2016
Καθ. Σπύρος Λάλης, Επικεφαλής επιβλέπων

Η εποχή της αύξησης των υπολογιστικών επιδόσεων και της ταυτόχρονης μείωσης της κατανάλωσης ισχύος μέσω της σμίκρυνσης των ημιαγωγών και της αύξησης της συχνότητας λειτουργίας (νόμος του Dennard) έχει παρέλθει. Παρ΄ όλ΄ αυτά, ο αριθμός των διαθέσιμων ημιαγωγών συνεχίζει να διπλασιάζεται. Αυτός είναι ο λόγος που η αρχιτεκτονική υπολογιστών έχει μετακινηθεί στη σχεδίαση πολυπύρηνων επεξεργαστών για την αύξηση των υπολογιστικών επιδόσεων μέσω της παράλληλης επεξεργασίας. Όμως, αυτή η κλιμάκωση στον αριθμό των επεξεργαστών έφερε στην επιφάνεια ένα νέο πρόβλημα, αυτό της κατανάλωσης ισχύος, που πλέον είναι σημαντικός περιοριστικός παράγοντας στην περαιτέρω κλιμάκωση των συστημάτων. Το επόμενη βήμα στην εξέλιξη της αρχιτεκτονικής συστημάτων είναι η σχεδίαση ετερογενών υπολογιστικών συστημάτων που μπορούν να βελτιώσουν σημαντικά τόσο τις υπολογιστικές επιδόσεις, την κατανάλωση ισχύος και τελικά την ενεργειακή αποδοτικότητα των συστημάτων. Λόγω αυτών των χαρακτηριστικών, τα ετερογενή συστήματα βρίσκονται σε κάθε έκφανση της υπολογιστικής, από φορητές συσκευές μέχρι εγκαταστάσεις υπερ-υπολογιστών και κέντρων δεδομένων. Για να επιτύχουν αυτά τα οφέλη, οι ετερογενείς αρχιτεκτονικές ακολουθούν την αρχή της εξειδίκευσης υλικού (hardware specialization), περιλαμβάνοντας συστήματα με διαφορετικές επιδόσεις και κατανάλωση ισχύος για την εκτέλεση προγραμμάτων. Ωστόσο, η

ποσοτικοποίηση για τον χαρακτηρισμό της απόδοσης των ετερογενών πόρων και ο χρονοπρογραμματισμός τους για την αποδοτική εκτέλεση προγραμμάτων παραμένει ανοιχτό πρόβλημα.

Σε αυτή τη διατριβή, συνεισφέρουμε νέες λύσεις, μεθόδους και τεχνικές στοχεύοντας στο πρόβλημα της απόδοσης και του χρονοπρογραμματισμού ετερογενών αρχιτεκτονικών. Συγκεκριμένα, στο επίπεδο των ετερογενών αρχιτεκτονικών επεξεργαστών, μελετούμε τις αρχιτεκτονικές διαμοιραζόμενου συνόλου εντολών (shared-ISA architectures) ως τεχνολογία αιχμής, και παρουσιάζουμε νέες τεχνικές για τη διαχείριση της ετερογένειας στο επίπεδο εντολών. Σ΄ αυτές περιλαμβάνονται τεχνικές μετεγγραφής δυαδικού κώδικα που επιτρέπουν τη φορητότητα δυαδικού κώδικα αλλά πιο σημαντικά επιτρέπουν τη μεταφορά νημάτων εκτέλεσης μεταξύ ετερογενών, επεξεργαστικών πυρήνων παρά τις ασυμβατότητες στο σύνολο εντολών. Επιπρόσθετα, παρουσιάζουμε μια καινούρια μέθοδο δυναμικού χρονοπρογραμματισμού που επιταχύνει την ταυτόχρονη εκτέλεση πολλαπλών προγραμμάτων, αντιλαμβάνοντας την επιτάχυνση λόγω της ετερογένειας για κάθε πρόγραμμα για να ανακατανείμει τους ετερογενείς πυρήνες μέσω της μεταφοράς νημάτων εκτέλεσης. Η αξιολόγηση των μεθόδων μας σε μια πλατφόρμα υλικού που υλοποιεί μια αρχιτεκτονική διαμοιραζόμενου συνόλου εντολών δείχνει ότι υπερτερούν σημαντικά σε επιδόσεις από τις υπάρχουσες προτεινόμενες τεχνικές. Επίσης, μελετούμε το πρόβλημα της διαχείρισης υπολογιστικών πόρων σε ετερογενείς αρχιτεκτονικές κέντρων δεδομένων. Παρουσιάζουμε μια νέα μεθοδολογία βασισμένη σε ισο-μετρικές που επιτρέπει τη σύγκριση ετερογενών εξυπηρετητών, όσον αφορά την υπολογιστική και ενεργειακή απόδοση τους, σε περιβάλλοντα εκτέλεσης online υπηρεσιών ανάλυσης δεδομένων. Μέσω αυτής της μεθοδολογίας, ποσοτικοποιούμε την απόδοση διάφορων τύπων εξυπηρετητών που περιλαμβάνουν τυπικούς εξυπηρετητές, μικρο-εξυπηρετητές και πλατφόρμες επιταχυντών πολλαπλών πυρήνων. Οι μέθοδοί μας εξετάζουν την ποσοτικοποίηση της ενεργειακής απόδοσης θέτοντας καθορισμένους στόχους στην αποδεκτή ποιότητα υπηρεσίας ως κοινό μέτρο της επίδοσης ετερογενών εξυπηρετητών. Η εφαρμογή της μεθοδολογίας μας μπορεί αποτελεσματικά να καθοδηγήσει τη διαμοιρασμό ετερογενών πόρων και τη προδιαγραφή ενεργειακά αποδοτικών κέντρων δεδομένων.

Related publications

[1] Giorgis Georgakoudis, Charles J. Gillan, Ahmed Sayed, Ivor Spence, Richard Faloon, and Dimitrios S. Nikolopoulos. Methods and Metrics for Fair Server Assessment Under Real-time Financial Workloads. *Concurrency and Computation: Practice and Experience*, 28(3):916–928, 2016. cpe.3704.

[2] Giorgis Georgakoudis, Charles Gillan, Ahmed Sayed, Ivor Spence, Richard Faloon, and Dimitrios S. Nikolopoulos. Iso-quality of Service: Fairly Ranking Servers for Real-time Data Analytics. *Parallel Processing Letters*, 25(03):1541004, 2015.

[3] C. J. Gillan, D. S. Nikolopoulos, G. Georgakoudis, R. Faloon, G. Tzenakis, and I. Spence. On the Viability of Microservers for Financial Analytics. In *High Performance Computational Finance (WHPCF), 2014 Seventh Workshop on*, pages 29–36, Nov 2014.

[4] G. Georgakoudis, D. S. Nikolopoulos, H. Vandierendonck, and S. Lalis. Fast Dynamic Binary Rewriting for Flexible Thread Migration on Shared-ISA Heterogeneous MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 156–163, July 2014.

[5] G. Georgakoudis, D. S. Nikolopoulos, and S. Lalis. Fast Dynamic Binary Rewriting to Support Thread Migration in Shared-ISA Asymmetric Multicores. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*, COSMIC '13, pages 1–10, New York, NY, USA, 2013. ACM – **Best paper award**.

[6] G. Georgakoudis, S. Lalis, and D. S. Nikolopoulos. Dynamic Binary Rewriting and Migration for Shared-ISA Asymmetric, Multicore Processors: Summary. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 127–128, New York, NY, USA, 2012. ACM.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computing in recent years observes a fundamental change from the the past. The sustained increase in computing performance through clock scaling as a result of transistor shrinking is not possible any more. This effect has been characterized as the *power wall* [81] and marks the end of exponential growth of computing performance, characterized empirically by Dennard scaling and Moore's law. Although clock frequency does not scale any more, transistor counts still increase, shifting the focus on multicore and manycore architectures, for increasing raw performance through hardware parallelism. Core counts are roughly doubling every two years (Figure 1.1) and multicore architectures are the mainstream in every aspect of computing, from embedded systems to supercomputers. Moreover, the power wall effect and the proliferation of battery-limited portable computing revealed the importance of power consumption as a limiting factor and as an optimization opportunity. Power and energy consumption are alongside performance important metrics when evaluating a system. The paradigm shift from unicore to multicore and the rising importance of power consumption raise new challenges in both hardware and software design to enable sustainable performance of future computing systems.

Initial designs of multicore architectures typically include replicas of the same core that share the memory address space through an interconnection network. This design is referred to as a *symmetric multiprocessor system* (SMP), where each core presents an identical interface to the software. Spending the transistor real estate on identical processing elements favors simplicity but it is hardly the optimal choice regarding performance and power consumption. Recent research in computing architecture [2, 7, 9, 19, 37, 39, 48, 59, 61, 63, 64, 91] shows that *heterogeneous multiprocessing* (HMP), which includes architecturally

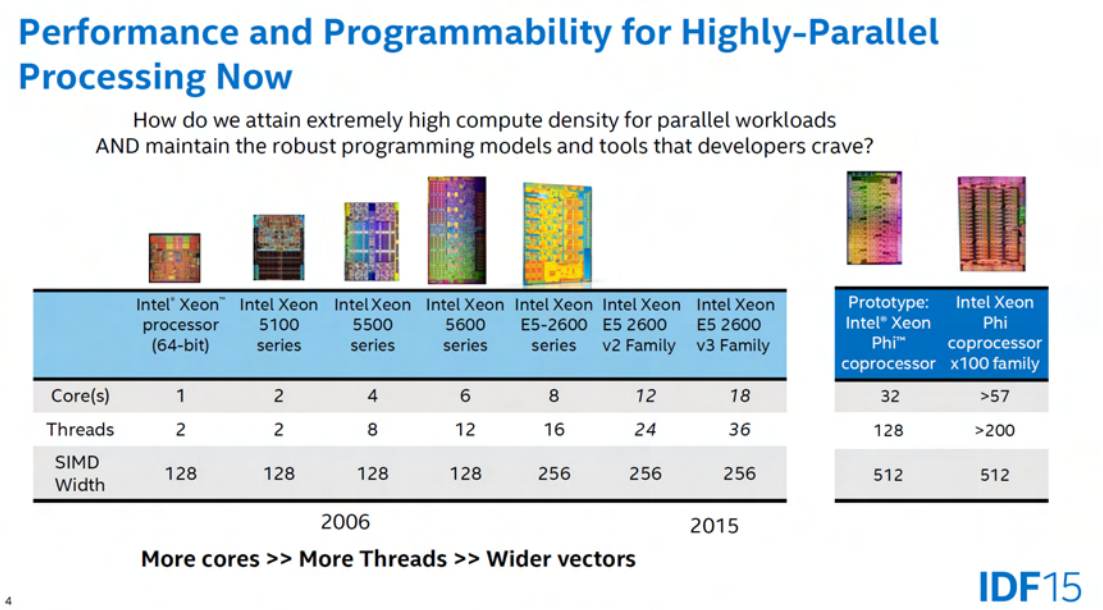


Figure 1.1 The evolution of Intel processors (Intel Developer Forum 2015)

different cores (or other processing elements such as DSPs, FPGAs, GPUs and others), that may not be software compatible, provides multiple advantages over SMP approaches:

- Increased performance, through customized, special-purpose hardware acceleration
- Better power efficiency, by including cores with different power-performance design points
- Larger core counts for increased hardware parallelism, by combining larger (in terms of area), more complex cores with smaller, simpler ones

These improved performance and power efficiency advantages of heterogeneous architectures have rendered them ubiquitous in all kinds of computing, from mobile devices, such as smartphones and tablets, to large-scale datacenters and supercomputers. However, the problem of how to efficiently expose heterogeneity to programs and how to map heterogeneous resources to execution for performance and energy-efficiency is challenging.

Existing state-of-the-art heterogeneous multicores are divided into two categories in relation to the interface they present to software: single-ISA, where all cores implements the same ISA, for example the ARM big.LITTLE [38] architecture; and disjoint-ISA, where cores implement different ISAs, such as CPU+GPU platforms, for example AMD Fusion APUs [15] and NVIDIA Tegra [85] MPSoCs, and the STHORM platform [78]. Single-ISA heterogeneous architectures preserve software compatibility, however, this limits heterogeneity only to the micro-architecture and precludes any functional acceleration possible

through ISA extensions. Disjoint-ISA architectures have high acceleration potential but are hard to program. Developers need to cope with new software tools and programming models [1, 66, 88, 107] but most importantly, they need to have an intimate understanding of the functional heterogeneity among cores. For bridging this gap between software compatibility and acceleration potential, *shared-ISA* architectures [33, 64] have been proposed recently. In these architectures, all cores implement a common, basic set of instructions – the basic ISA – which preserves software compatibility among them. At the same time, some cores include functional extensions to accelerate performance-critical operations. These are accessible to software through ISA extensions. A program compiled to target any accelerating instruction can run only on the subset of cores which support ISA extensions but this may cause load imbalance hurting performance. Compiling a program to target the basic ISA means it can run on any core, without ISA-related core affinities, but it must forgo acceleration. Managing programmability for shared-ISA heterogeneity presents two challenges. First, it should be transparent to the programmer to avoid replicating the programmability issues of disjoint-ISA architectures and second, it must avoid setting ISA-related program-to-core affinities that may degrade performance.

Contribution: In this thesis we present new techniques for transparently executing programs on shared-ISA architectures, regardless of the ISA targeted during compilation, and without setting any core affinities. It employs a fast, dynamic binary adaptation method which employs lightweight instruction rewriting and translation techniques. Our novel techniques re-target binary code for the host core ISA with minimal overhead by leveraging the partial binary compatibility offered by the common, basic ISA of shared-ISA architectures. Dynamic binary adaptation is implemented as an OS service, invoked on-demand by the system scheduler. Most importantly, our method does not impose any restrictions on scheduling, thus enabling the implementation of optimized, dynamic scheduling policies for heterogeneous aware resource allocation which we discuss next.

Prior work [5, 10, 32, 54, 55, 60, 62, 96, 97, 97, 98, 100, 104, 106, 111] on optimizing the performance and power efficiency of execution on heterogeneous architectures, regardless of their ISA interface, has shown that the mapping of heterogeneous resources to programs is crucial. Specifically, heterogeneous resources should be allocated to those programs which make the most of their acceleration potential. In disjoint-ISA systems, the programmer is burdened to map heterogeneous resources explicitly to programs for optimizing execution. In single-ISA systems, heterogeneity and the associated acceleration is ISA-neutral, and for optimizing the mapping of programs to heterogeneous resources it is sufficient to observe program-level characteristics, such as compute and memory intensiveness or sequential and parallel execution phases. By contrast, heterogeneity of shared-ISA architectures is exposed

at the instruction level, by the inclusion of accelerating ISA extensions on certain cores. Identifying and quantifying acceleration potential of programs at the instruction level is an open problem. Moreover, the fact that acceleration depends on extended instruction execution renders the dynamic mapping of programs to heterogeneous cores necessary, dependent on the control flow within programs.

Contribution: In this thesis we present a novel, dynamic scheduling solution for shared-ISA architectures with the purpose of maximizing the speedup of co-executing programs. Our solution combines a new, cross-core profiling technique to quantify program speedup from accelerating instructions at runtime, and a heuristic search algorithm which exploits this information to find program-to-core mappings that maximize workload speedup. Notably, the scheduler leverages dynamic binary adaptation to perform on-demand, cross-core migrations for mapping dynamically programs to cores. Next we discuss on resource allocation for performance and energy efficiency at a larger scale, in the context of heterogeneous datacenters.

Datacenters are essential infrastructures for providing large-scale data analytic services. However, they carry an immensely high total cost of ownership and energy footprint. The unsustainable growth in power consumption of datacenters [22, 35, 94] poses a major challenge. Moreover, datacenters are notoriously wasteful, often using as little as 10% of the supplied power for actual data storage and processing, while exhibiting less than 20% node utilization [43, 58]. Leveraging a setup of heterogeneous servers in datacenter architectures is key to improve their utilization and energy efficiency through resource allocation for computation and energy proportionality [27, 28, 69, 95, 108]. In fact, datacenters are inherently heterogeneous, because during their lifetime they include different generations of servers [43, 75, 76]. Moreover, recent research has shown that the addition of micro-servers [14, 90] and accelerators [21, 65] to the datacenter ecosystem has significant potential to further improve performance and energy efficiency. Resource allocation in a diverse, heterogeneous datacenter ecosystem for meeting Quality of Service (QoS) goals set by online data analytics while promoting energy efficiency is a difficult problem. In particular, quantifying the performance and energy efficiency across heterogeneous servers calls for new methodologies to draw insight driven by application requirements.

Contribution: In this thesis, we present a rigorous methodology and define new metrics for assessing the performance and energy efficiency across heterogeneous servers. Our methodology enables resource provisioning and allocation on heterogeneous datacenters for energy efficiency while meeting service-driven performance targets of online data analytics. Central to our methodology are iso-comparisons using iso-QoS and iso-energy metrics to accurately and fairly quantify both performance and energy efficiency across heterogeneous

servers, despite their architectural differences. Characterization of servers using our iso-comparison method enables dynamic resource allocators to leverage heterogeneous servers for computation and energy proportional execution of workloads.

1.2 Synopsis

This thesis presents novel methodologies, tools and mechanisms to quantify and optimize the efficiency of execution on emerging heterogeneous systems.

In Chapter 2 we present an overview of heterogeneous architectures both in the context of processor multicores, including the state-of-the-art shared-ISA ones, and heterogeneous datacenters. We discuss open problems and optimization opportunities from heterogeneity and describe the hardware and software platforms used for experimentation and evaluation.

Chapter 3 describes our novel solution to address transparent execution despite binary incompatibility on shared-ISA heterogeneous architectures. We present a dynamic binary adaptation method which enables any binary to execute on any core in the system, without needing programmer involvement or re-compilation. Moreover, our dynamic binary adaptation makes possible dynamic, cross-core migrations at any point in time, to overcome the load imbalance issues of existing, state-of-the-art fault-and-migrate techniques. Evaluating our dynamic binary methods shows that they are indeed fast and have minimal overhead that can be readily recovered through informed dynamic scheduling.

Chapter 4 introduces a novel scheduling framework that we developed for optimizing the performance of co-executing programs on shared-ISA architectures. The framework implements a new profiling method at runtime for assessing a thread's acceleration potential and predicting its cross-core speedup. We present a novel heuristic algorithm for scheduling which maximizes workload speedup by periodically adjusting thread-to-core assignments, using cross-core migrations. Results comparing our scheduler to the state-of-the-art scheduling based on fault-and-migrate scheduling show that it results in significantly lower execution times across multi-program workloads and different configurations of the hardware platform.

Chapter 5 discusses heterogeneous datacenters and presents the problem of fairly assessing performance and energy efficiency across heterogeneous servers. We present a new iso-comparison methodology using new metrics in the context of online analytics services to evaluate a heterogeneous setup using micro-servers, standard Intel Xeon servers and Intel Xeon Phi manycore platforms. Moreover, we apply our methodology to thoroughly investigate platform specific optimizations, such as vectorization, and alternate hardware configurations impacting performance and power consumption by varying resource allocation.

Chapter 6 concludes the thesis, presenting a summary of the contributions and main findings, as well as future work directions.

Chapter 2

Background on Heterogeneous Architectures

2.1 Disjoint-ISA Architectures

Disjoint-ISA architectures have high performance potential by combining cores with vastly different design characteristics for acceleration and power efficiency. Examples of these architectures include CPU+GPU platforms [25, 83], such as Nvidia Tegra family MPSoCs and AMD Fusion APUs, the Cell BE [40], the STHORM platform [78] and other proposals from the research community [17, 18, 29, 42]. The flexibility to not conform to a single ISA allows those designs to provide a high potential for both performance and power optimization at the cost of programmability.

Developers need to learn new tools and programming model to cope with binary incompatibility and architectural intricacies. Most importantly, developers need to have an intimate and deep understanding of the functional heterogeneity and low-level details of the architectures involved. These programmability issues render disjoint-ISA architectures hard to program and make difficult to extract their full acceleration potential. Besides programmability issues, mapping efficiently computation on those architectures is challenging. It is left to the developer to map computation parts to processing elements and again this requires deep knowledge of the different architectures to maximize acceleration or energy efficiency or a combination of both.

2.2 Single-ISA architectures

Single-ISA architectures have a uniform ISA, thus all cores are binary compatible and executables can run unmodified on any core in the system. However, this uniform ISA restriction limits heterogeneity in traits that must be transparent to software, in the micro-architecture of cores. State-of-the-art research has proposed numerous types of single-ISA heterogeneity architectures: combining in-order and out-of-order cores [106], different clock scaling domains [8, 60, 62] and lately there is the industrial proposition of ARM's big.LITTLE architecture [38].

In these proposed single-ISA architectures, there are two types of cores: big (as in area), high clocked, power hungry cores and small, low clocked, power efficient ones. We name the former collectively as *fast* cores to emphasize performance while the latter are deemed as *slow* cores. Moreover, these designs target to have few fast cores but numerous slow cores to comply with a fixed power or area budget. The rationale is to sacrifice single-core performance for a larger core count, nonetheless considering that spending extra resources to increase single-core performance has diminishing returns. In summary, such a design exhibits the following characteristics:

- fast cores are fewer, power hungry but offer better single thread performance,
- slow cores are numerous and power efficient, hence they boost parallelism

Optimizing execution on these architectures is based on the concept of *efficiency specialization* [97, 98], as in running a thread to the core it executes most efficiently. Efficiency can be defined in terms of performance, power or energy consumption or a combination of these metrics. In this context of single-ISA designs, efficiency specialization translates into discovering compute-intensive or memory-intensive phases during execution and whether there is thread-level parallelism [10, 32, 57, 62, 70, 73, 96, 98–100, 106, 111]. Specifically, compute-intensive execution phases run more efficiently on fast cores. This is because they can reap the performance benefits of the extra hardware resources of these cores, justifying the increased power consumption. By contrast, memory bound phases execute more efficiently on slow cores since they have limited computational needs. Executing them on slow cores is sufficient for their performance needs, while saving energy at the same time. Regarding thread-level parallelism, sequential phases should execute on fast cores, to avoid the bottleneck, while parallel phases benefit from executing on the multitude of slow cores to make use of hardware parallelism. Overall, while single-ISA heterogeneous multicores provide more customization possibilities than homogeneous architectures, they have limited acceleration and power reduction potential. The fact that heterogeneity is restricted to ISA

transparent features precludes any functional heterogeneity which has high acceleration potential.

2.3 Shared-ISA Architectures

Recently proposed shared-ISA (also known as overlapping-ISA) architectures [33, 64] bridge the dichotomy between disjoint-ISA and single-ISA multicores. In shared-ISA architectures all cores provide a common, basic ISA, which preserves binary compatibility for programmability purposes. At the same time, certain cores provide functional specialization, which is visible to software through ISA extensions. Software can exploit the acceleration potential of these functionally enhanced cores by specializing code to include these accelerating extension instructions.

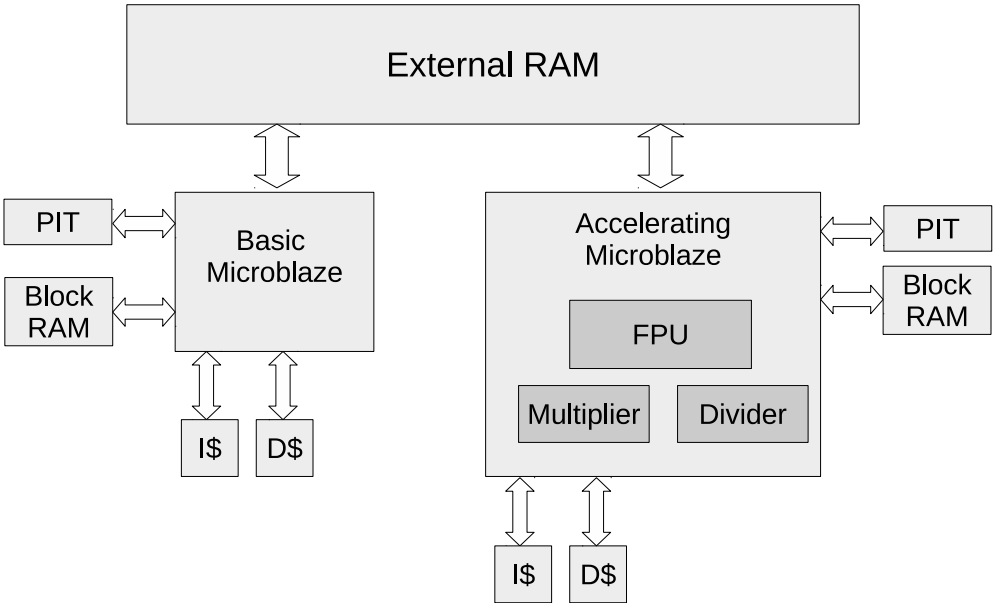
Shared-ISA architectures attempt to combine programmability with the performance potential of functional heterogeneity. The basic ISA typically includes control flow instructions, memory operations and simple arithmetic instructions which are the bulk of common operations across binaries. Selected, functionally enhanced cores implement ISA extensions, which accelerate performance-critical operations, such as floating point computations, cryptographic extensions and SIMD operations. However, if a binary is compiled to target any accelerating instruction, it is not compatible to those cores which lack acceleration support. On the other hand, binaries compiled to target the basic ISA can execute on any core but without acceleration. This partial, instruction-level heterogeneity combined with the high acceleration potential of functional heterogeneity presents unique challenges and possibilities from the efficient use of shared-ISA architectures.

2.3.1 Experiment Platform

Hardware Prototype

Previously proposed approaches [33, 64] implement ISA heterogeneity by disabling hardware accelerating components, specifically the FPU of selected cores. We follow a similar approach by creating a shared-ISA multicore architecture on real hardware, an FPGA prototype that consists of ISA heterogeneous RISC Microblaze cores. This platform choice is motivated by the FPGA's extensibility for architectural exploration and its ability for hardware reconfiguration. Each Microblaze core implements a 32-bit RISC ISA which is extensible with accelerating instructions, by optionally instantiating additional hardware units. In this prototype functional extensions include FPUs and fast integer multiplier/divider

Figure 2.1 Overview of the shared-ISA FPGA prototype



units for accelerating numerical computations. Table 2.1 lists the accelerating instructions pertaining to those units, along with their cycle latency for fully pipelined execution.

Other than ISA heterogeneity, cores have the same micro-architectural characteristics: 100 MHz clock frequency, single issue in-order 5-stage pipeline, separate 64KB L1 instruction and data caches, a 512-entry branch target cache for branch prediction, and a dedicated connection to the external memory controller. Figure 2.1 shows a schematic overview of the shared-ISA multicore architecture of our platform, including a basic version of the Microblaze and an accelerating one.

Software Platform

The hardware runs the Xilkernel operating system [110] with extensions that we introduced to enable multi-processing support. The Xilkernel OS is an open-source operating system developed by Xilinx that is partly POSIX compliant, including a POSIX thread interface for thread creation and management. The platform has no hardware support for cache coherence but our multi-processing extensions enforce cache consistency only for shared OS data, through software-controlled cache invalidations. At the application level, cache consistency should be enforced in the application itself, through explicit calls to cache flushing and invalidation routines whenever there are shared data.

Table 2.1 Possible accelerating instructions

Instruction	Function	Hardware unit	Clock cycles
mul	INT multiplication	Integer Multiplier	1
idiv	signed INT division	Integer Divider	32
idivu	unsigned INT division	Integer Divider	32
fadd	FP addition	FPU	4
frsub	FP subtraction	FPU	4
fmul	FP multiplication	FPU	4
fdiv	FP division	FPU	4
fcmp	FP comparison	FPU	4

2.4 Heterogeneous Datacenter Architectures

Interestingly, datacenter installations are inherently heterogeneous as they are provisioned over a 15-year period [43], during which servers are gradually installed and replaced. This means that servers of different architecture generations co-exist in datacenters. Exploiting this inherent heterogeneity for resource allocation has shown [75, 76] to improve utilization and energy efficiency on datacenters. However, standard servers are by design high-power machines and there is need to revisit the datacenter architecture [58] to sustain the growth of the energy consumption.

Micro-servers have been recently proposed [14, 52, 53, 71, 90, 103] as a low-power alternative to standard servers. For reducing their power consumption, micro-servers follow design methodologies of embedded systems, replacing typical server-grade processors with low-power SoCs, similar to the those found on smartphones and tablets. Moreover, they are designed for a small form factor to increase compute density by bundling together multiple compute nodes in a rack blade.

Adding more to the heterogeneity, accelerators, including FPGAs, GPUs and Xeon Phi coprocessors, are deployed within datacenters. Besides using accelerators as offload coprocessors of a specific node, recent research work [50, 89, 109] has shown that breaking away from this model and instead allowing multiple nodes to access those accelerators as distinct computing elements improves utilization and power efficiency through sharing and load distribution.

Heterogeneous datacenter architectures promise sustained growth by including servers or computational elements with different performance and power efficiency design points. These architectures improve energy efficiency and performance of installations through

hardware specialization [23, 27, 41] and efficient resource allocation to workloads. That is, allocate those heterogeneous resources which are the most energy-efficient and at the same time achieve the computational and QoS requirements of applications and services. However, quantifying the performance and energy efficiency across heterogeneous servers is necessary to guide resource allocation for specialization. As our experiment platforms to evaluate heterogeneity in the datacenter, we make use of standard Intel Xeon SandyBridge servers, ARM-based microservers and Xeon Phi coprocessors as a manycore platform.

Chapter 3

Dynamic Binary Adaptation on Shared-ISA Architectures

Shared-ISA multicore architectures expose their functional heterogeneity to software via accelerating instructions. A program binary compiled to target the basic ISA can execute on any core but must forgo acceleration. Conversely, binaries targeted for the accelerating ISA can execute only on accelerating cores and this may lead to over-subscription degrading performance. Existing state-of-the-art approaches addressing this issue are universal binaries and the fault-and-migrate technique. However, both may affect adversely performance by setting core affinities to cope with binary incompatibility although they can be sub-optimal.

In this chapter, we present fast dynamic binary adaptation techniques which can re-target any binary to the basic or accelerating ISA. Most importantly, it avoids limitations existing in state-of-the-art techniques and enables on demand, cross-core migration that can be leveraged from dynamic scheduling policies. We evaluate our techniques on the real hardware, shared-ISA prototype (discussed in Chapter 2) and show that they have little overhead which is readily recoverable using a proof-of-concept dynamic scheduler.

3.1 Introduction

On shared-ISA architectures all cores provide a common, basic ISA, which preserves binary compatibility and improves programmability. At the same time, selected cores provide functional specialization, which is visible to software through ISA extensions. Software can exploit the acceleration potential from these acceleration enhanced (ACC) cores, through code specialization. However, instruction-based asymmetry renders execution non-transparent at the binary level. Binaries must forgo acceleration and implement only the basic ISA to

execute on all cores for full system utilization. If binaries include accelerating instructions they can execute only on ACC cores. Transparent execution is crucial for shared-ISA systems by avoiding programmability issues but at the same time make use of ISA acceleration when possible.

The state-of-the-art technique for transparent execution on shared-ISA systems is *fault-and-migrate* (FAM) [64, 91]. It assumes binaries implement the ACC ISA but a thread can execute in any core in the system. If a thread executes an accelerating instruction on a non-ACC core, FAM forcibly migrates it to an ACC core. FAM moves back a thread to its originator core after a *migrate-back* timeout expires, to alleviate congestion on ACC cores. This approach has severe limitations: (i) FAM forced migration sets dynamically core affinities on a per-thread basis, precluding any other dynamic scheduling from implementing global optimization policies, (ii) it may over-subscribe ACC cores with threads and limit acceleration of those threads due to time-sharing the ACC core, (iii) FAM may suffer from an excessive number of unavoidable thread migrations and the associated loss of performance. Most importantly, FAM forced migration prohibits any dynamic scheduling approach to improve performance or power consumption through efficient utilization of heterogeneous resources.

In this chapter we present two different techniques to support transparent execution with acceleration while enabling flexible, cross-core migration in shared-ISA systems. Through our techniques, a binary can execute on any core in the system, using accelerating instructions when possible. Also, any thread can migrate to any core type, at any point in time allowing dynamic schedulers to leverage flexible migration for implementing global optimizing policies. Specifically our contributions are:

- A lightweight Dynamic Binary Rewriting (DBR) method which rewrites binary code implemented in the basic ISA depending on the host core accelerating instruction extensions. DBR dynamically discovers code execution paths and replaces basic instructions with accelerating instructions when executing on an ACC core or reverses previous code changes to execute on a Basic core.

- A fast Dynamic Binary Translation (DBT) technique which implements *fault-and-rewrite*: when an accelerating instruction execution faults, DBT sets up a trampoline to an emulation routine translating this instruction to the basic ISA. DBT reverts trampoline jumps to the original accelerating instruction for native execution on an ACC core.

- We evaluate our techniques against FAM on the real hardware prototype of a shared-ISA MPSoC, discussed on Chapter 2, using single-program workloads from the SPEC

CPU2006 [45] and Rodinia [20] suites. We measure the performance of DBR and DBT by triggering migration periodically between a Basic and an ACC core. Periodic migration under DBR has an average slowdown of about 40% while DBT average slowdown is around 10% compared to FAM opportunistic acceleration scheduling. This slowdown can be readily recovered through informed dynamic scheduling instead of arbitrary periodic migration.

- We also show results for a *S*peedup *P*roportional *D*ynamic *S*cheduler (SPDS) enabled by flexible migration provided from our binary-level techniques. SPDS dynamically migrates threads between Basic and ACC cores to accelerate threads in proportion to their speedup. We compare it against FAM scheduling by running heterogeneous, multi-program workloads on a hexa-core MPSoC consisting of 2 ACC and 4 Basic cores. SPDS, employing our binary-level techniques, out-performs FAM scheduling by as much as 50%.

The rest of this chapter is organized as follows: Section 3.2 briefly discusses fault-and-migrate and other transparent executions schemes. Section 3.3 presents our dynamic binary adaptation methods. Section 3.4 presents the evaluation of our work, including the experiment methodology and implementation details. Section 3.5 concludes the chapter.

3.2 Previous State-of-the-art

3.2.1 Universal Binaries

Universal (also known as “fat”) binaries bundle together different versions of binary images for every ISA available on a heterogeneous multicore. At deployment time, the scheduler assigns a core (usually the least loaded one) to execute the binary and selects the binary image compatible with this core. However, universal binaries by design lack support for thread migration: code deployed on a specific ISA must execute only on cores with the same ISA. Cross-architectural migration between distinct ISA cores requires dynamic transformation of the binary code itself and the runtime state, such as the call stack. Such migrations, however, can be too costly or even infeasible, if there is not enough state information to resume a thread on a core with a different ISA [29].

3.2.2 Fault-and-migrate

Fault-and-migrate (FAM) [64, 92] has been proposed as a method for transparent execution with opportunistic acceleration on shared-ISA architectures. FAM assumes code is statically

compiled to target the ACC ISA for performance. However, if a thread executes an ACC instruction on a Basic core, FAM forcibly migrates this thread to an ACC core. FAM induced migrations can cause ACC cores to become over-subscribed and system performance may suffer from load imbalance. Furthermore, the scheduler is bound by forced migrations and has no option to select which thread(s) to accelerate using ACC instructions, when such a choice would affect application or overall system performance. Contrary to FAM, our binary adaptation techniques transform a binary to execute transparently on any core in the system, without forcing migration or any other disruption in scheduling. This enables the scheduler to implement dynamic scheduling policies through on-demand, cross-core thread migrations as is the case for scheduling policies proposed in later chapters. We have extensively compared our work with FAM to show that flexible migration, provided by our binary-level techniques, enables dynamic scheduling which out-performs FAM's forced migration scheduling.

3.2.3 Dynamic Binary Rewriters

Existing dynamic binary rewriters, such as DynamoRIO [16] and Pin [72], execute a binary in managed mode. They need to manage software code caches to enable code profiling, security checking and complex optimizations. This approach permits elaborate code analysis and fine-grain binary instrumentation but comes with significant overhead. Instead, DBR and DBT are designed to be lightweight and fast, aiming for as much unobstructed execution as possible to be efficient. Also, traditional rewriters assume that code is immovable, which allows them to do time consuming optimizations, whereas our binary-level techniques target flexible, cross-core migration.

3.2.4 Discussion

The fundamental difference of our binary-level adaptation techniques compared to these methods is that they enable flexible migration which can be leveraged by dynamic schedulers to implement global optimizing policies. By contrast, FAM enforces thread migrations by the necessity to execute accelerating instructions only on ACC cores. FAM implicitly assumes that forced migration will opportunistically accelerate execution to offset the migration cost. However, this approach disrupts load balancing by oversubscribing ACC cores and hinders any global dynamic scheduling due to forcing core affinities. Universal binaries on the other hand forbid thread migration and enforce static, application-level scheduling decisions that may compromise system performance during the execution of multi-program workloads.

With our binary-level techniques, a binary can execute transparently to any core in the system, without forcing migration or any other disruption in scheduling. Also DBR and DBT

differ from other frameworks that aim at binary portability, including interpreted execution, virtual machines and heavyweight binary rewriters, in that they do not necessitate continuous execution monitoring, costly binary analysis or prior code instrumentation. In particular, DBR operates on stripped binaries, and discovers live execution paths and rewriting opportunities for accelerating instructions only once, storing metadata to enable rewriting on demand. Likewise, DBT needs no binary instrumentation and incurs significantly lower overhead because it is invoked on demand and rewrites only in the scope of a single faulted instruction.

3.3 Dynamic Binary Adaptation Techniques

In this section we present two novel binary adaptation techniques, namely Dynamic Binary Rewriting (DBR) and Dynamic Binary Translation (DBT) for shared-ISA systems. Both techniques operate on stripped binaries [44, 93], that is binaries without any prior instrumentation or debugging information.

Those techniques differ on the binary input they can operate on and the adaptation methods each one employs. DBR operates on binaries statically targeted for the basic ISA. It discovers dynamically live code paths and employs peephole optimization to rewrite parts of the binary. DBR rewrites original binary code targeting basic instructions to the ACC ISA, if the executor thread sets to run on an ACC core. DBR can revert to the original basic instructions, if the thread sets to run on a Basic core.

By contrast, DBT operates on binaries targeting the ACC ISA. It employs a *fault-and-rewrite* techniques: when the execution of an ACC instruction triggers a fault on a Basic core, the instruction is replaced by a call to a translating routine, implementing the functionality of this particular ACC instruction in the basic ISA. DBT also can revert calls to translation routines back to the original instructions, when the thread sets to execute on an ACC core.

Both our techniques are designed to be lightweight. They execute on demand and strive for native execution. DBR and DBT are complementary in the way that each one operates on binaries produced on different static compilation targets. The combination of them can adapt any binary, regardless of the static compilation target ISA, to the ISA capabilities of the core which hosts the thread executing the binary code. We present and benchmark our binary adaptation techniques separately, for clarity reasons and to highlight their individual advantages and limitations.

3.3.1 Dynamic Binary Rewriting

Overview

Dynamic Binary Rewriting (DBR) adapts code compiled statically for targeting basic instructions. It specializes this code with accelerating instructions depending on the host core ISA. DBR runs as an OS service which is aware of the host core ISA type and operates transparently to application-level software. The OS bootstraps DBR operation by executing its entry routine when a new thread enters the system. This initiates binary analysis to discover live binary code for the thread. After binary code analysis, DBR rewrites accelerating instruction in the discovered code, provided the host core implements the ACC ISA. The OS invokes again DBR if a thread migrates to another core, due to scheduling activity, to adapt code to the target core ISA.

On a brief overview of DBR operation, it dynamically discovers execution flow paths by instrumenting branch instructions. Peephole analysis on the discovered basic blocks identifies possible rewriting targets. Provided the host core implements the ACC ISA, DBR patches binary code, replacing basic instructions with accelerating instructions. In case a thread migrates to a Basic core from an ACC core, the OS invokes DBR to revert any changes in the rewritten code the thread executes to the original basic instructions. DBR enables unrestricted cross-core migration at any point in time, with accelerated execution when the host core implements the ACC ISA. Moreover, it is applicable on stripped binaries without prior instrumentation or debugging information. By design, DBR is lightweight by minimizing time spent during managed execution and aiming for native execution whenever possible.

Dynamic Control Flow Discovery

The OS bootstraps DBR control flow analysis by providing the thread's start function, passed on through a thread creation *pthread_create* call. DBR dynamically discovers the execution path by instrumenting control flow instructions to discover basic blocks. Information on the discovered basic blocks is stored in a list, within the OS thread descriptor structure, accessible to later DBR invocations to avoid analyzing already discovered blocks. Next, we expand in more detail and discuss the handling of indirect branches.

During analysis, DBR disassembles the basic block to find those branch instructions which exit out of the block. The control flow analysis resolves the target address of the branch and creates a new basic block datum, inserting it in the thread list. Note that the target address of a direct branch can be resolved statically during analysis. However this is not true for indirect branches which need special handling, discussed later. The datum of the

new block discovered includes the block's starting address, which is the branch target, and the original instruction at this address. DBR replaces this instruction (the first instruction of the target basic block) with a *software break* pointing to an entry routine invoking DBR discovery and resumes native execution. If the break does execute, this entry routine saves thread context (to restore it upon resuming native execution), replaces the break with the original instruction and invokes DBR control flow analysis to repeat the basic block discovery process. Control flow analysis does not set breaks for branches targeting code inside already discovered basic blocks, since they have been already examined. By using the software break mechanism, DBR discovers only live, executing code to save analysis overhead. For example, this applies to conditional branches which may or may not be taken, pruning possibly large code paths.

Indirect branches Although the target address of direct branches can be calculated statically, indirect branch targets can be only resolved at runtime. When control flow analysis encounters an indirect branch instruction, DBR replaces the branch instruction with a software break to a special handler routine and inserts a helper basic block datum to save it. When the software break for this indirect branch executes, DBR resolves the actual target by consulting the saved thread context. Indirect branches use registers for addressing and the thread context contains the actual register state before the branch's execution. DBR proceeds as with direct branch targets, inserting a new basic block datum for the target and replacing the target address instruction with a software break for invoking DBR discovery. Indirect branch analysis is commonly a significant source of overhead for binary instrumentation frameworks [49]. For indirect branch handling, each execution of an indirect branch would need to break into DBR which may add considerable overhead to execution. However, indirect branches have good target locality [30]. Most of the time DBR resolves the same target. DBR mitigates indirect branch analysis overhead by following a *sampling approach* for processing indirect branches. Immediately after the branch target is resolved, DBR restores the original indirect branch instruction, replacing the software break instruction. At each scheduling interval, the OS invokes DBR to reset indirect branch instructions to software breaks for re-probing target addresses if those branches are still live. In other words, indirect branches are sampled once every scheduling interval. This sampling strategy allows hotspot code targeted by indirect branches to be discovered without the overhead of continuous analysis.

Binary Rewriting

At the same time with execution path discovery, DBR identifies rewriting targets using peephole analysis techniques on the disassembled code. Peephole analysis identifies rewriting

```
1 ...
2 imm 0xc000
3 brlid r15, divsi3
4 addik r5, r5, 6
5 ...
```

(a) Original code

```
...
nop
addik r5, r5, 6
idiv r3, r6, r5
...
```

(b) Patching an accelerating integer division instruction

Figure 3.1 Replacing a software emulation routine with an accelerating instruction

targets, that is instruction sequences replaceable by equivalent accelerating instructions. The rewriting targets are saved in a list within the OS thread descriptor for future reference. The rewriting list grows with more rewriting targets as new basic blocks are discovered and analyzed. In the current implementation, DBR peephole analysis identifies instruction sequences containing calls to routines that emulate ACC instructions in the basic ISA. Provided the host core implements the ACC ISA, DBR patches the binary code, replacing the rewriting targets with the accelerating instructions. Note that DBR is aware of the ABI conventions for routine calls, and uses those conventions to setup the registers operands for patching in the accelerating instructions.

If a thread migrates to a different core type, the OS invokes DBR before completing migration to adapt the code section of the thread. DBR will apply or undo patches on identified rewriting targets, depending on the target core type. If a thread running on an ACC core migrates to a basic core, DBR reads the rewriting list and undoes changes in the binary of the thread's code section. Conversely, if a thread migrates from a basic core to an ACC core, DBR patches the code with accelerating instructions for the discovered rewriting targets.

Rewriting happens in-place, on the application binary code section itself. DBR may need to remove or reorder instructions, and may amend the machine state because of rewriting. Figure 3.1 shows an indicative example. A routine call implementing integer division using basic instructions can be replaced with an accelerating *idiv* instruction in the ACC ISA. The call is implemented as a delay-slotted, branch-and-link instruction, preceded by an *imm* instruction for extending the branch operand address. DBR writes a *nop* in place of the unused *imm* instruction. Furthermore, it swaps places between the patched, accelerating *idiv* instruction and the instruction in the delay slot to preserve execution order.

Following the same example, DBR needs to update the thread's machine state due to rewriting. Let a thread execute the rewritten code snippet in Figure 3.1(b) on an ACC core. If the scheduler sets this thread to migrate to a basic core, DBR has to undo the accelerating

instruction patch and possibly amend the *context-switch resume register* (CSRR). The CSRR points to the application code address saved before entering to the OS context. This is the address the thread will resume execution at the user-level context, after migration. If CSRR points to the instruction at line (3), DBR will move the CSRR to the preceding instruction at line (2) to correctly resume the restored routine call. In a more challenging case, if CSRR points to the previously patched accelerating instruction at line (4), the originally delay-slotted instruction has been already executed. Thus it is not possible for DBR to backtrack the CSRR in the original code to ensure correct execution after migration. DBR resolves this case by executing the accelerating instruction in its own context, updating the thread machine context as if the instruction has executed normally. CSRR is set to point to the subsequent instruction at which the thread resumes after migration completes. DBR detects such cases and performs any necessary amendments.

3.3.2 Dynamic Binary Translation

Dynamic Binary Translation (DBT) enables transparent execution and flexible migration, operating on binary code which has been statically compiled for the ACC ISA. DBT is implemented as an OS service too, being transparent to the application layer. DBT replaces ACC instructions with equivalent, lightweight emulation routines, or vice versa. DBT is triggered either when a thread executes an ACC instruction on a Basic core through a fault handling mechanism, or by a scheduling decision to migrate a thread from a basic core to an ACC core, for policy reasons. In the former case, DBT replaces the faulted ACC instruction with a trampoline to an equivalent emulation routine implemented with basic instructions. In the latter case, DBT restores any trampolines set previously to the original ACC instructions for accelerated execution on the ACC core.

DBT Operation

In this section we discuss in detail the internal operation of DBT. Note that ACC instructions execute at full speed, without DBT intervening, when a thread executes on an ACC core. However, DBT needs to adapt the binary targeting the ACC ISA when a thread executes the code section an a basic core. Figure 3.2 depicts DBT operation triggered by an instruction fault. In more detail, when a thread executes an ACC instruction on a basic core, an illegal instruction exception triggers the OS exception handler which has been extended to invoke the DBT management routine. The DBT management routine follows a series of steps to handle the fault and adapt the binary. Firstly, DBT sets up an *emulation handler* for the faulted ACC instruction which translates this instruction to an equivalent implementation in

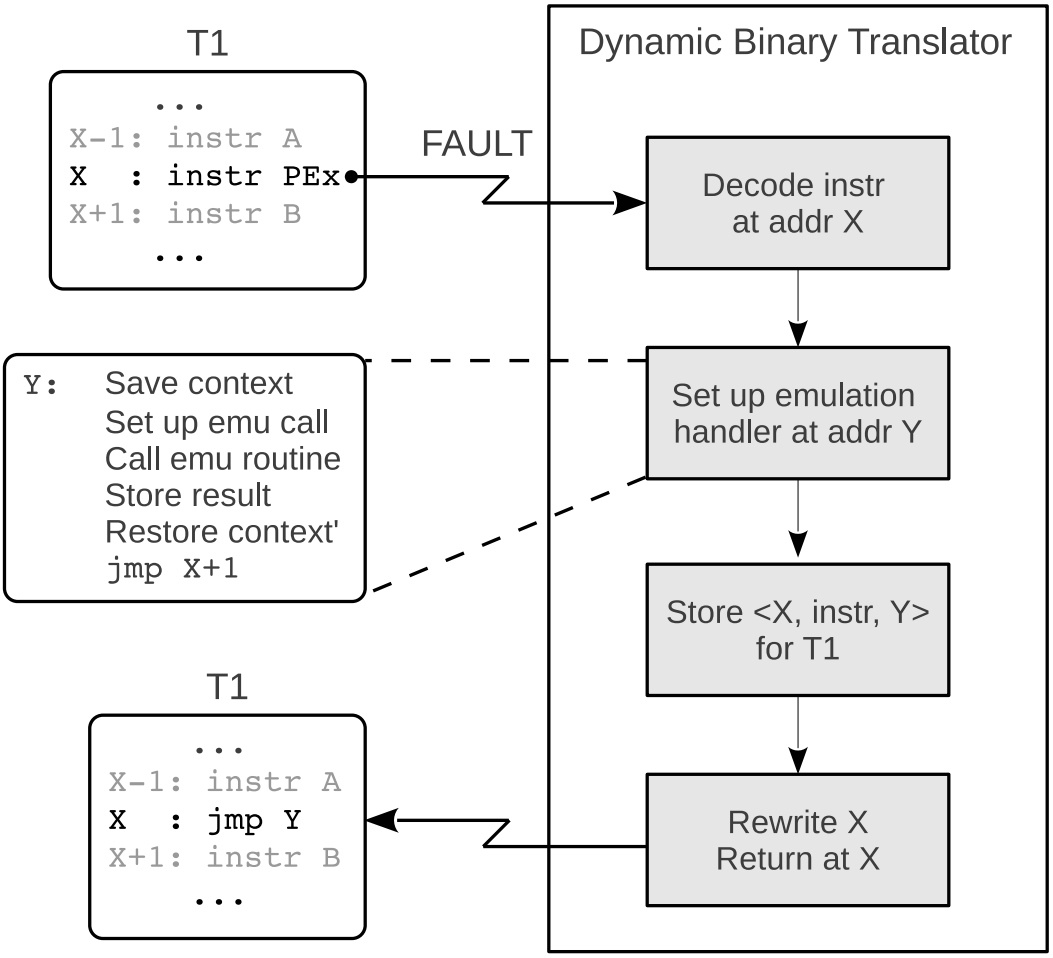


Figure 3.2 ACC instruction fault triggers DBT to set up the trampoline to the emulation routine

the basic ISA. Secondly, it patches the faulting address storing the ACC instruction with a trampoline jump to the emulation handler. Specifically, the emulation handler code performs the following sequence of operations:

(1) saves part of the thread execution context to avoid altering it when emulating

(2) sets up the call to the emulation routine, loading registers with input

(3) calls the emulation routine which is functionally equivalent to the ACC instruction

(4) stores back emulation results, saving them to output registers

(5) restores the unmodified execution context and finally,

(6) jumps back to the instruction following the trampoline branch.

Finally, after setting the emulation handling, DBT switches control back to the application to resume at the faulted address where the trampoline has been written. Thread execution resumes and jumps for the first time to the emulation handler code. Subsequent executions of the rewritten instruction address take the trampoline jump to emulation, without DBT intervening.

Moreover, DBT stores information on the emulation handlers and faulted instructions In more details, DBT stores a data triplet containing the faulted address, the binary encoding of the faulted instruction and the emulation handler's address. This triplet is inserted in a per-thread rewriting list, saved within the OS thread descriptor. If a thread, which has undergone DBT rewriting previously, migrates to an ACC core, the OS invokes DBT which reads the thread's rewriting list and replaces trampoline jumps with the original ACC instructions for accelerated execution after migration.

3.4 Evaluation and Results

3.4.1 Implementation Details

We extend the OS to implement DBR and DBT as system-level services transparent to the application level. The OS bootstraps DBR with a thread's start routine to perform dynamic binary analysis and instruction patching. On thread migration to a different ISA core, the OS invokes DBR to specialize code for the target core. DBR patches accelerating instructions if the target core implements the ACC ISA, or reverts previous patches, restoring basic instructions if the target is a basic core. DBR stores a total of 36 bytes of metadata per

rewriting target, including the target's rewriting type, the target instruction address, the instruction itself, a flag for delay-slotted execution and other data structure variables. Also, DBR stores 68 bytes per basic-block discovered, including the instruction at the block start address, the block's start and end addresses and other data for fast searching in the basic block list.

Regarding DBT, the OS invokes it when a thread triggers an illegal instruction fault by executing an unsupported accelerating instruction on a basic core. Also, the OS invokes DBT again in case a thread migrates from a basic core to an ACC core, to remove any patched trampoline jumps and restore the original accelerating instructions. DBT stores emulation handlers in core private, low-latency scratchpad memories which are 32KB in size in our implementation. Scratchpad memories function as emulation handling buffers with fast, cache-like access times (1 cycle) and are addressable with a single branch instruction from the trampoline. DBT saves rewriting management data in per-thread OS descriptors for later reference, when managing the handler buffer or reverting back rewritten instructions. Specifically, DBT stores a total of 28 bytes per rewriting target including the target instruction's address, the instruction itself, and the emulation handler's address. On migration from a basic to an ACC core, DBT flushes any thread emulation handlers resident in the local scratchpad buffer and restores previously emulated ACC instructions to execute natively at the target core.

3.4.2 Experiment Methodology

We port benchmarks from the SPEC CPU2006 [45] and Rodinia [20] suites to our platform. Table 3.1 lists the ported benchmarks. The table shows the executed instruction breakdown per-benchmark and the *end-to-end speedup*. End-to-end speedup is computed by taking the turnaround time of the benchmark executing on a basic core, using only basic instructions, versus executing it on an ACC core, accelerated by ACC instructions.

We categorize benchmarks based on their end-to-end speedup as *High*, *Medium* and *Low* speedup. High speedup benchmarks achieve more than 10× acceleration from ACC instructions. This class includes *streamcluster*, *cfd*, and *lud*, which are computational kernels from the Rodinia suite making frequent use of FP operations. Medium speedup benchmarks achieve speedup between 2× and 10×. These are: *milc*, *bfs*, *namd*, *srad*, *backprop*, *hmmer*. In those benchmarks most of their speedup comes from accelerating integer instructions and to a lesser extend from infrequent use of FP operations. Interestingly, compiling *milc* and *namd*, which are part of SPEC FP benchmarks, produces a binary which does not use hardware, single-precision FPU instructions because of double-precision FP arithmetic in the code. Nevertheless, hardware integer instructions accelerate double-precision compiled

Table 3.1 Benchmarks from SPEC CPU2006 and Rodinia suites

Benchmark	Executed instructions breakdown (% of total)									$SP = \frac{TT_{Basic}}{TT_{ACC}}$	*SP* class
	mul	idiv	idivu	fadd	frsub	fmul	fdiv	fcmp	basic		
streamcluster	0.11	-	-	4.33	4.46	4.47	-	0.13	86.50	24.27	High
cfd	0.29	-	-	4.81	0.66	5.34	0.59	0.40	87.91	14.4	High
lud	3.79	-	-	-	3.78	3.78	0.01	-	88.64	14.25	High
milc	3.53	-	-	-	-	-	-	-	96.47	4.7	Medium
bfs	2.25	-	0.53	-	-	-	-	-	97.22	3.9	Medium
namd	3.06	-	-	-	-	-	-	-	96.94	3.59	Medium
srad	1.90	-	-	0.09	0.05	0.12	0.04	0.02	97.78	3.4	Medium
backprop	0.55	-	-	0.16	-	0.17	-	-	99.12	2.63	Medium
hmmer	0.62	0.04	-	0.20	-	-	-	0.20	98.94	2.08	Medium
sjeng	0.68	-	0.05	-	-	-	-	-	99.27	1.53	Low
h264ref	0.61	0.04	0.02	-	-	-	-	-	99.33	1.37	Low
hotspot	0.15	-	-	-	-	-	-	-	99.85	1.14	Low
astar	0.08	-	-	0.02	-	0.03	-	-	99.87	1.13	Low
libquantum	0.08	-	-	0.01	-	0.02	0.01	-	99.88	1.05	Low
bzip2	0.01	-	-	-	-	-	-	-	99.99	1.01	Low

code, providing more than 3× speedup over basic instructions. Benchmarks that achieve less than 2× speedup from ACC instructions are categorized as low-speedup ones and include: *sjeng*, *h264ref*, *hotspot*, *astar*, *libquantum* and *bzip2*. Those benchmarks execute mostly basic instructions and very few accelerating ones.

Next, we present our methodology for comparing FAM again our binary adaptation techniques. We built a hardware platform consisting of an ACC and a basic core. Each benchmark runs alone, in single-threaded mode, to avoid interference from co-running programs. To evaluate FAM, the benchmark thread is initially placed on the basic core. FAM induces forced migration to the ACC core based on instruction faults and the OS moves back the thread to the basic core when the migrate-back timeout expires. In our experiments we vary the duration of the migrate-back timeout to measure its impact to execution time.

For our binary adaptation techniques we follow an induced migration approach. Since DBR and DBT enable flexible migration at any point in time migration decisions should be driven by an optimizing dynamic scheduler. Instead, to expose the overhead and limitations of our techniques, we experiment with a periodic migration approach. Specifically, the OS artificially migrates the benchmark thread between the basic and the ACC core, or vice versa, at periodic intervals, invoking DBR or DBT as needed.

The turnaround time for benchmark execution is measured in quantum scheduling intervals. The migrate-back timeout of FAM and the migration period for DBR and DBT range from 1 to 16 scheduling intervals in multiples of two. Note that in the implementation, the quantum scheduling interval is 10ms.

3.4.3 Results

In this section we discuss the results of experimentation, contrasting induced periodic migration under our binary-level techniques and FAM opportunistic acceleration. Compared to FAM, periodic migration under DBR has an average slowdown of around 40% across all benchmarks and migration periods while DBT slowdown is less, about 10%. One reason for the greater slowdown of DBR is the fact that it operates on less optimized binary code, statically targeted for basic instructions. By contrast, both FAM and DBT work with code statically targeted for the ACC ISA. The compiler targeting the ACC ISA may emit additional accelerating instructions than those identifiable by DBR on the basic binary through peephole analysis. Furthermore, the compiler may do extra, ISA-neutral, compile-time code optimizations exposed by code generation for the ACC ISA. Nevertheless, we leave extending DBR with more potent binary analysis techniques as future work.

Moreover, on analyzing the performance results, it should be noted that FAM opportunistically migrates threads to ACC cores on an instruction fault to achieve accelerated execution. In our setup, which is a single-program workload, there is no time sharing of the ACC core which would degrade performance. Instead, the benchmark thread is accelerated fully with only the penalty of instruction fault handling and migration. Note this overhead is mitigated as the migrate-back period increases. Single-program, single-threaded execution does not expose FAM's limitation of over-subscribing ACC cores.

Further, the periodic migration approach for DBR and DBT does not perform any kind of scheduling optimization and exposes their limitations. For example, under FAM, high speedup benchmarks execute mostly on the ACC core due to forced migration from instruction faulting on the basic core. Whereas in periodic migration, a benchmark thread will execute a full period on the basic core before it is migrated to the ACC core and vice versa. This is possible because DBR and DBT enable cross-core execution. We present results on a dynamic scheduler leveraging flexible migration next.

3.4.4 Speedup Proportional Dynamic Scheduling

We show results for a dynamic scheduler to illustrate the efficiency of flexible migration enabled by our binary-level techniques. The dynamic scheduler implements a *speedup*

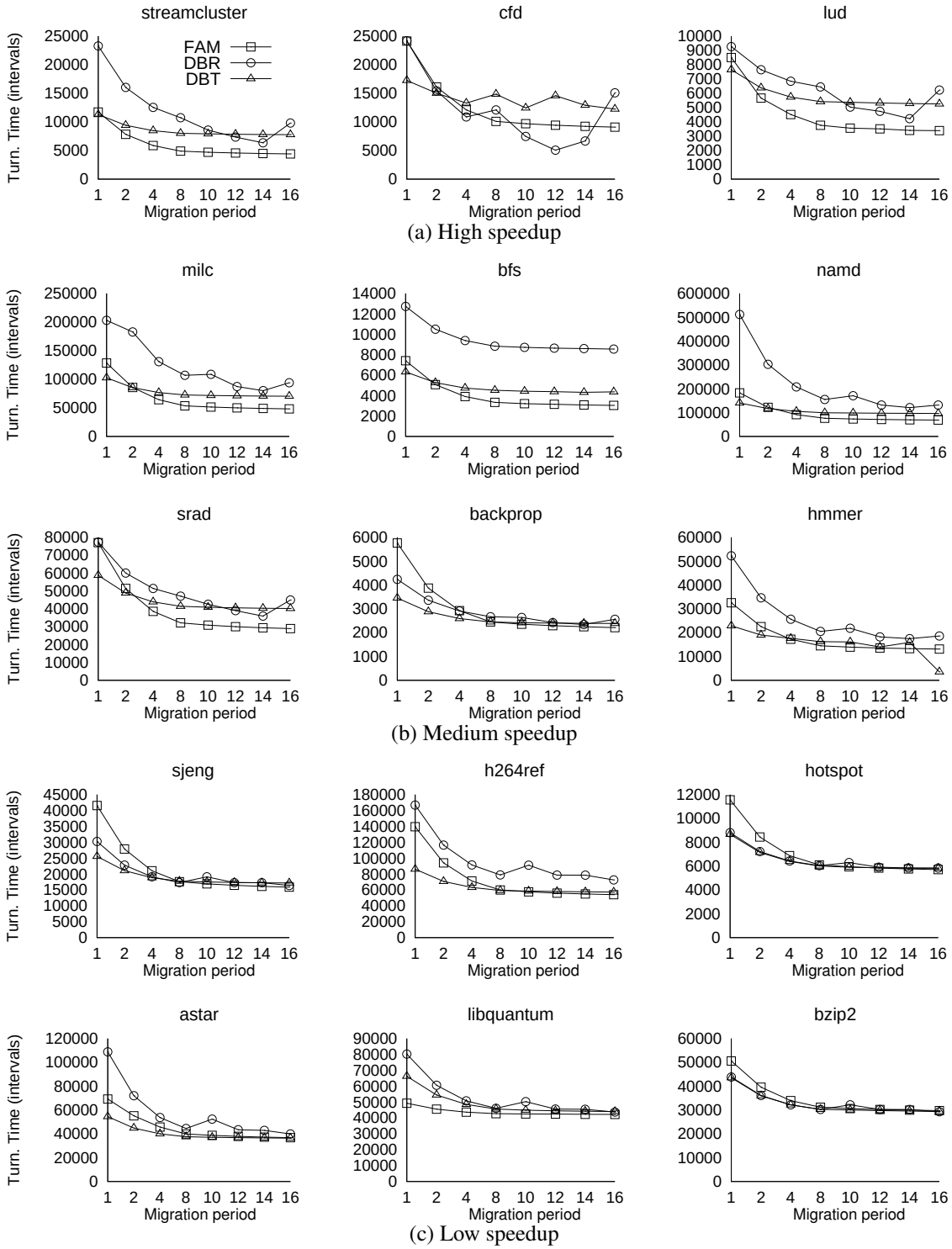


Figure 3.3 Comparing performance of DBR, DBT periodic migration vs. FAM. The horizontal axis denotes migration periods for DBR, DBT and migrate-back timeout values for FAM.

proportional policy for time sharing ACC cores between executing threads in proportion to each thread's speedup.

We briefly discuss the *S*peedup *P*roportional *D*ynamic *S*cheduler (SPDS) internals. The scheduler operates in *rounds* for time sharing ACC cores. A round is a system-wide scheduling epoch which consists of a fixed number of quantum scheduling intervals. During a round, the scheduler monitors the number of ACC intervals for each thread, i.e., the number of intervals each thread has executed on an ACC core. It employs a thread swapping mechanism, migrating threads between Basic and ACC cores, so that each thread executes on an ACC core for a number of intervals which is proportional to its speedup, compared with the speedup of the other threads in the system. At the end of a round, the scheduler resets the ACC interval counters of all threads to begin a new round. For our implementation we retrofit benchmark end-to-end speedup, shown in table 3.1, to the scheduler and set the round duration to 100 quantum scheduling intervals.

Table 3.2 Speedup heterogeneous, multi-program workloads

	Workload
3H-3L	cfd, streamcluster, lud, sjeng, bzip2, hotspot
3H-3M	cfd, lud, streamcluster, milc, backprop, hmmer
3M-3L	hmmer, srad, backprop, bzip2, sjeng, hotspot

For the evaluation we built a hexa-core MPSoC consisting of 2 ACC and 4 Basic cores. In our experiments the system is kept fully-subscribed running speedup heterogeneous, multi-program workloads. This means that multi-program workloads contain as many benchmarks as the total number of cores (irrespective of core type) and that after a benchmark completes execution, it is restarted to ensure a fully-subscribed system throughout workload execution. An experiment finishes after each benchmark has completed at least three runs. Table 3.2 shows the particular benchmarks in each multi-program workload mix. A workload is denoted by the number of benchmarks from each speedup class (High, Medium, Low). For example, the workload denoted as 3H-3M has 3 High-speedup and 3 Medium-speedup benchmarks. After the experiment is done, we compute the *average turnaround time* for each benchmark by taking the mean of turnaround times of completed runs. Per-benchmark average turnaround times are aggregated to compute SPDS speedup over FAM scheduling to define *workload speedup*. The workload speedup is the geometric mean of per-benchmark speedup values, which are calculated as the ratio of the benchmark turnaround time executing

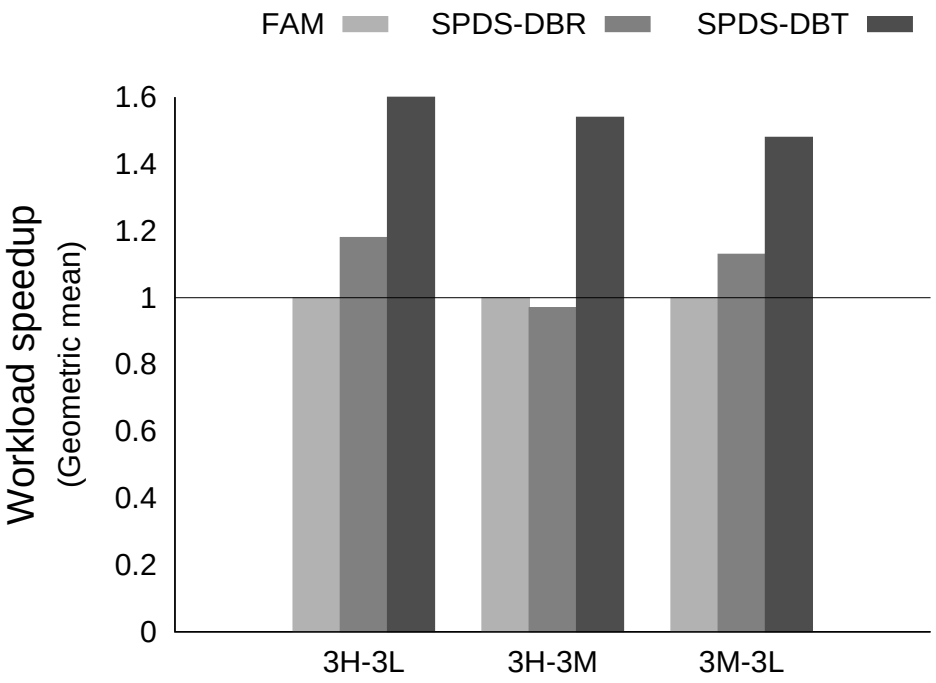


Figure 3.4 SPDS-DBR and SPDS-DBT vs. FAM for co-executing programs

under FAM over SPDS. Formally, workload speedup (WSP) is:

$$WSP_{SPDS/FAM} = \sqrt[N]{\prod_{b \in W} \frac{AvgTT_{b,FAM}}{AvgTT_{b,SPDS}}}$$

where N is the total number of benchmarks (6 in our setup) and the product is taken over all benchmarks (b) in the workload set (W).

For FAM scheduling, the (a priori known) highest speedup benchmarks threads known are placed on ACC cores at loading time to reduce the number of faults causing forced migrations. The FAM migrate-back timeout is set to 10 scheduling intervals which shows good performance based on the single-program evaluation. Initial thread placement for SPDS, either based on DBR (SPDS-DBR) or DBT (SPDS-DBT), is not important since dynamic scheduling will migrate threads for sharing ACC cores. In our implementation, SPDS-DBR and SPDS-DBT perform round balancing, that is thread swaps, every 10 scheduling intervals to be comparable with FAM's migrate-back timeout.

Figure 3.4 shows the results. SPDS-DBT performs significantly better than FAM forced migration scheduling. It improves performance by about 50% compared to FAM, across all multi-program workloads. SPDS-DBR improves workload speedup in comparison to FAM by about 20% and 15% for the 3H-3L and 3M-3L workloads respectively, while it performs

on par with FAM for the 3H-3M workload. This is because DBR works with code statically targeted for basic instructions as we discussed in the previous section.

The results show that dynamic scheduling, enabled by flexible migration, can efficiently leverage our binary-level techniques for implementing optimizing policies to out-perform FAM forced migration scheduling.

3.5 Chapter Conclusion

In this chapter we presented two fast dynamic binary adaptation techniques, DBR and DBT, for flexible, cross-core thread migration in shared-ISA MPSoC platforms. DBR adapts binary code on demand for the host core ISA. DBT employs *fault-and-rewrite* in conjunction with a translation scheme to execute hardware unavailable accelerating instructions. Both methods achieve software transparent execution despite of ISA heterogeneity. Additionally, they enable flexible, cross-core thread migration in contrast to *fault-and-migrate* (FAM) which is the state-of-the-art approach for transparent execution on shared-ISA platforms. We have also shown a speedup-proportional dynamic scheduler which leverages flexible migration provided by our binary-level techniques to out-perform significantly FAM forced migration scheduling.

In the next chapter we focus on dynamic scheduling to present a complete scheduling framework which optimizes the execution of a workload as whole using online profiling techniques and heuristic algorithm to decide on optimizing cross-core migrations.

Chapter 4

Speedup Aware Dynamic Scheduling on Shared-ISA Architectures

Efficient execution on shared-ISA architectures, as on any heterogeneous architecture, depends on the optimal mapping of the computation to heterogeneous resources. Existing state-of-the-art scheduling for shared-ISA architectures relies on the fault-and-migrate technique. However, this limits scheduling by dynamically setting core affinities that over-subscribe accelerating cores, resulting in loss of performance.

In this chapter we present a speedup-aware, dynamic scheduling method which adapts thread-to-core assignments of co-executing programs to maximize the speedup of the workload. Our method leverages dynamic binary adaption for on demand, cross-core migrations and online, instruction-level profiling to quantify speedup from accelerating instructions. It implements a heuristic scheduling algorithm which evaluates different possible thread-to-core assignment and selects the one which maximizes speedup.

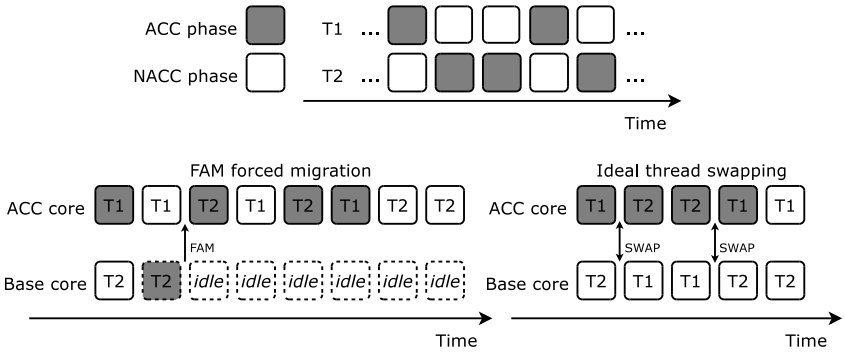
4.1 Introduction

State-of-the-art schedulers for shared-ISA systems prioritize programmability and transparent execution over performance. They are based on the *fault-and-migrate* (FAM) [64, 91] technique which assumes binaries implement the ACC ISA but a thread can execute in any core in the system. If a thread executes an accelerating instruction on a non-ACC core, FAM forcibly migrates it to an ACC core. FAM-based schedulers implement a *migrate-back* timeout in an effort to alleviate congestion on ACC cores due to forced migrations. When the timeout expires, a migrated thread is moved back to its original core. This approach has severe limitations: (i) FAM forced migration sets dynamically core affinities on a per-thread

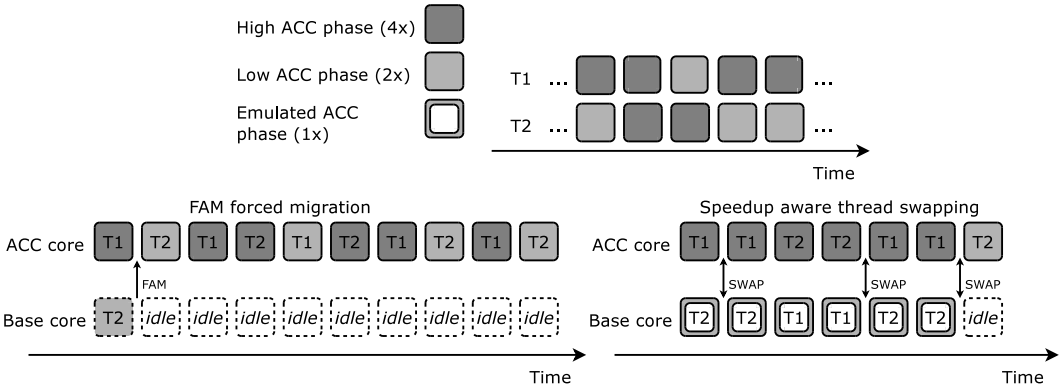
basis, precluding any other dynamic scheduling from implementing global optimization policies, (ii) it over-subscribes ACC cores with threads and limit acceleration of those threads due to time-sharing the ACC core, even when the migrate-back timeout is in effect. (iii) FAM may trigger an excessive number of unavoidable thread migrations and the associated loss of performance. Most importantly, FAM forced migration prohibits any dynamic scheduling approach to improve performance or power consumption through efficient utilization of heterogeneous resources. Dynamic scheduling is crucial to adapt execution to the availability of these resources and to the the varying acceleration potential of different threads executing concurrently [62, 82, 96, 99, 100].

Figure 4.1 shows indicative scenarios of FAM-based scheduling which result to degraded performance, either due to unavoidable over-subscriptions or because of ignoring the acceleration potential during thread execution. Figure 4.1a shows an indicative example, following an execution snapshot of two threads, T1 and T2. The threads go through ACC phases (shaded) and NACC phases (non-shaded), while running on a shared-ISA multicore with one ACC and one non-ACC (Basic) core. A FAM scheduler would force T2 to migrate to the ACC core when entering an ACC phase, effectively sharing the ACC core between T1 and T2 and leaving the Basic core idle. However the Basic core could execute NACC phases concurrently to speedup workload execution. In an ideal thread swapping scheme, the scheduler would swap T1 and T2 between the ACC and Basic cores given that T2 executes an ACC phase and T1 a NACC phase, or vice versa. In the same spirit, even when threads have overlapping ACC phases but with different speedup, the scheduler should adapt thread-to-core assignment for efficient sharing of ACC cores. Figure 4.1b shows an example of thread execution with overlapping ACC phases of different speedup. Note that swapping two threads which both execute ACC phases has the additional complexity of binary incompatibility, motivating our DBT method. ACC instructions on a thread being swapped out of a ACC core would need to be replaced with equivalent non-ACC instruction sequences, in order to execute on a non-ACC core. T1 and T2 go through high and low ACC phases. A FAM scheduler would force T1 and T2 to time share the ACC core. Thread swapping would intelligently swap threads between the ACC and Basic core to execute high ACC phases on the ACC core. Low ACC phases can execute emulated on the Basic core, albeit without acceleration.

In this chapter we present a dynamic scheduling framework and explore alternative scheduling policies for shared-ISA asymmetric multicores. Specifically, the framework consists of heterogeneous-ISA aware mechanisms to quantify speedup and enable transparent execution and dynamic thread migration despite ISA incompatibilities. Leveraging these techniques, we explore speedup-aware policies that selectively swap threads between ACC and non-ACC cores for maximizing workload speedup. Our policies are based on two



(a) Thread execution snapshot with disjoint ACC phases



(b) Thread execution snapshot with overlapping ACC phases but with different speedup

Figure 4.1 Execution snapshots motivating thread swapping vs. FAM forced migration

observations: first, that threads execute through phases with (ACC) and without (NACC) acceleration potential and second, different threads have different acceleration potential during ACC phase execution. The acceleration potential in ACC phases varies depending on the number and type of ACC instructions retired during these phases. In summary, this chapter makes the following contributions:

- We present a set of distinct mechanisms for heterogeneous ISA execution profiling and management. These include mechanisms for hardware and software assisted profiling at runtime and a cross-core speedup estimation methodology. Profiling gathers data from customized, per-core hardware counters and Dynamic Binary Translation (DBT), presented in Chapter 3, software counters for accounting the type and frequency of ACC instructions executed. Cross-core speedup estimation takes as input these profiling data and estimates cross-core execution speedup for each thread. The combination of these techniques lifts the key limitations of FAM, which are forced migration and speedup unawareness, and allows the implementation of global, optimizing scheduling policies.

- We present the SPEEDSWAP scheduler, a scheduler that optimizes workload performance on shared-ISA heterogeneous multicores. SPEEDSWAP achieves this by adjusting periodically thread-to-core assignments, using thread swapping, with the criterion of maximizing workload speedup. It leverages continuous profiling coupled with speedup estimation to quantify thread performance and DBT for enabling cross-core dynamic migration. SPEEDSWAP operates in time epochs during which it keeps the scheduling history and associated speedup profiles for all executing threads. Basicd on this history and the most recent speedup estimate for each thread, SPEEDSWAP swaps threads between heterogeneous cores only when this results in maximizing the workload speedup. This informed scheduling approach is in contrast to FAM scheduling, which operates obliviously to thread profiles and workload wide scheduling information.

- We evaluate SPEEDSWAP on a hardware prototype of a shared-ISA system, using multi-program workloads. Our experiments include different hardware configurations, varying the ratio of ACC and non-ACC cores, to factor in the effects of possible hardware configurations on FAM and SPEEDSWAP scheduling. Multi-program workloads include threads of applications with varying acceleration potential, from low to high, to assess the performance and robustness of SPEEDSWAP and FAM scheduling. SPEEDSWAP outperforms FAM in all tested hardware configurations. As the ratio of Basic over ACC cores increases, so does the performance advantage of SPEEDSWAP over FAM. SPEEDSWAP achieves as much as 2.5× faster execution times over FAM

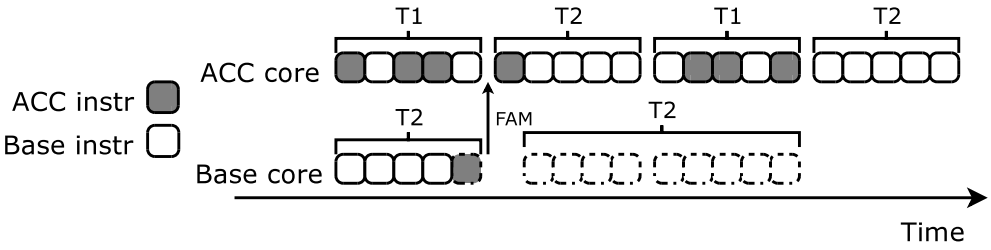


Figure 4.2 A single ACC instruction triggers FAM, resulting in migration and unneeded ACC core sharing

when there is a 4:1 ratio of Basic to ACC cores. FAM scheduling limitations are less prominent when there is a 1:1 ratio of Basic to ACC cores. Even in this case, SPEEDSWAP results in no less than 1.04× workload speedup compared to FAM.

The rest of this chapter is organized as follows: Section 4.2 reviews related work. Section 4.3 describes the hardware and software assisted profiling and section 4.4 presents the cross-core speedup estimation methodology. Section 4.5 presents scheduling including SPEEDSWAP speedup-aware approach and the FAM scheduler. Section 4.6 includes the evaluation of our work including the experiment methodology and implementation details. Section 4.7 concludes the chapter.

4.2 Background

4.2.1 Fault-and-migrate Scheduling on Shared-ISA Architectures

FAM transparent execution requirement relies on forced migration which implicitly implements a scheduling policy of migrating threads executing ACC instructions to ACC cores. FAM is fundamentally dependent on the assumption that a thread forcibly migrated to an ACC core would obtain enough speedup to amortize the cost of migration and the cost of potential core sharing. This assumption is restrictive and can lead to degraded performance. Figure 4.2 shows an indicative example. Thread T2 migrates to a ACC core due to a single ACC instruction but later executes only non-ACC instructions. The acceleration gained from executing the ACC instruction natively does not amortize the cost of migrating the thread and time sharing the ACC core.

The shortcomings of FAM are summarized to its obliviousness to core load on forced migration and its unawareness of acceleration potential, as shown in figure 4.2. We enhance the FAM scheduler with two techniques to address those shortcomings. The first is that a FAM forced migration will always choose the least loaded ACC core to move a thread to, that

is the ACC core with the fewest runnable threads. The second technique is *migrate-back* [64]. When FAM scheduling implements migrate-back, a thread migrated to an ACC core does not run indefinitely on this core, instead it is moved back to a Basic core after a fixed time interval. This technique alleviates both congestion and the fact that migrated threads may transition to a non-accelerated phase, under-using ACC core resources. However, migrate-back may induce extra migrations if a thread enters again an accelerated phase on a Basic core.

Overall, FAM has significant deficiencies because of its constraining approach to implement transparent execution through forced migration. Most importantly, it deprives any scheduling policies to optimize workload performance, since thread-to-core assignment is dictated by necessity rather than performance. Note that the DBT mechanism overcomes the forced migration limitation by using binary emulation for transparent execution. The SPEEDSWAP schedulers leverage DBT to implement on demand, cross-core migration for implement their workload optimizing policies.

4.2.2 Other Dynamic Scheduling Solutions for Asymmetric Platforms

Asymmetric multicore platforms, categorized by ISA heterogeneity, are divided into single-ISA, shared-ISA and disjoint-ISA systems. In single-ISA systems performance asymmetry comes from variation of ISA-transparent characteristics, and typically consist of few powerful, large area cores and many smaller and power-efficient cores [60, 62]. Single-ISA performance asymmetry is most frequently emulated by varying core clock frequency, preserving software binary compatibility [32, 97, 98, 100]. Shared-ISA architectures extend micro-architectural, single-ISA asymmetry with *functional asymmetry*: all cores implement a common, basic ISA while few of them extend it with accelerating instructions for executing certain operations accelerated. Shared-ISA architectures enable ISA heterogeneity, hence providing more opportunities for functional specialization and power optimization. However, ISA heterogeneity exposes to software because of binary incompatibilities. Disjoint-ISA architectures, such as the Cell BE and CPU+GPU configurations, consist of cores that have entirely different architectures and follow the offload model, where critical parts of the computation are explicitly offloaded to specialized accelerators. Disjoint-ISA challenges relate more to programmability rather than scheduling. System software must provide ways to abstract hardware diversity to ease developer effort.

Kumar [60] and Hill [47] establish the case for single-ISA, asymmetric architectures for power reduction and performance. The two key observations for scheduling on asymmetric systems are *efficiency specialization* [60], meaning powerful cores should host execution phases that make use of their performance enhanced resources, and *TLP specialization* [5], meaning parallel execution phases should execute on the multitude of basic cores to make

use of hardware parallelism whereas sequential execution phases should execute on powerful cores for maximum single-threaded acceleration.

Kumar [62] and Becchi [10] define efficiency specialization on single-ISA systems as assigning compute intensive phases on accelerated cores and memory intensive phases to basic cores. They characterize execution phases by using a sampling based method for training: threads are forced to periodically migrate to different cores in the system for probing performance counters, such as IPC, to identify as compute or memory boundness. For efficient utilization of cores, scheduling assigns those threads with maximum sampled IPC to the accelerated cores. Alternatively, when scheduling aims for global optimization, it picks the one sampled thread-to-core assignment which has the maximum average IPC across all threads. Periodic migration for sampling introduces significant overhead, since execution cycles are spent for monitoring. Also, correct detection of execution phases depends on the sampling frequency during training. Higher sampling frequency gives better detection results but produces intolerable overhead. Sampling all possible thread assignments does not scale and becomes practically infeasible as the number of co-executing threads and different cores increases.

Shelepov [100] proposes *architectural signatures*, to avoid the overhead of online profiling. Architectural signatures are generated statically, one-off, by running an application with a training input set and calculating its reuse-distance profile to estimate memory boundness. The scheduler reads per-application architectural signatures and estimates performance on different cores. Also, it takes into account time-sharing because of threads co-located on the same core. A change in thread count during a particular schedule triggers a re-evaluation of thread-to-core assignments. Architectural signatures build an application's profile offline to avoid runtime overhead but doing so cannot capture dynamic and input-dependent phases during thread execution. Application characterization depends on the training data set which is possibly different than the actual input, thus affecting signature's accuracy or requiring re-training.

Fedorova [32] presents a TLP specialization method for single-ISA asymmetric systems. The idea is that sequential execution accelerates more efficiently on accelerated cores, whereas parallel phases execute more efficiently on the numerous basic cores to exploit hardware parallelism. The scheduler decides on thread-to-core assignments on a per-application basis, examining the application's thread count, to determine whether being in a sequential or in a parallel phase.

Saez [96, 97] combines efficiency and TLP specialization into a single metric, the so-called *utility factor* of a thread. LLC misses are used to estimate memory boundness and the application's thread count is used for detection of sequential or parallel execution phases.

The utility factor is dynamically re-evaluated and guides the scheduler to migrate threads between accelerated and basic cores but without tracking scheduling history or attempting to optimize workload speedup.

Li [64] abstracts performance estimation by assuming static performance ratings for asymmetric cores and uniform intra-thread speedup through execution, depending only on a rating of the executing core. This work proposes a credit-based, round-robin scheduling algorithm for fair sharing of accelerated cores. As this work targets fair sharing, it is not concerned with optimizing performance and makes simplifying assumptions about asymmetry and heterogeneity that obviate the need for performance estimation.

Van Craeynest [106] considers in-order versus out-of-order cores to argue that memory intensity alone is not a good indicator for scheduling. Instead, they show that both ILP and MLP should be factored in. Similar to our work, they build a performance estimation model based on performance counters and known micro-architectural differences to estimate the performance of a thread in any of the system's core types. They experiment using a simulated environment and consider both static scheduling, disallowing migrations by choosing a schedule one-off, and dynamic scheduling. Dynamic scheduling periodically re-evaluates thread-to-core assignment decisions to capture different execution phases. In both cases performance estimation drives scheduling decisions. However, they consider single-ISA asymmetry, hidden from software, unlike instruction-based heterogeneity in shared-ISA systems. Furthermore, simulation allows for a synchronized, global view of potential schedules whereas in our work we devise a thread-swapping algorithm on a real system, running in a distributed manner to experiment with real-time latencies.

Efficiency specialization has been applied in areas where application-level information can improve performance. Suleman [104] and Joao [54] propose to use the accelerated cores for accelerating bottlenecks in multi-threaded execution. Specifically, to accelerate threads executing critical sections by placing them to accelerated cores or to accelerate threads lagging behind for reaching barrier synchronization. Kazempour [55] views efficiency specialization in relation to virtualization. The idea is to share accelerated cores between VMs to improve performance, or prioritize some VMs to schedule on accelerated cores for service differentiation.

4.3 Online Speedup Profiling

Implementing a dynamic, system-wide performance optimization policy for shared-ISA multicores necessitates a mechanism for profiling thread execution. Profiling needs to be done at the instruction level to measure the impact of ISA heterogeneity on thread execution,

including the effect of emulation. In particular, the number of occurrences and type of ACC instructions a thread executes determine its acceleration potential when running on an ACC core or equivalently its performance handicap when running them emulated on a Basic core. Profiling at this instruction level must be done only at runtime, since instructions executed depend on the program's control flow and possibly on its input. Note that our profiling method measures execution latency at the binary level, that is for the ACC ISA. Because of DBT emulated execution, a hardware counter solution is insufficient as it can only account for natively executed ACC instructions. We present a combined hardware and software assisted profiling method, comprising of a specialized hardware performance monitoring unit (PMU) coordinating with DBT for profiling emulated execution.

The function of the profiling method is to provide execution latencies broken down into the different instruction types at the binary level (ACC ISA) regardless of executing on an ACC or a Basic core. Those profiling data are necessary to do speedup estimation, discussed later. Regarding the hardware requirements for profiling, each core has a dedicated PMU attached to it. The PMU monitors execution at the instruction level and operates in two modes: *native* and *emulation* mode. Note it can be signaled dynamically to switch between modes, explained later.

The native mode is the initial operating mode of the PMU, performing monitoring of natively executing instructions. Native execution includes basic instructions, which by definition are implemented natively on all core types, and ACC instructions when they execute on an ACC core. In this mode, the PMU implements different sets of hardware counters to count the number and type of instructions executed natively as well as their clock cycle latency. At each quantum scheduling interval, the OS queries the PMU to retrieve the values of those counters and stores them in the corresponding thread descriptor.

Next we discuss profiling when emulating with DBT. Our binary level profiling method needs to attribute the emulation handler latency to each emulated ACC instruction, in the same way native mode measures native execution latency. The PMU alone cannot provide this functionality, thus profiling implements a combined hardware and software assisted technique. On the software side, DBT adds a prologue and an epilogue code fragment to each emulation handler routine. In the prologue, the emulation handler increments the occurrence counter for this specific instruction type directly in the OS thread descriptor structure. Also within the prologue, the emulation handler signals the PMU to enter the emulation profiling mode. During this mode, the PMU counts only clock cycles by incrementing a dedicated hardware counter for accumulating emulation latency. In the epilogue code, the handler signals the PMU to resume native profiling and reads the PMU emulation latency counter which holds the actual execution latency of the emulation handler. This latency is also the

effective latency of executing this particular ACC instruction emulated. Finally, the epilogue code stores the emulation latency value within the OS thread descriptor. Note that emulation latencies are stored using different data fields for each ACC instruction, distinct from native latency fields.

Those profiling data are the input to the cross-core speedup estimation methodology we present in the next section.

4.4 Cross-core Speedup Estimation

Profiling attributes execution latencies at the instruction level but our dynamic scheduler needs a method to estimate each thread's performance on an ACC core. In particular, a thread may achieve speedup from ACC instructions and this varies between threads depending on their acceleration potential. This thread speedup metric needs to be quantified accurately at runtime, regardless of the host core the thread runs, to rank threads based on acceleration potential. We present a methodology for cross-core speedup estimation at the thread-level, using the data gathered through hardware and software assisted profiling.

We state our assumptions for applying our methodology upfront, before deriving our estimation model. We selectively restate some of the assumptions in the context they are used. Without loss of generality, we assume the underlying architecture consists of ACC cores, implementing in hardware both ACC and Basic instructions, and Basic cores which implement only the basic ISA. Basic instructions execute in hardware on either the ACC cores or Basic cores, thus have the same latency. This is because other than ISA heterogeneity cores have the same memory hierarchy and micro-architectural traits, so pipeline stalls from basic instructions have the same effect on both core types. ACC instructions perform computational operations, they do not implement memory operations or branches. As such, they execute fully pipelined, having a fixed latency documented in the hardware architecture specification. ACC instructions execute natively on ACC cores but they execute emulated on Basic cores through our DBT mechanism.

Following, execution time can be broken down into time for executing instructions of the Basic ISA, and time for executing ACC instructions. Formally, execution time for a core p over a number of instructions executed is:

$$T_p = \sum_{i \in Basic \cup ACC} n_i \cdot CC_{i,p}^{real}$$

$CC_{i,p}^{real}$ is real instruction execution latency, including pipeline stalls, where n_i denotes the number of type i instructions executing on core p.

On an ACC core, ACC instructions execute natively and execution latency is broken down as:

$$T_{ACC} = \sum_{i \in Basic} n_i \cdot CC^{real}_{i,ACC} + \sum_{i \in ACC} n_i \cdot CC^{real}_{i,ACC}$$

Note the hardware PMU counts the occurrences of instructions and the latency in clock cycles per instruction type.

On a Basic core, ACC instructions execute emulated instead, through emulation handlers set up by DBT. Thus, execution latency can be broken down into native and emulated components:

$$T_{Basic} = \sum_{i \in Basic} n_i \cdot CC^{real}_{i,Basic} + \sum_{i \in ACC} n_i \cdot CC^{emu}_{i,Basic}$$

where emulation latency is the number of type i emulated ACC instructions multiplied by the execution latency of the respective emulation routine which translates them to the basic ISA. Note that for profiling emulated ACC instructions, DBT coordinates with the PMU. Specifically, DBT counts the occurrences of emulated ACC instructions and signals the PMU to switch to emulation profiling mode on entering an emulation handler and resume normal profiling once the handler is done. This break down is necessary to cross-estimate the execution latency on an ACC core while actually executing on a Basic core, explained later.

Cross-estimating execution latency on a Basic core running on an ACC core makes use of the assumption that Basic instructions execute with the same latency on both cores. PMU profiling on the ACC core provides the number of occurrences for each ACC instruction. However the emulation latency of these instructions is not constant as it depends on the input to emulation routines. We solve this by combining offline measurements with an online calibration technique. In more detail, emulation latency starts with an estimated value based on offline measurements averaged over a wide variety of inputs. Online calibration operates asynchronously, updating those estimates in a per-thread basis to account for variability in emulation inputs depending on code executed by each thread. Specifically, if the SPEEDSWAP scheduler (discussed later) migrates a thread to a Basic core, profiling will measure real emulation latency and update model latencies for this thread. If at a later time the thread is migrated to an ACC core, model latencies will be updated by real measurements. To avoid bias effects, model latencies converge to the initial, average estimates using an exponential decay technique. Formally, cross-estimating execution latency on a Basic core when running on an ACC core is:

$$T_{Basic \leftarrow ACC} = \sum_{i \in Basic} n_i \cdot CC^{real}_{i,ACC} + \sum_{i \in ACC} n_i \cdot CC^{emu'}_{i}$$

Additionally, cross-estimating execution on an ACC core running on a Basic core makes use of the assumption that ACC instructions accelerate computational operations and cause no pipeline stalls. As such, their execution latency for fully pipelined execution, depends only on the instruction type. This would be a number mentioned in the architecture *spec*ification. DBT profiling on the Basic core provides the number of emulated ACC instructions executed, per-type. Formally, cross-estimated latency on an ACC core when running on a Basic core is:

$$T_{ACC \leftarrow Basic} = \sum_{i \in Basic} n_i \cdot CC_{i,Basic}^{real} + \sum_{i \in ACC} n_i \cdot CC_i^{spec}$$

Basicd on the real execution latency measured on the core a thread actually runs and on the cross-estimated latency, the speedup of executing a set of instructions on an ACC core versus a Basic core formally is:

$$SP_{ACC/Basic} = \begin{cases} \dfrac{T_{Basic}}{T_{ACC \leftarrow Basic}} = \dfrac{\sum\limits_{i \in Basic} n_i \cdot CC_{i,Basic}^{real} + \sum\limits_{i \in ACC} n_i \cdot CC_{i,Basic}^{emu}}{\sum\limits_{i \in Basic} n_i \cdot CC_{i,Basic}^{real} + \sum\limits_{i \in ACC} n_i \cdot CC_i^{spec}} & , \textit{executing on a Basic core} \\ \\ \dfrac{T_{Basic \leftarrow ACC}}{T_{ACC}} = \dfrac{\sum\limits_{i \in Basic} n_i \cdot CC_{i,ACC}^{real} + \sum\limits_{i \in ACC} n_i \cdot CC_i^{emu'}}{\sum\limits_{i \in Basic} n_i \cdot CC_{i,ACC}^{real} + \sum\limits_{i \in ACC} n_i \cdot CC_{i,ACC}^{real}} & , \textit{executing on an ACC core} \end{cases}$$

This speedup value quantifies the acceleration of a set of instructions executed on an ACC core versus a Basic core and characterizes the execution of a thread at a certain point in time. In the evaluation section, we show that our methodology for cross-estimating execution latencies, and thus speedup, is accurate, having a mean standard error of no more than 5% across our benchmarks. The SPEEDSWAP scheduler, discussed in the next section, samples each thread's speedup value in regular intervals to discover accelerated execution *phases*, that is phases during a thread's execution where speedup is almost constant.

4.5 Speedup-aware Scheduling

4.5.1 SPEEDSWAP Scheduling

SPEEDSWAP is a decentralized scheduler running on ACC cores and invoked periodically by the OS. It swaps a thread running on a Basic core with a thread running on the host ACC core, only if this swap maximizes the workload speedup. Next we discuss on the specifics of the scheduling algorithm.

SPEEDSWAP operates in epochs where an epoch is a fixed-length collection of OS quantum scheduling intervals. Additionally, SPEEDSWAP utilizes as input the *scheduling*

Algorithm 1: SPEEDSWAP scheduling algorithm

Input: P algorithm's invocation period, n number of past scheduling intervals,
T_a thread executing on this ACC core, $\mathscr{T}$ set of threads in Basic cores
$SP_{i,t}$ speedup of thread t on interval i,
SP_t estimated speedup of thread t

Output: T_{max} thread maximizing global speedup if scheduled on this ACC core

```
/* Initialize workload speedup WSP and T_max extending the schedule
   (no swapping) for the next P intervals                          */
```

$$NxtSP_{T_a} = \frac{1}{n+P}\left[\left(\sum_{i=1}^{n} SP_{i,T_a}\right) + P \times SP_{T_a}\right]$$

```
/* Threads in 𝒯 have unit speedup for the next P intervals         */
```

$$NxtSP_{\mathscr{T}} = \prod_{t \in \mathscr{T}} \frac{1}{n+P}\left[\left(\sum_{i=1}^{n} SP_{i,t}\right) + P \times 1\right]$$

$$MaxWSP = NxtSP_{T_a} \times NxtSP_{\mathscr{T}}$$

$$T_{max} = T_a$$

for $t \in \mathscr{T}$ **do**

```
/* Calculate speedup if t → ACC and T_a → Basic                    */
```

$$NxtSP_t = \frac{1}{n+P}\left[\left(\sum_{i=1}^{n} SP_{i,t}\right) + P \times SP_t\right]$$

$$NxtSP_{T_a} = \frac{1}{n+P}\left[\left(\sum_{i=1}^{n} SP_{i,T_a}\right) + P \times 1\right]$$

$$NxtSP_{\mathscr{T}\setminus t} = \prod_{n \in \mathscr{T}\setminus t} \frac{1}{n+P}\left[\left(\sum_{i=1}^{n} SP_{i,n}\right) + P \times 1\right]$$

```
/* Calculate projected workload speedup                            */
```

$$WSP = NxtSP_t \times NxtSP_{T_a} \times NxtSP_{\mathscr{T}\setminus t}$$

if $WSP > MaxWSP$ **then**

$MaxWSP = WSP$

$T_{max} = t$

end

end

if $T_a \neq T_{max}$ **then**

swap $T_a \longleftrightarrow T_{max}$

end

history of each thread during an epoch for computing the workload speedup. The scheduling history of a thread keeps track of the speedup of thread execution during an epoch. For a single scheduling interval, the speedup value is the unit if the thread has executed on a Basic core, whereas cross-estimation provides the value if it executed on an ACC core. Overall, a thread's cumulative speedup in an epoch is the arithmetic mean of quantum interval speedup values. At a higher level, the workload speedup across all threads executing in parallel is the geometric mean of each thread's cumulative speedup value. This epoch based approach captures the recent execution history for each thread and avoids bias effects when calculating workload speedup.

SPEEDSWAP is invoked periodically by the OS and subdivides the epoch to thread-swapping decision intervals. When invoked, it searches for a thread swap which maximizes workload speedup. Specifically, SPEEDSWAP, when invoked on an ACC core, performs a dry run of possible thread swaps and calculates their impact on workload speedup. It uses cross-estimation to compute projected speedup values for all threads running on a Basic core, as if they were to run on the ACC core. Then it computes the projected workload speedup assuming thread swapping. Note that the projected speedup for the swapped out thread on the host ACC core is the unit. Only if thread swapping maximizes the projected workload speedup, it is actually performed.

We describe in more detail how scheduling history is built. After every scheduling interval, the OS logs the actual speedup of the application-level thread which has executed during that interval. For threads having executed on a Basic core the speedup value is the unit. For threads having executed on an ACC core the speedup value is cross-estimated using online profiling data. Let the epoch consist of N equally sized, quantum scheduling intervals. As execution progress within an epoch, the cumulative thread speedup is: $\frac{1}{n}\sum_i SP_i$ where $n \leq N$. The cumulative thread speedup is the only metric the SPEEDSWAP algorithm needs from the scheduling history. In fact it can be computed incrementally and it is the only value the OS stores in thread descriptors. When transitioning to a new scheduling epoch, old history is discarded by zeroing each thread's cumulative speedup to avoid the bias from possibly stale execution phases. Then history is re-built based on recent profiling data.

The SPEEDSWAP algorithm is described in pseudocode in Algorithm 1. On each ACC core the OS invokes SPEEDSWAP periodically, every P quantum scheduling intervals, and effectively subdivides the epoch into P-length scheduling units distributable among threads through swapping. SPEEDSWAP initially calculates the projected workload speedup from extending the existing thread-to-core assignment until the next invocation. Next, it iterates over all possible thread swaps, hypothetically swapping a thread executing on a Basic core with a thread on the ACC core. Note we assume full subscription so each core has exactly

one runnable thread – investigating over-subscription is left as future work. The algorithm computes the expected workload speedup, by using cross-estimation method to predict performance of presumably swapped threads under the new schedule. Iff there is such a swap which maximizes workload speedup, then it is actually performed. The swapped-in thread will execute on the ACC core for the next P intervals.

We also implement SPEEDSWAP-0 which is a special case of the SPEEDSWAP algorithm that disregards execution history by assuming a 0-length epoch. SPPEDSWAP-0 targets to maximize the utilization of ACC cores by always running the threads that have the highest speedup on ACC cores. The SPEEDSWAP-0 special case for maximizing ACC core utilization contrasts the policy for maximizing the speedup of the whole workload of the original SPEEDSWAP algorithm.

4.5.2 Example

For an indicative example of SPEEDSWAP, consider a hardware system of one ACC and one Basic core executing two threads: T_1 and T_2. For simplicity, let those threads have constant speedup throughout execution $SP_{T_1} = 10$ and $SP_{T_2} = 5$. In this example, SPEEDSWAP is invoked every $P = 1$ quantum scheduling interval and the epoch length is $N = 10$. Table 4.1 shows a sample run of the algorithm, where T_1 is initially assigned to the ACC core and T_2 to the Basic core. Because of constant thread speedup values, the workload speedup can be formulated analytically:

$$\begin{aligned} WSP &= (a \cdot SP_{T_1} + (1-a)) \cdot ((1-a) \cdot SP_{T_2} + a) \\ &= (a \cdot 10 + (1-a)) \cdot ((1-a) \cdot 5 + a) \end{aligned}$$

where $a \in [0,1]$ is the ratio of ACC intervals over total intervals during which T_1 executes accelerated. Equivalently, because of swapping, a also equals the ratio of Basic over total intervals during which T_2 executes without acceleration. Moreover, the ratio $1-a$ holds the ratio of time that T_1 executes without acceleration and conversely it holds the ratio of time that T_2 executes on the ACC core. The analytic solution of $\frac{dWSP}{da} = 0$ finds the value of a which maximizes WSP, being the optimal assignment ratio. In this case $a = 0.57$ for a maximum workload speedup of $WSP = 16.67$. That is T_1 should execute $\frac{57}{100}$ intervals on the ACC core while T_2 should execute $\frac{43}{100}$ on it, and the rest on the Basic core. Note that this result contradicts *speedup proportional* assignment which would have $a = \frac{SP_{T_1}}{SP_{T_1}+SP_{T_2}} = \frac{10}{15} \simeq 0.67$. The algorithm snapshot at table 4.1 converges to $a = \frac{6}{10} \simeq 0.6$ for the scheduled ACC intervals assigned to T_1, approximating closely the optimal ratio found by the analysis.

Table 4.1 SPEEDSWAP example

$N = 10$	$SP_{T_1 \to ACC} = 10$	$SP_{T_2 \to ACC} = 5$	$P = 1$
n	T_1	T_2	SWAP
1	ACC	Basic	$\{WSP_{T_1 \to ACC} = 10.00 < WSP_{T_2 \to ACC} = 16.50\} \Rightarrow T_1 \longleftrightarrow T_2$
2	Basic	ACC	$\{WSP_{T_2 \to ACC} = 14.67 < WSP_{T_1 \to ACC} = 16.33\} \Rightarrow T_2 \longleftrightarrow T_1$
3	ACC	Basic	$\{WSP_{T_1 \to ACC} = 15.50 < WSP_{T_2 \to ACC} = 16.50\} \Rightarrow T_1 \longleftrightarrow T_2$
4	Basic	ACC	$\{WSP_{T_2 \to ACC} = 15.64 < WSP_{T_1 \to ACC} = 16.64\} \Rightarrow T_2 \longleftrightarrow T_1$
5	ACC	Basic	$\{WSP_{T_1 \to ACC} = 16.33 < WSP_{T_2 \to ACC} = 16.50\} \Rightarrow T_1 \longleftrightarrow T_2$
6	Basic	ACC	$\{WSP_{T_2 \to ACC} = 15.96 < WSP_{T_1 \to ACC} = 16.67\} \Rightarrow T_2 \longleftrightarrow T_1$
7	ACC	Basic	$\{WSP_{T_2 \to ACC} = 16.50 < WSP_{T_1 \to ACC} = 16.56\} \Rightarrow$ *no swap*
8	ACC	Basic	$\{WSP_{T_1 \to ACC} = 16.33 < WSP_{T_2 \to ACC} = 16.67\} \Rightarrow T_1 \longleftrightarrow T_2$
9	Basic	ACC	$\{WSP_{T_2 \to ACC} = 16.50 < WSP_{T_1 \to ACC} = 16.64\} \Rightarrow T_2 \longleftrightarrow T_1$
11	ACC	Basic	$\{WSP_{T_1 \to ACC} = 16.51 < WSP_{T_2 \to ACC} = 16.65\} \Rightarrow T_1 \longleftrightarrow T_2$

The epoch length (N) controls the impact of historic data on calculating the cumulative thread speedup and by extension the workload speedup. The shorter the epoch length is, the more sensitive the speedup calculation becomes to recent profiling data for sensing changes in the execution phases of threads. Shorter epoch lengths, however, have fewer scheduling intervals for the algorithm to converge. The algorithm's invocation period (P) controls how many times thread-swapping will be decided within an epoch. The shorter this period is, the faster the algorithm converges to the optimal assignment, but by inducing more swaps. Both N and P are statically configurable in our implementation – we leave the investigation of dynamically setting those parameters as future work.

SPEEDSWAP-0 is a special case of the SPEEDSWAP algorithm, where $N = 0$, thus ignoring scheduling history. Every P intervals, SPEEDSWAP-0 effectively selects the highest speedup thread to execute on the ACC core since it disregards history when computing workload speedup. SPEEDSWAP-0 is similar to existing scheduling algorithms proposed for asymmetric platforms. They strive for efficiency specialization, that is maximum ACC core utilization in our case, based only on most recent performance indicators.

4.6 Experiments and Results

4.6.1 Implementation Details

We extend the operating system to implement DBT, profiling and speedup estimation and our FAM and SPEEDSWAP schedulers. DBT is provided as an OS service for rewriting binary code and updating thread related OS and system state. The OS invokes DBT either when a thread triggers an illegal instruction exception or before the scheduler migrates a thread to an ACC core. DBT stores emulation handlers in core private, 32KB, low-latency scratchpad memories. Scratchpads function as emulation handling buffers with fast, cache-like access times (1 cycle) and are addressable with a single branch instruction. DBT saves rewriting management data in per-thread OS descriptors for later reference, when managing the handler buffer or reverting back rewritten instructions. Specifically, DBT stores a total of 28 bytes per rewriting target including the instruction's address, the instruction itself, and the emulation handler's address. On migration from a Basic to an ACC core, DBT flushes any thread emulation handlers resident in the local scratchpad buffer and restores previously emulated ACC instructions to execute natively at the target core.

As has been discussed, speedup estimation relies on hardware and software assisted profiling to quantify performance. Each Microblaze core has its own PMU. The core communicates with it through a point-to-point, low latency link using special instructions to read and write data. The OS manages the PMU issuing start, stop and query commands. Performance counters are read at each scheduling interval and then the PMU is flushed. After sampling the PMU counters, speedup estimation updates the estimate for the host thread, calculating its potential speedup from running on a ACC core. The estimated speedup is a *moving average* to smooth out the calculated value but with increased weighting of recent estimations to quickly sense changes in acceleration potential.

FAM Implementation

Each core implements locally a round-robin (RR) scheduler for time-shared execution of hosted runnable threads. An illegal instruction exception invokes the OS exception handler that implements FAM. FAM moves the faulting thread to the least-loaded ACC core natively executing the faulted instruction, and initializes the thread's migrate-back timeout to a value T, statically configurable in our implementation. The low-level RR scheduler decrements by one the migrate-back timeout at each scheduling interval the thread runs on the migrated, ACC core. When the timeout reaches 0, the scheduler migrates back the thread to its original core. The value of T timeout affects FAM performance and no single value is optimal for

all workloads [64]: a small timeout may cause excessive migrations when a thread executes frequently faulting instructions, while a large timeout may degrade performance because of over-subscribing ACC cores. In our implementation $T = 8$ scheduling intervals, which is experimentally verified to achieve a good compromise between migration frequency and over-subscribing ACC cores.

SPEEDSWAP Implementation

SPEEDSWAP implements dynamic thread scheduling by swapping threads between core run queues. The SPEEDSWAP algorithm is distributed and runs independently on each ACC core, with synchronized access to shared data. Each core's scheduling routine invokes the algorithm periodically, before executing the low-level RR scheduler, to look for thread swaps which maximize global speedup. In our implementation, SWAP has an epoch length of $N = 100$ scheduling intervals (each quantum scheduling interval equals 10 ms) and an invocation period of $P = 10$ scheduling intervals. We have found experimentally that these values show good performance with various workloads.

4.6.2 Experiment Methodology

For our experiments, hardware configurations consist of Basic cores, implementing just the basic ISA, and ACC cores, implementing the additional ACC instructions. Moreover, we exploit the FPGA's flexibility and vary the number and type of implemented cores to analyze the impact on scheduling. In our platform, there can be a maximum of 8 cores connecting directly to the external RAM memory controller, without incurring bus contention. We experiment with different ratios of Basic over ACC cores, while instantiating up to the maximum number (8) of cores: 4:1 (4 Basic and 1 ACC core), 2:1 (4 Basic and 2 ACC cores) and 1:1 (4 Basic and 4 ACC cores). Different ratios allow us to study the performance of asymmetric platforms that mix and match simple and complex cores under a fixed area or power budget. We anticipate SPEEDSWAP to perform better with a higher ratio of Basic to ACC cores, by avoiding contention on ACC cores while scheduling threads to maximize workload-wide speedup.

For our experiments the system is kept fully-subscribed. This means that multi-program workloads contain as many benchmarks as the total number of cores (irrespective of core type) and that after a benchmark completes execution, it is restarted to ensure a fully-subscribed system throughout workload execution. An experiment finishes after each benchmark has completed at least three runs. After the experiment is done, we compute the *average turnaround time* for each benchmark by taking the mean of turnaround times of completed

runs. Full subscription allows us to explore how the schedulers behave when the system operates near its maximum capacity. Under-subscription is a special case where some cores sit idle and known scheduling rules such as faster-first [64] can be used to improve utilization of ACC cores. We leave the study of over-subscription (running more threads than the total number of cores), which may have additional performance benefits in some workloads, as future work.

We compare SPEEDSWAP to FAM by computing the relative *workload speedup* after the workload execution finishes. For this speedup calculation, FAM is used as the baseline as it is the state-of-the-art method for scheduling in shared-ISA platforms. The workload speedup for comparing schedulers is the geometric mean of the measured speedup values for each benchmark in the workload. The benchmark speedup is defined as the ratio of turnaround times when the benchmark executes under FAM versus under SPEEDSWAP. Formally, workload speedup (WSP) is:

$$WSP_{SWAP/FAM} = \sqrt[N]{\prod_{b \in W} \frac{AvgTT_{b,FAM}}{AvgTT_{b,SWAP}}}$$

where N is the total number of benchmarks and the set $b \in W$ includes each benchmark in the workload.

Complementary to workload speedup, we define a fairness metric for evaluating the impact of dynamic scheduling in the workload, in sharing the ACC cores among benchmarks. We define the workload *Unfairness* [54] as the ratio of the maximum slowdown over the minimum slowdown considering all benchmarks in a workload. Slowdown is defined as the ratio of the benchmark turnaround time when it executes standalone on an ACC core, without sharing, over its turnaround time when it executes in a multi-program setup. Formally, it is defined as:

$$Unfairness = \frac{\max_{b \in W}(TT^{SP}_{b,\,ACC}/TT^{MP}_{b})}{\min_{b \in W}(TT^{SP}_{b,ACC}/TT^{MP}_{b})}$$

where $TT^{SP}_{b,\,ACC}$ is the turnaround time of benchmark b when executed standalone on a ACC core, while TT^{MP}_{b} is the turnaround time in the multi-program workload. The Unfairness metric is an indicator of performance disparity due to dynamic scheduling affecting ACC core sharing. Values closer to 1 indicate strong fairness, so sharing ACC cores is fair and no benchmark slows down. Larger values indicate unfair sharing of ACC cores. FAM scheduling is expected to be the fairest, because it does not give any priority in sharing ACC cores: a benchmark migrates to an ACC core as soon as illegal instruction faults. This means that benchmarks either time-share ACC cores when they have overlapping accelerated phases,

resulting in uniform slowdown, or each benchmark executes on an ACC core alone when there is no overlap, hence there is no slowdown.

SPPEDSWAP-0, selects always the highest speedup threads to execute on ACC cores, prioritizing those threads over the rest. It is expected to be the least fair because of prioritizing high speedup threads. On the other hand, SPEEDSWAP dynamically shares the ACC cores taking into account scheduling history for maximizing workload speedup. We expect SPEEDSWAP to be fairer than SPPEDSWAP-0 but less fair than FAM. However SPEEDSWAP is expected to result in higher workload speedup because of its maximizing workload speedup algorithm.

4.6.3 Multi-program Workloads

We evaluate FAM and SPEEDSWAP using multi-program workloads. The workloads include sets of benchmarks from the SPEC CPU2006 [45] and Rodinia [20] suites. Table 3.1, shown in the previous chapter, presents the list of the ported benchmarks, including a breakdown of executed and the *end-to-end* speedup executing on an ACC versus a Basic core. As discussed in section 3.4.2, We categorize benchmarks in *High-speedup*, *Medium-speedup* and *Low-speedup* to heterogeneous multi-program workloads.

Mixing benchmarks from several speedup classes in the same workload raises scheduling challenges for efficient execution on our shared-ISA platform. Table 4.2 shows the composed multi-program workloads for each hardware configuration we experiment with. Hardware configurations are denoted as XB-YACC, where X is the number of Basic cores and Y the number of ACC ones in the system. For example 4B-4ACC denotes a system of 4 Basic and 4 ACC cores. Notation for workloads shows the number of benchmarks from each speedup class (High, Medium, Low) in the workload mix. For example, a workload denoted as 2H-3M has 2 High-speedup benchmarks and 3 Medium ones, for a total of 5 benchmarks. Table 4.2 also shows the particular benchmarks present in the mix. For some workloads there is more than one instance of the same benchmark in order to achieve the intended speedup mix. Different instances are denoted by appending a numerical extension to the original benchmark's name. For example in the 5H workload, there are two instances of the *lud* benchmark, denoted as lud.1 and lud.2.

An experiment finishes when all benchmarks in the workload have completed at least three runs. The system is kept fully subscribed throughout an experiment: whenever a benchmark finishes its run, it is restarted until the experiment is done.

Table 4.2 Co-executing program workloads per hardware configuration

4B-1ACC	Benchmarks
5H	cfd, streamcluster.1, lud.1, streamcluster.2, lud.2
3H-2M	cfd, lud, streamcluster, milc, hmmer
2H-3M	lud, streamcluster, milc, namd, hmmer
2H-3L	streamcluster, lud, sjeng, bzip2, hotspot
5M	hmmer, srad, backprop, bfs, gobmk
2M-3L	hmmer, milc, sjeng, h264ref, nw
5L	hotspot, bzip2, sjeng, astar, nw

4B-2ACC	
6H	cfd.1, streamcluster.1, lud.1, cfd.2, streamcluster.2, lud.2
3H-3M	cfd, lud, streamcluster, milc, backprop, hmmer
3H-3L	cfd, streamcluster, lud, sjeng, bzip2, hotspot
6M	hmmer, milc, srad, backprop, bfs, namd
3M-3L	hmmer, srad, backprop, bzip2, sjeng, hotspot
6L	hotspot, bzip2, sjeng, astar, nw, libquantum

4B-4ACC	
8H	cfd.1, streamcluster.1, lud.1, cfd.2, streamcluster.2, lud.2, cfd.3, streamcluster.3
4H-4M	cfd, lud.1, streamcluster, lud.2, milc, backprop, hmmer, srad
4H-4L	cfd, lud.1, streamcluster, lud.2, bzip2, sjeng, hotspot, nw
8M	hmmer.1, milc, srad, backprop, bfs, namd, gobmk, hmmer.2
4M-4L	hmmer, srad, backprop, milc, bzip2, sjeng, hotspot, nw
8L	hotspot, bzip2, sjeng, astar, nw, libquantum, h264ref, hotspot

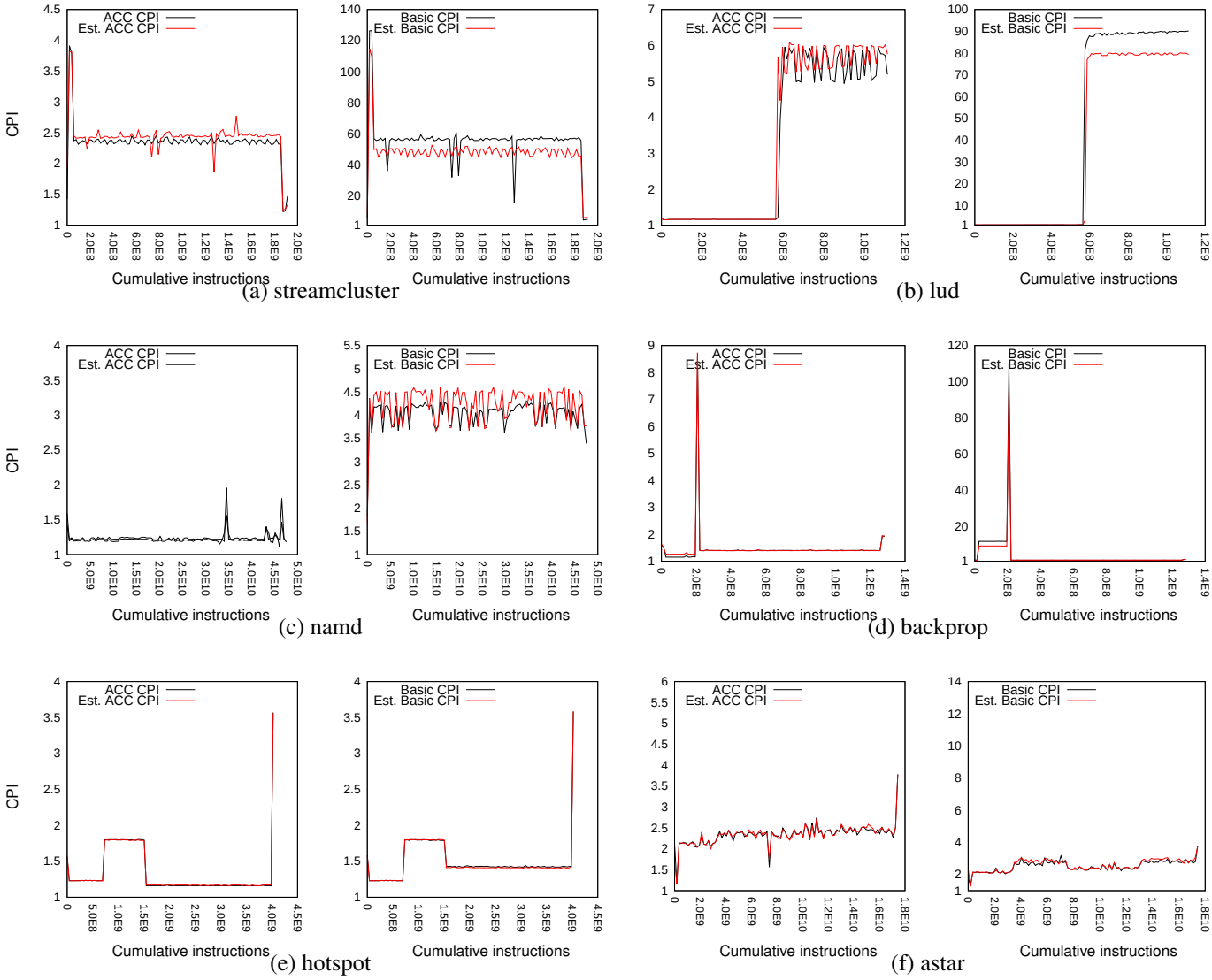


Figure 4.3 Comparison of real and estimated CPI values

4.6.4 Results

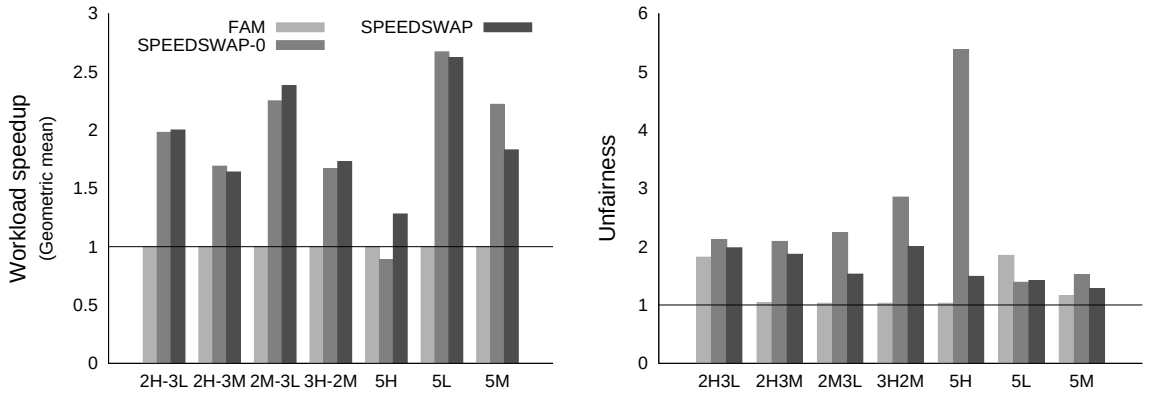
Speedup Estimation

We illustrate the accuracy of the cross-core speedup estimation technique by comparing the measured and estimated cycles per instructions (CPI). For this experiment, each benchmark runs alone, first on a Basic core and then on an ACC core. Running the benchmark on the Basic core, speedup estimation predicts CPI on the ACC core. Vice versa, running it on the ACC core, speedup estimation predicts CPI on the Basic core. For each quantum scheduling interval, we record the actual, measured CPI on the running core and the estimated CPI for the opposite core type. At the end of the experiment, there are measurements for the actual Basic CPI, the estimated Basic CPI, the actual ACC CPI and the estimated ACC CPI of the benchmark. Note that the estimated speedup is equivalent to the ratio of the Basic CPI over the ACC CPI. Figure 4.3 shows graphs for a selection of benchmarks which captures all speedup classes, plots are similar for the rest of the programs. Each graph shows the real and estimated CPI per core type, plotted against the cumulative count of executed instructions in the ACC ISA, i.e., at the binary level. ACC instructions counted have executed either in hardware (on a ACC core) or in emulation (on a Basic core).

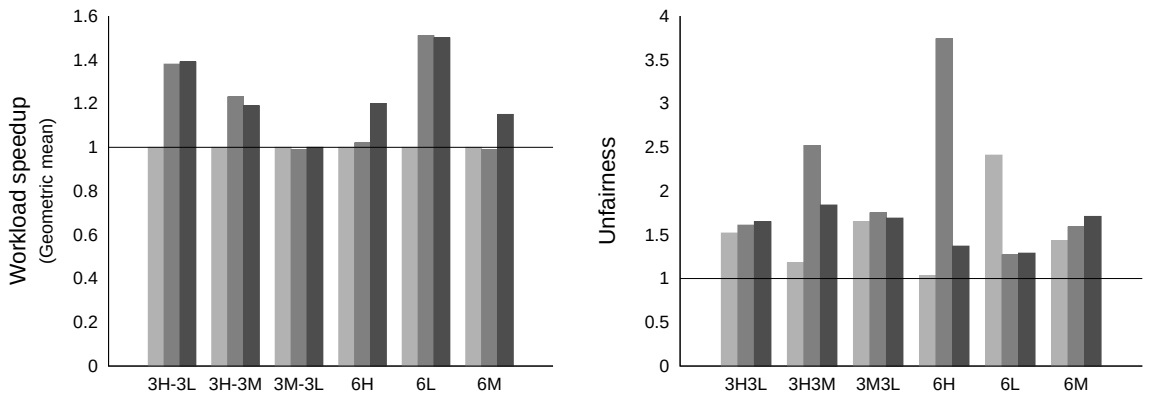
Following from observing the results, the Estimated CPI indeed tracks closely the actual CPI, either when running on a Basic core estimating CPI on the ACC core or vice versa. The mean standard error is less than 5% across all benchmarks, much less in the majority of them. Note that in these experiments, estimating the Basic CPI on an ACC core uses the initial, average latency values for emulation routines, without any calibration due to swapping. Nevertheless, estimation is accurate and, in a real deployment, swapping to a Basic core due to scheduling, will probe actual emulation latencies, thus improving future estimation accuracy.

Workload Speedup and Unfairness Metrics

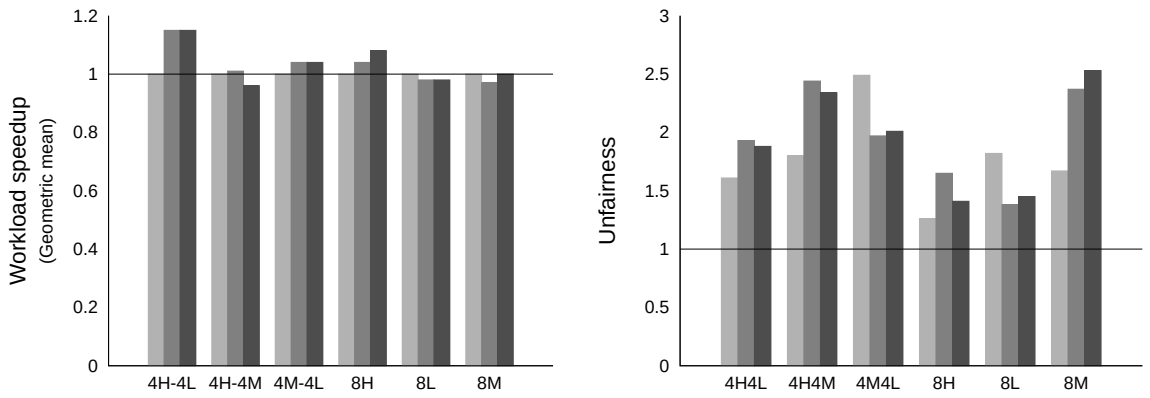
Figure 4.4 shows the results on the workload speedup and unfairness metrics for all multi-program workloads and hardware configurations. In all cases SPEEDSWAP outperforms FAM, by a higher margin as the Basic-to-ACC core ratio increases. The experiments expose FAM limitations, incurring congestion on ACC cores. Focusing the 4B-1ACC configuration, SPEEDSWAP results in up to 2.5× faster execution times than FAM. Overall, taking the geometric mean across all workloads in this configuration, shows that SPEEDSWAP has an average speedup of 1.82× compared to FAM while SPEEDSWAP-0 slightly improves this result being 1.88× faster than FAM. For the 4B-2ACC configuration, SPEEDSWAP is on average 1.22× faster than FAM. SPEEDSWAP-0 has slightly less speedup being 1.17×



(a) 4 Basic - 1 ACC core



(b) 4 Basic - 2 ACC cores



(c) 4 Basic - 4 ACC cores

Figure 4.4 Workload speedup across all HW configurations, using FAM as the reference

faster than FAM. SPEEDSWAP scheduling remains superior to FAM. In the 4B-4ACC configuration, SPEEDSWAP has the least speedup across workloads, being 1.04× faster than FAM, SPEEDSWAP-0 speedup is the same. This is because the 1:1 ratio of Basic and ACC cores alleviates the congestion effect incurred by FAM. Specifically, due to full-subscription of cores, the worst case in this configuration is to have at most two threads executing concurrently on the same ACC core. Nevertheless, SPEEDSWAP scheduling results in up to about 15% speedup over FAM, provided the workload comprises of benchmarks with speedup disparity, such as the 4H-4L workload mix.

In some cases SPEEDSWAP-0 exhibits slightly higher speedup than SPEEDSWAP. This occurs when the accelerated phases of benchmarks in the workload are disjoint in time. SPEEDSWAP-0 avoids ACC core sharing by ignoring scheduling history, swapping always highest speedup threads to ACC cores. However, in workloads where accelerated phases overlap, SPEEDSWAP-0prioritizes highest speedup benchmarks, resulting in workload speedup degradation and high unfairness. This is the case for the 5H and 6H workloads. In these workloads, all benchmarks are of high speedup and execute mostly accelerated phases. SPEEDSWAP resolves extreme unfairness, by taking into account scheduling history, thus effectively sharing ACC cores and converging for maximizing workload-wide speedup.

4.7 Chapter Conclusion

Efficiency specialization and thread migration driven by execution phase characteristics are essential in heterogeneous platforms to make use of performance and power optimization opportunities. Shared-ISA systems introduce instruction-based heterogeneity which limits thread mobility due to binary incompatibilities. The current state-of-the-art for scheduling in shared-ISA platforms is based on fault-and-migrate which causes forced migrations, interfering with efficient execution. In this chapter we introduced SPEEDSWAP, a distributed, speedup-aware scheduling algorithm to perform thread swapping between heterogeneous cores to maximize workload speedup.

SPEEDSWAP leverages a set of essential mechanisms for ISA heterogeneity and speedup awareness. These include: a DBT method which enables execution on any shared ISA-type core without imposing scheduling constraints, hardware and software-assisted profiling at runtime coupled with a cross-core speedup estimation methodology and a heuristic algorithm to find optimizing cross-core migrations.

Chapter 5

Performance Characterization on Heterogeneous Datacenter Architectures

5.1 Introduction

Datacenters carry an immensely high total cost of ownership (TCO) and energy footprint. This can be traced to the fact that datacenters are notoriously wasteful, often using as little as 10% of the supplied power for actual data storage and processing, while exhibiting less than 20% node utilization [43, 58]. Market analysts Frost and Sullivan report that in 2013 the aggregate power consumption of datacenters worldwide was approximately 30 GigaWatts [35], of which the USA was the largest consumer (11.56 GW) followed by the UK (3.1 GW) and Germany (3.85 GW). Furthermore the North American and European requirements are currently growing at a rate of 6% annually.

Micro-servers, comprised of low-power, embedded processors instead of server-class ones, have recently emerged promising to reduce the energy consumption of datacenters. Back-of-the-envelope estimates suggest that micro-server processors such as the Samsung Exynos [90] and Calxeda Highbank ECX-1000 [84] consume 5 to 30 times less power than high-end server-class counterparts, such as Intel's Sandy Bridge and are 5 to 15 times more energy-efficient (in integer performance per Watt) [86, 105].

Financial service providers operate datacenters which are co-located with market data feeds, to meet the low latency requirements of real-time analytics. These datacenters incur extraordinarily high cost of ownership due to hardware overprovisioning, their scale, and their energy consumption. The choice of compute server architecture for these datacenters fundamentally dictates their cost of ownership and sustainability. Unfortunately, there is a

lack of methodologies and metrics for fair assessment of server architectures as computational workhorses for emerging analytics workloads.

In this chapter we present a rigorous methodology and a set of metrics for evaluating performance and energy efficiency across heterogeneous servers in the context of financial analytics workloads. We introduce a platform-independent workload setup and optimization methodology, in conjunction with platform-independent metrics of performance and energy-efficiency. Based on these tools, we fairly and thoroughly compare three server architectures that represent vastly different price, cost, performance and power trade-offs: a Calxeda ECX-1000 micro-server [84] based on ARM Cortex A9 cores; a Dell server based on Intel Sandy Bridge cores; and a machine configuration based on an Intel Xeon Phi co-processor.

Our methodology yields important and often counter-intuitive findings. We establish that a server based on the Xeon Phi processor delivers the highest performance and energy-efficiency. However, by scaling out energy-efficient ARM micro-servers within a 2U rack-mounted unit, we achieve competitive or better energy-efficiency than a power-equivalent server with two Intel Sandy Bridge sockets, despite the micro-server's slower cores. Using a new iso-QoS (iso-Quality of Service) metric, we find that the ARM micro-server scales enough to meet market throughput demand, i.e. a 100% QoS in terms of timely option pricing, with as little as 55% of the energy consumed by the Sandy Bridge server.

Using the same C and SIMD code basis, we uncover several more findings of interest. The servers exhibit vast differences in energy-efficiency due to differences in the implementation of transcendental functions in hardware, as well as the implementation of vector units and ISAs. We find that power saving modes employing DVFS are invariably less energy-efficient than performance boosting modes for financial analytics workloads. We also find that while servers exhibit good performance scaling allocating more cores, they also exhibit non-ideal energy scaling, thus providing an opportunity for energy conservation via throttling concurrency.

This chapter starts by discussing related work on metrics and characterization for performance and energy efficiency in Section 5.2. Section 5.3 provides the background on server comparison methodologies and the financial analytics workload. Section 5.4 discusses code optimization and vectorization methodologies. Section 5.5 presents our experiment and measurement methodology and metrics. Section 5.6 shows the results from applying our methodology on different financial kernels running on heterogeneous machines. Section 5.7 elaborates on the QoS metric and presents an analytical approach for applying it. Section 5.8 concludes the chapter.

5.2 Related Work

Recent related work explores the performance and power consumption of low-power ARM processor models attempting to enter the server [86, 105] and HPC markets [90]. We focus on the domain of financial real-time analytics applications and further provide a comparison between fully integrated ARM-based micro-servers in scale-out configurations, against high-end Intel servers. Our study provides more insight on the viability of the ARM ecosystem for datacenters and high-performance computing, using new platform-independent metrics and a comprehensive method to ensure fair comparison. The work of Blem et al [13] is perhaps the closest to ours, exploring the performance and power consumption of several ARM and Intel processors. However, their study focuses on the energy and performance implications of ISA choices on the different processors, rather than the use of the processors at scale or their deployment in specific application domains. Anwar et al [6] study micro-server performance with Hadoop workloads. Our work further advances the state of knowledge in micro-servers, by exploring the performance and energy implications of scaling their resources up and out and by comparing micro-servers to both general-purpose servers and accelerators targeting the HPC market.

There is also related work measuring the energy-efficiency of specific algorithms and numerically intensive kernels. Alonso et al [4] model the power and energy of a specific task-parallel implementation of Cholesky factorization. Dongarra et al [31] explore the energy footprint of dense numerical linear algebra libraries on multicore systems. Korthikanti et al [56] analyzed the energy scalability of parallel algorithms on shared-memory multicore architectures, focusing on the sensitivity of the the algorithms to the critical path length and the power consumption of instruction execution and memory accesses. Our work differs in that it explores a dynamic workload with real-time execution requirements and a mixture of event-driven and data parallelism. We evaluate the power and energy of option pricing kernels both in standalone mode and in the context of end-to-end market sessions to obtain a complete view of the implications of the choice of server processors on total cost of datacenter ownership.

Iso-metrics are common tools parallel and distributed computing. Iso-efficiency [36] in terms of sustained to theoretical maximum speedup has routinely been used to compare combinations of parallel algorithms and architectures. Iso-energy-efficiency [101, 102] explores the influence of core scaling and frequency scaling on the energy-efficiency of algorithms and architectures. We establish a new metric that caters to the needs of real-time analytical workloads and emerging architectures that differ vastly in power budgets and form factors, and further establish that the new metric is more appropriate to compare server value propositions given modern hardware diversity.

Related to our work is also prior research on improving the energy-efficiency of real-time financial workloads. Schryver et al [26] present a methodology for efficient design of hardware accelerators for option pricing, whereby they cap the power consumption of the accelerator and the system as a whole. Morales et al [79] propose an FPGA design, programmable using OpenCL to build energy-efficient versions of binomial option pricing algorithms. They report a performance of 2,000 Options/second which is consistent or lower than the performance attained by our Xeon Phi and scaled-out Viridis implementations, but with a power budget of 20W, which is lower than that of any of our platforms. The method presented in this chapter fixes a workload-centric QoS metric instead of a system-centric metric, while allowing flexibility in tuning both system and workload parameters to meet the objective metric.

5.3 Background

Comparing servers requires a fair and unbiased methodology which takes into consideration production workloads and deployment environments. A major challenge lies in the fact that different server propositions represent vastly different price, performance and operating power points.

We stipulate that a comparison methodology should be platform neutral and require that the server architectures under consideration meet the same, tangible target, be it performance, Quality of Service (QoS), power consumption, or energy consumption. We thus allow scale-out or scale-up of each server to meet the set target, while comparing other metrics of interest to rank the servers. Furthermore, the metrics used should reflect workload fluctuation, but not be unduly influenced by non-deterministic environmental, hardware or system artifacts. Finally, we stipulate that any comparison should be based on the same code base, developed with similar coding effort on each server.

We use real-time option pricing workloads, which represent one of the most important and performance-critical domains of financial analytics. Option pricing is an inherently parallel problem, as each option contract is independently defined over a single stock. A portable implementation with similar effort across servers, described in our earlier work [34], exploits parallelism in the workload by using UDP multicast channels to distribute prices and POSIX threads to scale-out the processing on each core and multiple hardware contexts on each core, if present.

In addition we use the same code base to exploit data parallelism offered by vector processing units. Our code base uses manual loop unrolling and vectorization using pragmas. Our implementation is expressed in the C language, which is the norm in computational

finance. The C language standard presents a number of challenges for implicit vectorization by a compiler [3], due to inability to verify loop bounds and pointer aliasing.

On the other hand, C compilers provide implicit vectorization with associated pragmas and command line options to control the vectorization process. These features help the authoring of portable SIMD code and we leverage them in this work. Still, there is no common standard for compilers to abide by to provide the same level of vectorization nor is there a method to enforce different compilers to produce comparable vectorization output. We offer insights on how to address this problem in our methodology.

5.3.1 Computing Option Prices

In finance, the term Option means a derivative product which is a contract giving the holder the right to either buy (Call option) or to sell (Put option) one or more underlying assets, such as a fixed number of shares in a company, for a defined price and either on or before a contract end date. An Options contract, unlike a Futures contract, does not impose an obligation on the holder to exercise their right. There are several types of Options distinguished by the terms in their contract. We construct real-time analytics workloads that continuously execute Monte Carlo or Binomial Tree option pricing models. These can price both the so-called exotic American or Bermudan options as well as European options. We focus on European options for which the Black-Scholes equation provides a closed form solution, against which our results can be compared for accuracy.

Black and Scholes [11, 12] proposed a second-order partial differential equation which models the variation of the price of a European vanilla option, contractual strike price P, over time in years T, assuming that:

- the underlying asset price (spot price), S follows a log normal distribution,
- the volatility σ of S is constant and
- the risk free interest rate, or rate of return, r, expressed with continuous compounding is constant.

Under these conditions an analytic solution is:

$$\text{Price} = (-1)^p \left(SN((-1)^p d_1) - P\mathrm{e}^{-rT} N((-1)^p d_2)\right) \tag{5.1}$$

In this equation $p = 1$ for a Call option and $p = 0$ for a Put option and d_1 and d_2 are defined by:

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left(\log\left(\frac{S}{N}\right) + \frac{r+\sigma^2}{2}T\right) \quad \text{and} \quad d_2 = d_1 - \sigma\sqrt{T} \tag{5.2}$$

where $N(x)$ is the cumulative normal distribution function (CDF).

Moving beyond the limitations of the Black-Scholes model, the price of an option on any given date can be modeled using stochastic calculus, essentially simulating the path of the underlying variables over a set of paths within a time window. Analytical solutions for the stochastic equations are not generally possible so that a variety of computational numerical solution methods have been developed. European vanilla options can also be priced using these numerical methods. We apply two models, with distinct computational characteristics, as described next.

Monte Carlo (MC)

At the center of a Monte Carlo simulation of option pricing is a rate-limiting for-loop, an aspect that it shares in common with HPC applications in many fields. For a Put option the formula is [87]:

$$\text{Price} = \frac{\mathrm{e}^{-rT}}{N}\sum_{i=1}^{N}\max\left(0, S - P\mathrm{e}^{(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}x_i}\right) \tag{5.3}$$

In equation 5.3, x_i $(i = 1\ldots N)$ are a set of random numbers drawn from the standard normal distribution. The formula for a Call option is similar to the Put option.

Given that σ, r and T are constant within the context of the loop, the implementation of equation 5.3 creates a compute bound process dominated by the exponential function, with relatively few loads and stores to memory. One bottleneck is the control of numerical round-off error associated with a floating point summation process over a large number of terms [46, 67]. Another is the generation of sets of good quality pseudo-random numbers (PRN) because the computed price converges only slowly as $O\left(\frac{1}{\sqrt{N}}\right)$. This is significant because the operation count during execution of the for-loop code scales as $O(N)$. In our work, uniform random numbers are generated using the 32-bit version of the Mersenne Twister algorithm [77], then transformed to a standard normal distribution using the Box-Muller formula.

Binomial Tree (BT)

The binomial tree option pricing model [51] builds a lattice of options prices with a root node at the starting date and multiple nodes at the end date. The model decomposes the time variable into discrete steps, each corresponding to a level in the lattice. The price can

be computed at any level in the tree, which corresponds to an intermediate date within the contract, thereby making the binomial model suitable for pricing American and Bermudan options. In general at each time point i, there is a vector of option prices computed, S_{ij}. Given price S_{ij} and a pair of factors u and d representing the up and down movements, the two possible prices at the next level are $S_{(i+1)j} = uS_{ij}$ and $S_{(i+1)(j+1)} = dS_{ij}$ where the factors u, d are constant across the tree and are formally defined as:

$$u = \mathrm{e}^{\sigma\sqrt{T}} \quad \text{and} \quad d = \frac{1}{u} \tag{5.4}$$

The complete algorithm therefore has three steps:

- Given S_0, the current spot price, work forward from today, the date of computation, to the expiration date (timepoint N), applying the up and down factors at each step and thereby computing all final node prices S_{Nj}.
- At each final node of the tree (level n) compute the exercise value.
- Iterate backwards from the final nodes in the tree and for every intermediate node compute the option price assuming risk neutrality, which means that the price is computed as the discounted value of the future payoff.

The final step; a nested for-loop, dominates the time taken by the computation. The number of updates performed scales as $O(\frac{N^2}{2})$ with each update consisting of two floating point multiplications and one floating point addition. This contrasts with the MC algorithm, where there is a need to repeatedly compute transcendental mathematical functions. The convergence of the method is a function of the number of timesteps chosen for the computations.

It is not necessary to hold all of the prices S_{ij} in memory at the same time. Only the prices at timepoint i are needed. A vector with the elements being overwritten as the values are processed during the backwards iteration process suffices for this purpose. The BT implementation is thus dominated by pointer dereferencing to generate array indices and by data move activity, in contrast to the random number selection and sum reduction characteristics of the MC algorithm.

5.4 Optimization Methodology

Initially we developed a common C language code base implementing the MC and BT kernels described above to compile on all servers. This includes an algorithmic optimization which we developed for the MC kernel which we describe below. We used the gcc and icc

compiler suites (icc is not available on the ARM system and gcc is not available on the Xeon Phi) to apply the loop unrolling and auto-vectorization facilities offered by each compiler to our code base. Setting command line flags on the compiler is all that is needed to enable the automatic parallelization. Furthermore, we experimented with an approach in which we applied manual loop unrolling and direct programming with available vector registers and vector instructions through either assembler code or compiler supplied intrinsic functions which map to these assembler instructions.

Our methodology is common across all platforms although the details of the manual implementation differ due to the characteristics of each instruction set architecture. In the following sub-sections, we explain the salient details of how our we instantiated our optimization techniques in each kernel. Our work reflects a fixed development effort of approximately thirty days to create, optimize and test the code base.

5.4.1 Algorithmic Optimization of the Monte Carlo Equation

The max() function in the MC kernel is an if-statement which hampers performance and power efficiency optimizations, as will be clear in the results section. Given that all variables in the MC loop are invariant except for the random number x_i, we calculated a threshold Thres such that any x_i need only be included in the the summation when $x_i >$ Thres. For Call options

$$\text{Thres} = \frac{1}{\sigma\sqrt{(T)}} \log_{\mathrm{e}} \frac{P}{S\mathrm{e}^{(r-\frac{\sigma^2 T}{2})}} \tag{5.5}$$

This pre-screening leads to algebraic simplifications facilitating loop vectorization as explained below and also allows several multiply operations to be factored outside the sum loop, which now becomes:

$$\sum_{j=1}^{M} \mathrm{e}^{\sigma\sqrt{T}\,x_j} \tag{5.6}$$

where $M < N$ is the number of random numbers passing the threshold test. The loop length is now shorter and many executions of the expensive exp() function are therefore avoided. We observed in some cases a four-fold reduction but this depends on the values of the input data defining equation (5.5).

5.4.2 Vectorization of the Monte Carlo Kernel

We first unrolled the loop in equation 5.6 and then used vectorization to expedite the work within each iteration. We compute the summation in single precision and on Intel SSE and ARM NEON unrolled by a factor of 4, populating a single 128 bit argument with the

exponents. For the AVX implementation, we exploited the wider 256 bit registers unrolling by a factor of 8. In all cases passing the argument to the vectorized version of the Cephes software library [80] allowed us to compute the exponentials in parallel. The Cephes library is an open source math library with versions tailored for the different vector units used in our study.

Vectorization of the Binomial Tree Kernel

The binomial tree kernel is dominated by an inner loop computation, a step of which can be expressed as

$$x_i = ax_i + bx_{i+1} \tag{5.7}$$

This loop step exhibits an anti-dependency between iterations. This dependency needs to be removed in order to boost performance through vectorization. Notably, the GCC compiler for all our servers failed to recognize this vectorizing opportunity. However, The Intel compiler (ICC) managed to resolve this dependency and produce vectorized code.

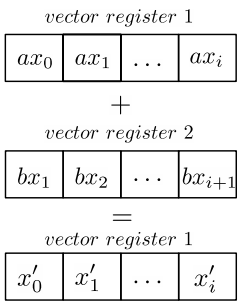


Figure 5.1 Vectorization of the Binomial Tree kernel

We have constructed two handcrafted vectorized solutions for the kernel: one which replaces the inner loop entirely by invoking a routine written in vectorized assembly, and another using compiler vector intrinsics placed inline with the C source code. The latter are referred to as INTRINSICS_xxx build and runs, where xxx depends on the platform. These two handcrafted vectorization solutions overcome the apparent anti-dependency and enable us to compare the low-level manual assembly solution versus a more abstract, intrinsic-based vectorization method. Both versions employ loop unrolling in conjunction with vectorization to reduce loop overhead. Figure 5.1 shows schematically the vectorized BT calculation. The unroll factor is equal to the vector register width measured in double-precision floating point numbers (single-precision in the case of the ARM Cortex A9 NEON unit). Similarly to the Monte Carlo algorithm, we present a detailed example of vectorizing the BT calculation,

using the the SSE 128-bit instruction set as an example. The sequence of steps for the vectorized iteration is as follows:

- Partition the XMM registers into three sets and load eight double precision values per iteration into one set of four XMM registers which we view as a unit.
- Avoiding repeated memory accesses, replicate the register contents into a separate set of four registers rearranging the contents as if performing a push-up operation on the unit. One extra memory access is needed to load a value to fill the gap at the lower end.
- Prepare a third set consisting of two vector registers such that one holds two copies of a and the other two copies of b. This can avoided in subsequent iterations.
- Apply multiply and add vector instructions to this register configuration to compute eight values for the left hand side of equation 5.7, where these values are in the first register set.
- Store contents of the first set of registers to RAM.

On the AVX and KNC architectures, the wider registers allow for sixteen values to be processed in each iteration. Notwithstanding the availability of only single precision vector floating point operations on the ARM NEON unit, a number of differences in the instruction set architectures emerged between platforms when working at the assembler code level. SSE and effectively the same throughput as AVX.

Regarding step 2, Intel SSE provides inter-lane shift operations on XMM registers making the task possible in three vector instructions. The AVX instruction set lacks inter-lane shifts on YMM registers so that an intricate sequence of seven permutation and blend operations are needed to perform the same operation and this affects the results as shown later. In contrast, the NEON and KNC instruction sets provide a vector align/extract operation thereby requiring just a single instruction. Regarding step 4, only KNC provides a fused multiply-add operation meaning that the step requires just two vector instructions while the others require three instructions.

5.4.3 Compiler Based Vectorization

To test compiler-based vectorization (labeled AUTOVECT in our analysis) with GCC, we first used the same GCC compiler flags on all servers: -O3 -Wall -g. We linked the libevent (needed by the data feed handler, to be discussed further in Section 5.5), ipmi-1.4.4, math and Pthread libraries statically during the compilation process. To test vectorization with ICC

on the Intel Sandy Bridge server, we used the -mavx flag to enable code generation for the AVX vector instruction set. For the Intel Xeon Phi server we used the ICC -mmic flag. As for ARM, we used the -mcpu=cortex-a9 -mfpu=neon GCC compiler flags. Alternatively the vectorization flags were all disabled for the plain vanilla no-vectorization (labeled NOVECT) builds and runs.

5.5 Experiment Setup and Measurement Methodology

Our experimental setup includes three platforms on which we execute the OptionPricer and collect workload-specific performance and energy metrics. The next section defines those metrics, section 5.5.2 describes the platforms and section 5.5.3 provides details of the methodology used to obtain the power readings and calculate the energy consumption.

5.5.1 Definition of Metrics

Option pricing in finance takes place by consuming a live streaming data feed of stock market prices, often within the context of high frequency trading (HFT), and for pre-trade risk analytics. The execution time characteristics of option pricing are different from those of numerical simulation in computational science using HPC.

By contrast to scientific codes which have measurable setup and post-processing phases, option pricing runs relatively small standalone kernels at very high frequency with little set up and post processing. Option pricing on live market data feeds is a form of event processing. Based on these distinctions we present and use three workload-specific metrics to compare servers under financial analytics workloads:

Joules/option (J_{opt}) The energy consumed per execution of a pricing kernel is a fundamental metric given that this step is repeated at very high frequency throughout a trading session. In the case of an actively traded stock, with a high number of defined option contracts, this building block is executed repeatedly throughout the trading day. Correspondingly, a reduction in this value can result in significant energy savings for providers offering option pricing services.

Time/option (S_{opt}) In contrast to providers, end users, particularly those engaged in HFT, are sensitive to end-to-end latency, thereby constraining the elapsed time per option metric. This metric in turn can be used to evaluate the total time to price all contracts for a given stock. Option pricing shares this time-to-solution performance metric in common with HPC applications.

QoS New prices may arrive at any time in a trading session. This means that any contracts not yet priced using the previous price update are abandoned and deemed unusable. Related to the Time/option metric, but also dependent on market activity, we define the Quality of Service metric (QoS) as the ratio of successful to the total requested option pricings. The QoS metric is an application-specific measure on meeting option pricing performance requirements. It is useful for characterizing application-related performance and scalability offered by deploying multiple nodes. It is worth noting that QoS depends on the stock price change rate and other market activities at the time of its calculation, so it will be different each time it is calculated in a live market scenario.

5.5.2 Hardware Platforms

We used three platforms, one state-of-the-art server architecture with Intel Sandy Bridge processors (briefly referred to as "Intel" in the rest of this chapter), one state-of-the-art HPC architecture with Intel Xeon Phi Knights Corner coprocessor (referred to as "Xeon Phi") and a Calxeda ECX-1000 micro-server with ARM Cortex A9 processors, packaged in a Boston Viridis rack-mounted unit (referred to as "Viridis"). We used the 4.7.3 version of the GCC compiler and the Intel Compiler ICC version 14.0.020130728 for code generation, the latter only on Intel platforms. The three platforms offer the possibility of scaling their frequency and voltage through a DVFS interface. We conducted experiments with the highest and lowest voltage-frequency settings on each platform, to which we refer as performance mode experiments and powersave mode experiments respectively. The details of the platforms are as follows:

Intel is an x86-64 server with Sandy Bridge architecture, with 2 Intel Xeon CPU E5-2650 processors operating at a default frequency of 2.00GHz and equipped with 8 cores each. The machine has 32GB of DRAM (4 $\times$ 8GB DDR3 @ 1600Mhz). The frequency in powersave is 1.2 GHz, while in performance mode it is 2.0 GHz. The server runs on Linux CentOS 6.5 with kernel version 2.6.32 ($2.6.32 - 431.17.1.el6.x86\_64$).

Xeon Phi (Knights Corner) is a many core, x86-64 co-processor board (5110P model) over PCIe. It features the many integrated cores (MIC) architecture which offers sixty, 4-way hyperthreaded cores, each equipped with a very wide (512-bit) vector unit. The board has 6 or more GB of GDDR5 as a DRAM. In performance mode the frequency is 1.053GHz and in powersave mode it is 842.104MHz. High performance and high energy efficiency are the result of featuring a highly parallel many core design while running in low clock speeds.

The system runs on Linux kernel 2.6.38.8+mpss3.2.1.

Viridis is a 2U rack mounted server containing sixteen micro-server nodes connected internally by a high-speed 10 Gb Ethernet network. The platform appears logically as sixteen servers within one box. Each node is a Calxeda EnergyCore ECX-1000 comprising 4 ARM Cortex A9 cores and 4 GB of DRAM running Ubuntu 12.04 LTS. Viridis has a frequency of 1.4GHz in performance mode and a frequency of 200MHz in powersave mode.

Note, when referring to the different platform settings we will be using the following notation to represent the platform configuration [Nodes used $\times$ Cores Used $\times$ Threads per Core].

5.5.3 Methodology

In this section we provide details of our proposed experimental and measurements methodology, which are both critical for fair comparison between our platforms.

Experiment Methodology

We collected Facebook stock price ticks during a full New York Stock Exchange session and replayed them using UDP multicast to all nodes in each platform, as shown in Figure 5.2. This is as close as an experiment needs to be to reality without any external glitches or factors affecting the setup or measurements. We used `libevent` to capture the event of an option price changing, then the OptionPricer to calculate new prices for 617 Facebook options at the maximum speed feasible. It is worth noting that `libevent` is only used for its capacity to trigger option pricing events throughout this work.

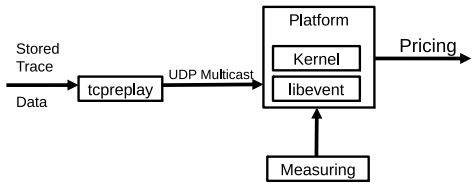


Figure 5.2 Measurement setup using trace data

Power and Energy Measurements

Each measurement path exhibits different characteristics [74]. The Power supply unit (PSU) converts the AC wall socket supply to DC, but can be up to 30% inefficient. The voltage regulator module (VRM) stabilizes the DC supply before it reaches the CPU. The exact form of the current supply path differs from one platform to the next but to provide a fair basis

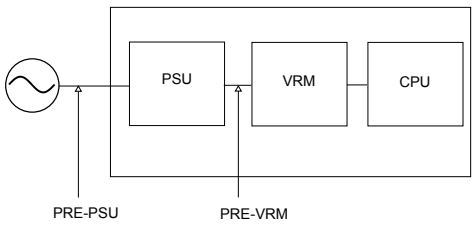


Figure 5.3 The path of the current supply to the CPU, showing power measurement points

for comparison we identified two distinct points on the path which are measurable on both platforms.

Measuring the power at the point before the PSU, which we label PRE-PSU, gives a value that corresponds to the true economic cost for operating each platform. This would give an overall picture of the external energy budget used by the different platforms. However, the energy consumed internally by the compute cores is also relevant to our study, as it isolates the energy effects of processors and discards artifacts of cooling and packaging, the study of which is beyond the scope of this work.

To isolate the energy consumption of processor packages, we capture power consumption at the point before the VRM, which we label PRE-VRM. For the Intel server, PRE-VRM measurement is facilitated by reading the Running Average Power Limit (RAPL) counters while the same functionality on Viridis is available through the Intelligent Platform Management Interface (IPMI) counters, which is also available on the Xeon Phi platform.

Figure 5.4 shows the power versus time plot for a standalone execution of the MC kernel. The BT execution plot is similar.

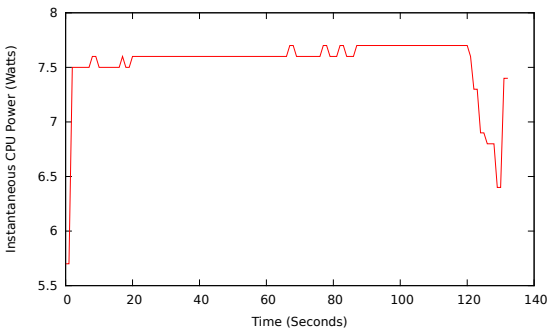


Figure 5.4 CPU power vs. time for the MC kernel

The profile of instantaneous power versus time follows a very sharp trapezoidal shape: the CPU is fully utilized during execution and there are no periods of inactivity. This is a common feature with other numerically intensive HPC applications. It means that the

measured average power is a representative measure of energy consumption throughout kernel execution.

In each experiment with multiple cores or nodes, we measure the worst execution time across the cores and use it as the elapsed time for all pricings. Each experiment was repeated at least three times to compute the mean kernel execution execution time and mean power consumption, so that the standard error of those means is below 5%. Energy consumption for a kernel is the product of the average power and the worst case elapsed time for the kernel execution, under the realistic assumption that kernels keep processors constantly busy while pricing options.

5.6 Fair Comparison of Servers and Micro-servers

In this section we are using our workload-specific and platform-independent metrics to directly compare the three server platforms. Detailed results are provided in Table 5.1. Results include comparisons using the vectorized or non-vectorized version of each kernel that achieves the fastest time per option priced on each platform. Furthermore, there are three sets of experiments for each kernel on all platform, with varying number of convergence iterations to reflect the different margins of error from the ideal analytical (Black Scholes) pricing solution. Trading accuracy for execution speed emulates realistic market scenarios. We also report PRE-VRM power measurements on all platforms.

5.6.1 Monte Carlo Pricing

A single Viridis micro-server underperforms a single Sandy Bridge socket in the MC kernel by one order of magnitude, with both running in performance mode. Among the many microarchitectural difference of the two platforms, our experiments suggest that the lack of efficient hardware implementation of transcendental functions on the ARM Cortex A9 processors is the critical culprit. That said, the Sandy Bridge socket expends almost six times as much power as the single micro-server.

Taking the MC kernel with one million iterations as an example, we observe that counter-balancing low performance with low lower consumption yields a modest 18% energy loss to price a single option on the Viridis. Notably, scaling out with the Viridis micro-server achieves a near ideal fifteen-fold terms of time per option priced, with sustained energy consumed per option, as power also scales linearly to the number of micro-server nodes. Scaling out from one to two Sandy Bridge sockets yields a non-ideal speedup of 1.63, while power consumption doubles. This narrows the gap in energy consumed per option to under

3% between the Sandy Bridge (better) and the Viridis (worse) when scaling out. Sixteen ARM micro-server nodes outperform two Sandy Bridge sockets in time per option, but incur an additional power tax of approximately 15 Watts. Running the kernel with more or less than a million iterations indicates similar performance and energy-efficiency trends. The gap in energy per option priced between the Viridis micro-servers and the Sandy Bridge in scale out setup is a marginal 1.4%.

The Xeon Phi server consumes more than two times less energy per option in the Monte Carlo kernel compared to both Viridis and the Sandy Bridge server. Notably, the Xeon Phi does so without employing vectorization and with a significant boost in performance and energy-efficiency from hyperthreading on each core. Taking the case with one million iterations as an example (other cases behave similarly), the Phi with all cores and threads activated provides a seemingly modest 34% performance improvement over the Viridis (even more compared to Sandy Bridge) but offers a significantly better power-efficiency proposition with 60 cores and 240 threads burning under 60 Watt, when the scaled out Viridis micro-servers offers 64 cores at just over 100 Watt.

Conclusion: For a real-time option pricing workload using Monte Carlo methods, the Xeon Phi is the best performing and most energy-efficient option. A scaled out ARM micro-server is on par with a heavyweight Sandy Bridge server in terms of energy cost, but can provide higher performance.

5.6.2 Binomial Tree

The lesser dependence on the BT kernel on transcendental functions becomes apparent from single-node experiments: A single micro-server is 5.5 times slower than a single Sandy Bridge socket (vs. 7.5 times in the MC kernel). A single Phi core is 2.5 times faster than a single ARM Cortex A9 core thanks to the higher quality vectorization of the BT kernel on the Phi and the more powerful SIMD set available, compared to the A9.

Scaling out the micro-server yields dramatically lower energy consumption per option priced than the Sandy Bridge server. For example, running the BT kernel with 7000 iterations consumes 81% less Joules per option on micro-servers compared to the Sandy Bridge. Sixteen micro-servers also outperform two Sandy Bridge sockets by as much as 46%. Note that in scale out mode the power consumption of both the Viridis and the Sandy Bridge servers are comparable. This surprising result – considering the difference in single-core performance between the two platforms – can be explained by workload effects. The Viridis server demonstrates the fact that the BT model has two computational phases. The first phase performs setup computations in linear complexity, involving transcendental mathematical

Table 5.1 Fastest S_{opt} profiles for standalone kernel experiments

Kernel and Platform	N	VEC TYPE	PRE-VRM $\bar{P}$(W)	S_{opt}	J_{opt}
MC Viridis(1x4x1)	0.5M	INTRINSICS	7.137	0.053298	0.380382
	1.0M	NEON128	7.258	0.105496	0.765691
	2.0M	NEON128	7.354	0.210560	1.548440
MC Viridis(16x4x1)	0.5M	NEON128	101.930	0.003761	0.383398
	1.0M	INTRINSICS	102.680	0.007205	0.739765
	2.0M	NEON128	103.100	0.014316	1.476006
MC Intel(1x8x1)	0.5M	AUTOVECT	44.899	0.007138	0.320485
	1.0M	SSE128	44.852	0.013896	0.623238
	2.0M	SSE128	45.078	0.027672	1.247364
MC Intel(2x8x1)	0.5M	SSE128	87.814	0.004331	0.379259
	1.0M	NOVECT	90.999	0.008274	0.753282
	2.0M	AUTOVECT	96.691	0.016213	1.568116
MC Xeon Phi(1x60x1)	0.5M	KNC512	48.690	0.004589	0.223363
	1.0M	AUTOVECT	52.022	0.008925	0.464319
	2.0M	AUTOVECT	54.065	0.017400	0.940726
MC Xeon Phi(1x60x2)	0.5M	NOVECT	51.821	0.003582	0.185607
	1.0M	NOVECT	55.246	0.006720	0.371162
	2.0M	NOVECT	57.632	0.012860	0.741166
MC Xeon Phi(1x60x4)	0.5M	NOVECT	55.087	0.002967	0.163309
	1.0M	NOVECT	59.288	0.005348	0.317089
	2.0M	NOVECT	62.804	0.010188	0.639805
BT Viridis(1x4x1)	4000	NEON128	6.246	0.007143	0.044611
	5000	NEON128	6.317	0.011250	0.071070
	7000	INTRINSICS	7.063	0.022279	0.157352
BT Viridis(16x4x1)	4000	NEON128	93.017	0.000649	0.060332
	5000	NEON128	93.783	0.000973	0.091239
	7000	INTRINSICS	98.650	0.001702	0.167785
BT Intel(1x8x1)	4000	INITRINSICS_AVX256	40.494	0.001302	0.052729
	5000	INITRINSICS_AVX256	45.754	0.002222	0.101594
	7000	AVX256	48.642	0.005192	0.252550
BT Intel(2x8x1)	4000	AVX256	86.159	0.000710	0.060584
	5000	INITRINSICS_AVX256	80.621	0.001464	0.118039
	7000	INITRINSICS_AVX256	95.177	0.003192	0.303158
BT Xeon Phi(1x60x1)	4000	INTRINSICS	23.833	0.000387	0.009229
	5000	INTRINSICS	25.417	0.000617	0.015824
	7000	KNC512	29.000	0.000745	0.021609
BT Xeon Phi(1x60x2)	4000	INTRINSICS	26.583	0.000401	0.010666
	5000	KNC512	29.861	0.000466	0.013968
	7000	KNC512	39.238	0.000731	0.028691
BT Xeon Phi(1x60x4)	4000	INTRINSICS	24.167	0.000544	0.013149
	5000	AUTOVECT	26.389	0.000586	0.015522
	7000	INTRINSICS	30.917	0.000891	0.027531

functions, which are accelerated by Intel hardware. The second phase is the tree scanning phase, which has quadratic complexity, using only floating point addition and multiplication operations. These operations execute natively on both platforms. While the Sandy Bridge has an advantage over the ARM on the initialization phase, this advantage diminishes in the computation phases and is marginalized as the number of iterations of the BT kernel grows.

The Xeon Phi offers once again an excellent power-efficiency proposition. Compared to the Viridis micro-servers in scale out setup, the Phi achieves twice the performance, at a third of the power budget, for a sixfold reduction of energy per option priced.

Conclusion: For a real-time option pricing workload using Binomial Tree methods, the Xeon Phi is the best performing and most energy-efficient option. A scaled out ARM micro-server is significantly faster and more energy-efficient than a Sandy Bridge server with equivalent power consumption.

5.6.3 QoS Discussion

In a live market situation the question arises whether option pricing can keep pace with the rate of arrival of new stock prices. We use the QoS metric defined in Section 5.5 to consider the implications of dynamic deadlines set by market conditions on the option pricing workload. We define an *iso-QoS* comparison method, where we compare the performance, power and energy of different platforms, with setups that achieve the same QoS for a given workload. One can conduct similar exercises using iso-metrics that equate power, performance, energy consumption, or combined efficiency metrics.

When a new stock price arrives, this event triggers a computation of all available option contracts. The most stringent QoS requirement is “all-or-nothing": the pricing of all options must complete before another stock price update, otherwise any computation performed before the deadline is discarded, having wasted time and energy. Indicatively, Figure 5.5 shows the percentage of successful all-or-nothing option pricing computations as a function of the cumulative number of price updates, sorted into bins of 0.25 second intervals. The data for our experiments are taken from a trading session of 6.5 hours where 10,156 price updates occurred for the Facebook stock, which results in the cumulative profile shown in the figure.

Figure 5.5 shows the QoS profile of the Viridis and Sandy Bridge platforms with two scenarios of Monte Carlo (half a million iterations) and Binomial Tree (4000 iterations). Both platforms run in performance mode and are scaled to the maximum number of cores available. The figure shows the performance gap between options pricing on the two platforms. The micro-server is able to price all options using Monte Carlo slightly faster ($[2.25-2.50]$ seconds time bin), while the Sandy Bridge is slower ($[2.50-2.75]$ seconds bin) but both

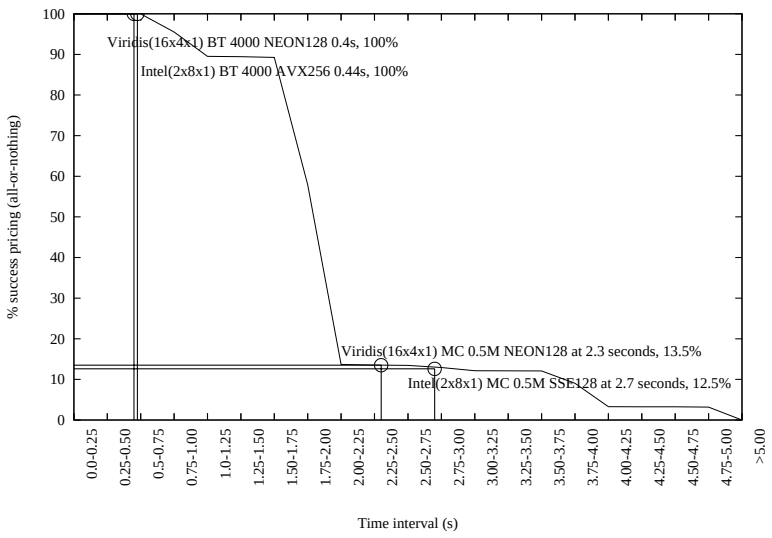


Figure 5.5 All-or-nothing pricing vs. stock price update intervals

platforms exhibit low QoS (13.5% and 12.5% respectively), which may be unacceptable for an end user. On the other hand, both platforms exhibit 100% QoS with the Binomial Tree kernel, where the Viridis micro-servers also reduce total energy consumption by as much as 55%.

In the next section we elaborate on the QoS metric and on the iso-QoS comparison methodology we devise to rank servers based on energy efficiency.

5.7 The Mathematical Basis of the QoS Metric

Many of the world leading financial trading venues are order driven markets. This means that investors, especially high frequency traders, independently submit buy and sell orders to matching engine software that needs to operate at high speed. These engines cross buy and sell orders to create trades and are a key part of the electronic trading platforms which underpin high frequency trading. Sequential models, which are the basis to analyze trading patterns in high frequency trading, assume a Poisson distribution [24] to model the arrival of orders affecting stock price into the system.

5.7.1 The QoS Cumulative Frequency Distribution

In this section we explain how we create a QoS curve as a function of the inter-arrival time intervals of stock price updates. Inter-arrival time intervals between successive stock updates are commonly referred to as *time gaps*. It is important to note that the QoS curve of a stock is solely dictated by the market activity, reflected to the frequency distribution of the time gaps of this particular stock, on a specific trading day. In the next section we explain how we can determine whether a given platform can meet the required QoS value or not, by using the S_{opt} ad J_{opt} metrics of this platform in conjunction with QoS curves.

From our data, we created a histogram of the distribution of time gaps between price updates for the Facebook stock. Further processing these data, we compute the cumulative frequency distribution (CFD) which indeed exhibits the characteristics of a Poisson CFD. This reflects the assumptions of the sequential model of financial trading.

Normally in a CFD the value assigned to bin i is the sum of all values in bins $1,\ldots,i$. In our case these are time bins, so that frequency refers to the number of price updates arriving at time intervals up to and including that represented by bin i. There is a value of the time gap, below which it is not possible to satisfy the hard constraint of computing prices for all defined options. We denoted this by G. This value depends on the performance of the platform, the number of options to be priced and the particular financial kernel used. Our QoS metric actually corresponds to the sum over all time bins greater than G. It follows that our QoS function is obtained by reflecting the initial CFD around its mid-point on the time axis. This means that we can fit our observed time gap distribution to the form

$$\mathrm{QoS(t)} = 1 - e^{-\lambda} \sum_{i=0}^{t} \frac{\lambda^t}{\lfloor t! \rfloor} \tag{5.8}$$

Furthermore, we define the QoS as the percentage of successful all-or-nothing pricings over the total number of pricings triggered by a price update.

The data for our experiments are taken from a trading session of 6.5 hours where $10,156$ price updates occurred for the Facebook (FB) stock, resulting in the cumulative distribution function representing the QoS shown in figure 5.6. The solid line shows the measured values while the dashed curve shows the result of fitting the measured data to the analytic expression for the cumulative Poisson distribution. Further confirmation of the Poisson-like behavior of the arrival of price updates is seen in the profile for the Google stock which is also presented in figure 5.6. Similar price update profiles occur in work [68] studying prices on the German DAX exchange.

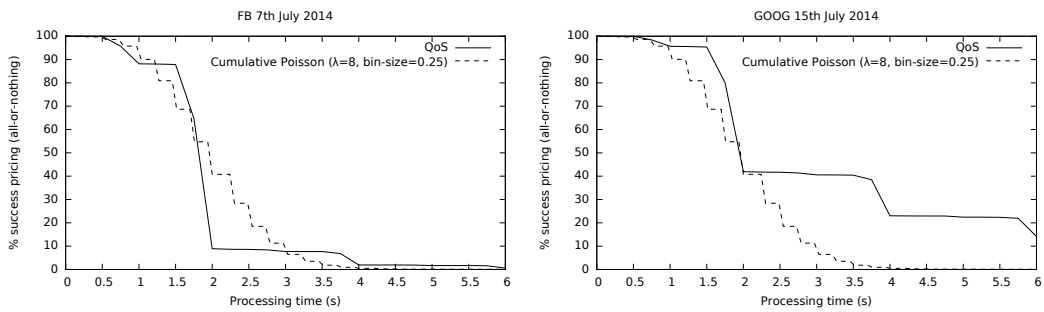


Figure 5.6 Cumulative frequency distribution of Facebook and Google stock price updates for full trading sessions on July 7th and 15th 2014

5.7.2 Iso-QoS and Total Energy Consumption

Let us set a required QoS $Y\%$ for all our platforms. From the QoS curve we can determine a minimum time constraint, G, that we must satisfy. Within G seconds we need to compute all N_{opt} options defined on the stock. First of all a platform can only satisfy this constraint if

$$G \geq N_{\text{opt}} \times S_{\text{opt}} \tag{5.9}$$

Assuming this is met, we know that the energy consumed in each time gap is then

$$E_{\text{gap}} = N_{\text{opt}} \times J_{\text{opt}} \tag{5.10}$$

where we ignore idle power. Next, we know from the definition of QoS that the total number of time gaps in which we will perform the computation is

$$N_{\text{gaps}} = \text{floor}(Y \times \text{Total number of updates for the session}) \tag{5.11}$$

so that the energy consumed doing option pricing while meeting QoS $Y\%$ is

$$E_{\text{QoS=Y}} = N_{\text{gaps}} \times E_{\text{gap}} \tag{5.12}$$

Platforms may then be ranked, for this QoS, in order of energy consumption.

5.7.3 Application to Platforms

We have applied the equations defined above using the QoS curve in figure 5.6. Table 5.2 is the result of the analysis of delivering option pricing with a 10% QoS using the MC kernel operated with 0.5M iterations. Only the five cases (platform plus software) which

can satisfy the constraint in equation (5.9) are reported. We noted that at 50% QoS, none

Table 5.2 MC kernel (N=0.5M and QoS=10%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Viridis(16×4×1)	INTRINSICS	0.0038	0.3830	239.85
Intel(2×8×1)	AUTOVECT	0.0044	0.3794	237.58
Xeon Phi(1×60×1)	KNC512	0.0046	0.2234	139.92
Xeon Phi(1×60×2)	NOVECT	0.0036	0.1856	116.26
Xeon Phi(1×60×4)	INTRINSICS	0.0030	0.1584	99.19

of our platform/software combinations could satisfy the constraint in equation (5.9). We have commented on this characteristic previously [34] explaining that it means only that a subset of all available options can be priced, but not the full set. The MC kernel involves relatively expensive evaluation of the natural logarithm in the Box Muller transform and the exponential function to compute the option price.

We repeated the analysis with the BT kernel, which is dominated by multiply add operations, and report results for QoS values of 80% and 40% in tables 5.3 - 5.8.

Table 5.3 BT kernel (N=4000 and QoS=80%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	AVX256	0.0007	0.0611	306.49
Viridis(16×4×1)	NEON128	0.0006	0.0603	302.41
Intel(1×8×1)	INTRINSICS	0.0013	0.0527	264.32
Xeon Phi(1×60×4)	INTRINSICS	0.0005	0.0131	65.88
Xeon Phi(1×60×2)	INTRINSICS	0.0004	0.0107	53.50
Xeon Phi(1×60×1)	INTRINSICS	0.0004	0.0092	46.27

In figure 5.7 we show how energy of the scaled out configurations varies with the number of points used. We are comparing Viridis(16×4×1) to Intel(2×8×1) and show how the Viridis can actually outperform Intel's Sandy Bridge while provisioning for an 80% QoS.

5.8 Chapter Conclusion

In this chapter we have presented a fair methodology for comparing server platforms for real-time financial analytics. Our methodology considers performance, energy-efficiency, and

Table 5.4 BT kernel (N=5000 and QoS=80%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	INTRINSICS	0.0015	0.1180	591.65
Intel(1×8×1)	INTRINSICS	0.0022	0.1017	509.69
Viridis(16×4×1)	INTRINSICS	0.0010	0.0912	457.05
Xeon Phi(1×60×1)	INTRINSICS	0.0006	0.0157	78.58
Xeon Phi(1×60×4)	INTRINSICS	0.0006	0.0152	76.23
Xeon Phi(1×60×2)	KNC512	0.0005	0.0139	69.76

Table 5.5 BT kernel (N=7000 and QoS=80%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	INTRINSICS	0.0032	0.3038	1522.85
Viridis(16×4×1)	INTRINSICS	0.0017	0.1679	841.83
Xeon Phi(1×60×2)	AUTOVECT	0.0007	0.0281	140.84
Xeon Phi(1×60×4)	INTRINSICS	0.0009	0.0275	138.02
Xeon Phi(1×60×1)	KNC512	0.0007	0.0216	108.28

Table 5.6 BT kernel (N=4000 and QoS=40%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	AVX256	0.0007	0.0611	153.24
Viridis(16×4×1)	NEON128	0.0006	0.0603	151.21
Intel(1×8×1)	INTRINSICS	0.0013	0.0527	132.16
Xeon Phi(1×60×4)	INTRINSICS	0.0005	0.0131	32.94
Xeon Phi(1×60×2)	INTRINSICS	0.0004	0.0107	26.75
Xeon Phi(1×60×1)	INTRINSICS	0.0004	0.0092	23.13

Table 5.7 BT kernel (N=5000 and QoS=40%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	INTRINSICS	0.0015	0.1180	295.82
Intel(1×8×1)	INTRINSICS	0.0022	0.1017	254.85
Viridis(16×4×1)	INTRINSICS	0.0010	0.0912	228.52
Xeon Phi(1×60×1)	INTRINSICS	0.0006	0.0157	39.29
Xeon Phi(1×60×4)	INTRINSICS	0.0006	0.0152	38.11
Xeon Phi(1×60×2)	KNC512	0.0005	0.0139	34.88

Table 5.8 BT kernel (N=7000 and QoS=40%)

Platform	**VEC TYPE**	S_{opt}	J_{opt}	**Energy(KJ)**
Intel(2×8×1)	INTRINSICS	0.0032	0.3038	761.42
Intel(1×8×1)	AVX256	0.0052	0.2526	632.95
Viridis(16×4×1)	INTRINSICS	0.0017	0.1679	420.92
Xeon Phi(1×60×2)	AUTOVECT	0.0007	0.0281	70.42
Xeon Phi(1×60×4)	INTRINSICS	0.0009	0.0275	69.01
Xeon Phi(1×60×1)	KNC512	0.0007	0.0216	54.14

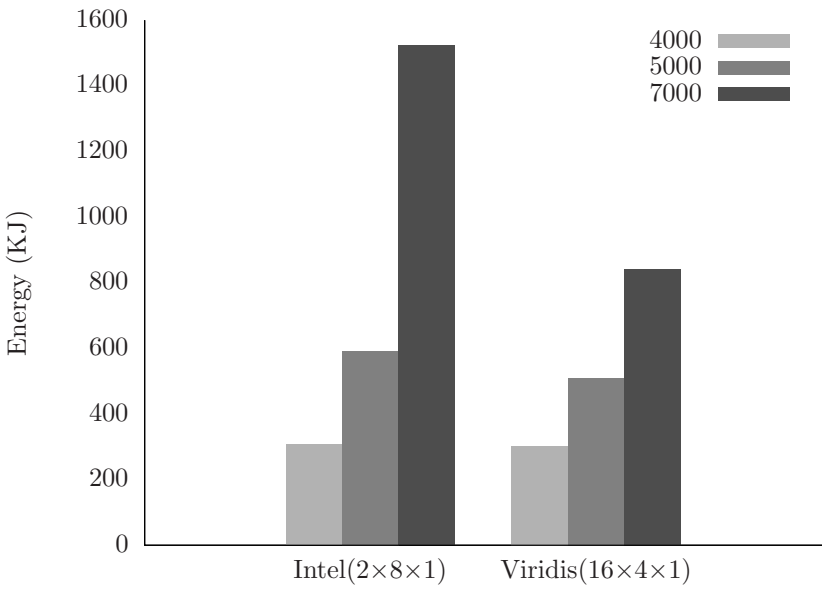


Figure 5.7 BT kernel energy consumption scaling (at QoS=80%) of Viridis(16×4×1) and Intel(2×8×1)

QoS impact on energy consumption and hence the cost of provisioning these services. We have based our methodology on workload-specific but platform-independent metrics, which facilitated direct performance and cost comparisons. These comparisons benefit directly datacenter operators during hardware procurement, but also developers of financial services during exercises to reduce service cost while sustaining competitive performance and QoS in their products. Moreover, they can guide dynamic resource allocation policies for meeting performance targets while achieving energy efficiency by selecting the most energy efficient but QoS compliant resources across heterogeneous servers. Notably, our study used real stock market streaming data and captured the dynamic, event-driven nature of real-time financial analytics workloads.

Our results show that micro-servers based on ARM processors are viable contenders to state-of-the-art general-purpose servers. Scaled out micro-server configurations achieve similar or better performance and similar or improved energy-efficiency under a fixed power budget. However, micro-servers cannot match the densely packaged many-core accelerators, such as the Xeon Phi. Based on our experimental results with real market data for a trading session, micro-servers promise up to 45% less energy compared to a standard HPC server while providing the same quality of service.

Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

6.1.1 Dynamic Binary Adaptation on Shared-ISA Architectures

We have presented two new binary adaptation techniques for Shared-ISA architectures, that is Dynamic Binary Rewriting (DBR) and Dynamic Binary Translation (DBT). Our novel binary-level techniques enable both cross-core, transparent execution of unmodified binaries but also cross-core thread migration at any point in time. By contrast, Fault-And-Migrate (FAM), which is the state-of-the-art technique for transparent execution, imposes forced migrations that preclude applying any optimizing scheduling strategies and can lead to degraded performance. DBR and DBT are designed to be fast and lightweight by adapting the binary on-demand and with a reduced scope on code. Moreover, they do not require any change on the compilation framework or prior binary instrumentation and they operate transparently to the programmer, invoked by the OS as system services.

Specifically, DBR operates on binary code which has been statically targeted for the basic ISA. It retargets parts of the code to accelerating instructions when the host core ISA includes them. Conversely it can revert retargeted code to the original, basic instructions, if the thread executing this code section is moved to a basic core because of scheduling policy. DBR builds on techniques such as dynamic control flow discovery, indirect branch sampling and peephole analysis, enhancing them for Shared-ISA execution.

DBT complements DBR by operating on binaries which have been statically targeted for the accelerated ISA. In DBT we implement a novel *fault-and-rewrite* approach. The OS traps instruction execution faults when an accelerating instruction executes on a basic core and invokes DBT on-demand to replace this faulted instruction with a trampoline to an emulation routine implemented with basic instructions. DBT can revert back trampolines to

the original accelerating instructions when scheduling policy moves the thread executing this code section to a basic core.

We evaluated DBR and DBT compare with FAM. The purpose of this evaluation is to quantify the overhead of dynamic binary adaptation that provides increase flexibility through anytime cross-core thread migration compared to the state-of-the-art FAM approach which imposes forced migrations. Evaluation has shown that dynamic binary adaptation does not add significant overhead, on a periodic migration scenario DBR increases execution time by about 40% while DBT has less overhead about 10% compared with FAM. A case study of a Speedup Proportional Dynamic Scheduling (SPDS) policy on multi-program workloads shows that this overhead is recoverable by dynamic scheduling compared to a FAM scheduling policy. Specifically, SPDS with DBT executes workloads about 50% faster than FAM, while SPDS with DBR executes them about 20% faster.

6.1.2 Speedup Aware Dynamic Scheduling on Shared-ISA Architectures

We have presented a novel scheduling framework for Shared-ISA architectures (SPEEDSWAP) which enables speedup-aware dynamic scheduling policies on Shared-ISA architectures. Our framework builds on cross-core migration, possible through our Dynamic Binary Translation (DBT) method, and a novel instruction-level profiling methodology at runtime for speedup awareness. These mechanisms enable the design and implementation of informed dynamic scheduling policies to optimize the execution of multi-program workloads.

In the context of our framework, we developed a dynamic scheduler which targets to maximize workload speedup using a heuristic algorithm to swap threads periodically between basic and accelerating cores. The heuristic algorithm operates in epochs (an epoch is a collection of successive scheduling quantum intervals) during which it stores per-thread scheduling information, including the time each thread spent executing on an accelerating core and its average speedup. The scheduling algorithm runs decentralized on each accelerating core and periodically assesses possible thread swaps evaluating the projected acceleration using the speedup estimation technique. It swaps threads only if this swap maximizes the estimated workload speedup.

We evaluated our dynamic scheduler against a state-of-the-art dynamic scheduler based on Fault-And-Migrate (FAM) and experimented with different hardware configurations, altering the number of cores and the ratio of accelerating to basic cores. SPEEDSWAP enables an informed dynamic scheduling approach enable by on-demand cross-core migration and speedup estimation as opposed to any state-of-the-art FAM scheduler which is oblivious

to speedup and relies on forced migration that can cause over-subscription. Evaluation has shown that SPEEDSWAP outperforms FAM based dynamic scheduling and results in faster execution times the fewer accelerating cores are. Indicatively, SPEEDSWAP can results in up to 2.5× faster execution times than FAM when there is one accelerating core per four basic ones. Even in the most favorable configuration for FAM, where there is one accelerating core for each basic core, the SPEEDSWAP scheduler is never worse than FAM and can improves up to 15% the workload execution time.

6.1.3 Performance Characterization on Heterogeneous Datacenter Architectures

We have presented a rigorous methodology and a set of new iso-comparison metrics to evaluate the performance and energy efficiency of resource allocation across a diverse set of server platforms, including micro-servers and many-core platforms. We define platform-independent though workload specific metrics for fairly comparing those servers despite differences in deployment, architecture and power/performance monitoring capabilities. At the center of our methodology are *iso-comparisons* on performance and energy. Specifically, the Quality-Of-Service (QoS) metric, built using machine-level metrics, expresses performance in a platform-independent way and is applicable to a wide variety of workloads in the datacenter domain. Comparing the energy efficiency of machines for iso-QoS targets is a fair measure of performance as perceived by the service user, thus enables a fair assessment despite heterogeneity. These comparisons also benefit datacenter operators for hardware procurement and provisioning.

We apply our methodology on a real-time financial workload for option pricing, which implements a financial analytics service. We evaluate a variety of heterogeneous servers, including an ARM-based micro-server, a typical Intel Xeon machine and a Xeon Phi many-core platform which represents the future generation of server processors. Moreover, we experiment with different levels of code optimization and scale-up and scale-out execution. Our conclusions are that micro-servers offer a viable, more energy efficient alternative to general-purpose servers consuming up to 45% less energy, provided scale-out execution is possible. However high throughput, manycore platforms, such as the Xeon Phi architecture, present the best performance and energy efficiency proposition.

6.2 Future Work

There are several open research opportunities regarding our dynamic binary adaptation techniques. We intent to expand our techniques to include more classes of accelerating instructions, such as SIMD instructions, cryptographic and DSP extensions. Moreover, we are investigating binary rewriting techniques not only for accelerating instructions but also for refactoring binary code to adapt execution to resource sharing in the workload. In addition, we are researching the impact of instructions on power consumption and ways to affect it through rewriting.

Future work on profiling and cross-core speedup estimation entails extensions for modeling performance of ISA-transparent heterogeneity too, including differences in the pipeline architecture, memory hierarchy and clock frequency. Furthermore, we are working on extending speedup estimation on different architectural processing elements, such as disjoint-ISA cores, GPU and FPGA accelerators.

Our plans for extensions on our dynamic scheduling framework include factoring in application-level information information and power and energy constraints. Application-level information includes inter-thread synchronization and critical section execution on parallel workloads. Moreover, we are looking on leveraging parallel runtime information for programs with structured parallelism expressed through a parallel programming model such as OpenMP, Cilk and others. We are investigating dynamic scheduling strategies which for enforcing power capping and energy constraints alongside performance objectives for energy-aware execution. Dynamic scheduling will make use of this information to accelerate the workload and meet power or energy constraints.

On our work for defining methods and metrics to assess heterogeneous servers, we have identified several topics for future research. Our approach can accommodate the need of exact provisioning datacenters by factoring in the target QoS and energy consumption to find a configuration of heterogeneous servers and accelerators that meets the specifications. Furthermore, we are researching dynamic resource allocation policies for datacenters. By using our metrics and online load monitoring, a resource allocator can decide how to dynamically distribute the workload in a datacenter of heterogeneous servers to meet a target QoS reducing at the same time energy consumption.

Bibliography

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] Tilak Agerwala and Siddhartha Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, 25(3):58–69, May 2005.

[3] R. Allen and S. Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. *SIGPLAN Not.*, 23(7):241–249, jun 1988.

[4] Pedro Alonso, Manuel F. Dolz, Rafael Mayo, and Enrique S. Quintana-Ortí. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - R\&D*, 29(2):105–112, 2014.

[5] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through Epi Throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Krish K.R. Anwar A. and Butt A.R. On the use of microservers in supporting hadoop applications. In *Cluster Computing (CLUST), 2014 International Conference on*, pages 66–74, Sep 2014.

[7] O. Arnold and G. Fettweis. Power aware heterogeneous MPSoC with dynamic task scheduling and increased data locality for multiple applications. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 110–117, July 2010.

[8] S. Balakrishnan, R. Rajwar, M. Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 506–517, June 2005.

[9] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, May 2005.

[10] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.

[11] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *J. Political Econonomy*, 81:637–54.

[12] F. Black and M. Scholes. The valuation of option contracts and a test of market efficiency. *J. Finance*, 27:399–418.

[13] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power Struggles: Revisiting the RisC vs. CisC debate on Contemporary Arm and x86 Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 1–12, 2013.

[14] P. Bogdan, S. Garg, and U. Y. Ogras. Energy-efficient computing from systems-on-chip to micro-server and data centers. In *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, pages 1–6, Dec 2015.

[15] A. Branover, D. Foley, and M. Steinman. AMd Fusion Apu: Llano. *Micro, IEEE*, 32(2):28–37, 2012.

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, and Jan-Philipp Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurr. Comput. : Pract. Exper.*, 24(7):663–675, May 2012.

[18] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 131–139, Feb 2013.

[19] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. *SIGARCH Comput. Archit. News*, 40(3):225–236, June 2012.

[20] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload*

Characterization, 2009. IISWC 2009. IEEE International Symposium on, pages 44–54, 2009.

[21] Jingsheng Jason Cong. Accelerator-rich architectures: From single-chip to datacenters. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 139–140, New York, NY, USA, 2014. ACM.

[22] Gary Cook and J Van Horn. How dirty is your data. *A look at the energy choices that power cloud computing*, 2011.

[23] S. P. Crago and J. P. Walters. Heterogeneous cloud computing: The way forward. *Computer*, 48(1):59–61, Jan 2015.

[24] Claudia Czado and A. Kolbe. Empirical Study of Intraday Option Price Changes using extended Count Regression Models, 2004.

[25] M. Daga, A.M. Aji, and Wu-Chun Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149, July 2011.

[26] Christian de Schryver, Matthias Jung, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. Energy Efficient Acceleration and Evaluation of Financial Computations towards Real-Time Pricing. In Andreas Knig, Andreas Dengel, Knut Hinkelmann, Koichi Kise, RobertJ. Howlett, and LakhmiC. Jain, editors, *Knowledge-Based and Intelligent Information and Engineering Systems*, volume 6884 of *Lecture Notes in Computer Science*, pages 177–186. Springer Berlin Heidelberg, 2011.

[27] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[29] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution Migration in a heterogeneous-Isa Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM.

[30] Balaji Dhanasekaran and Kim Hazelwood. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, pages 11–18.

[31] J. Dongarra, H. Ltaief, P. Luszczek, and V.M. Weaver. Energy Footprint of Advanced Dense Numerical Linear Algebra Using Tile Algorithms on Multicore Architectures. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 274–281, Nov 2012.

[32] Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, and Manuel Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12):48–57, December 2009.

[33] E. Flamand. Strategic directions towards multicore application specific computing. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1266–1266, April 2009.

[34] Charles J Gillan, Dimitrios S. Nikolopoulos, Giorgis Georgakoudis, Richard Faloon, George Tzenakis, and Ivor Spence. On the Viability of Microservers for Financial Analytics. In *Proceedings of the 7th Workshop on High Performance Computational Finance*, WHPCF '14, pages 29–36, Piscataway, NJ, USA, 2014. IEEE Press.

[35] G. Gnanajothi. Power and Cooling the Evolving Data Center Infrastructure. May 2014.

[36] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, aug 1993.

[37] R.E. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[38] Peter Greenhalgh. Big. little processing with arm cortex-a15 and cortex-a7. *ARM White Paper*, 2011.

[39] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *Proceedings of the IEEE International Conference on Computer Design*, ICCD '04, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Michael Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, June 2007.

[41] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Market mechanisms for managing datacenters with heterogeneous microarchitectures. *ACM Trans. Comput. Syst.*, 32(1):3:1–3:31, February 2014.

[42] Linley Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.

[43] James Hamilton. Internet-scale service infrastructure efficiency. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 232–232, New York, NY, USA, 2009. ACM.

[44] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33:63–68, December 2005.

[45] John L. Henning. SPec Cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, sep 2006.

[46] Nicholas J. Higham. The Accuracy Of Floating Point Summation. *SIAM J. Sci. Comput*, 14, 1993.

[47] Mark D. Hill. Amdahl's Law in the multicore era. In *HPCA*, page 187, 2008.

[48] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[49] Jason D. Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Trans. Archit. Code Optim.*, 8:9:1–9:28, June 2011.

[50] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang. Cost effective data center servers. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 179–187, Feb 2013.

[51] S. A. Ross J. C. Cox and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7:779.

[52] S. Jain, H. Navale, U. Ogras, and S. Garg. Energy efficient scheduling for web search on heterogeneous microservers. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 177–182, July 2015.

[53] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[54] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based acceleration of multithreaded applications on asymmetric Cmps. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 154–165, New York, NY, USA, 2013. ACM.

[55] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. AAsh: an asymmetry-aware scheduler for hypervisors. *SIGPLAN Not.*, 45(7):85–96, mar 2010.

[56] Vijay Anand Korthikanti and Gul Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 157–165, New York, NY, USA, 2010. ACM.

[57] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[58] Christos Kozyrakis. Resource Efficient Computing for Warehouse-scale Datacenters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1351–1356, 2013.

[59] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, Nov 2005.

[60] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-Isa Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[61] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, November 2005.

[62] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-Isa Heterogeneous Multi-Core Architectures for Multithreaded

Workload Performance. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[63] Viren Kumar and Alexandra Fedorova. Towards better performance per watt in virtual environments on asymmetric single-isa multi-core systems. *SIGOPS Oper. Syst. Rev.*, 43(3):105–109, July 2009.

[64] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-Isa heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[65] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[66] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.

[67] P. Linz. Accurate Floating Point Summation. *Numerical mathematics*, 13:361–2.

[68] Xiaoyan Liu, Xindong Wu, Huaiqing Wang, Rui Zhang, James Bailey, and Kotagiri Ramamohanarao. Mining distribution change in stock order streams. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 105–108, 2010.

[69] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[70] L. Lugini, V. Petrucci, and D. Mosse. Online thread assignment for heterogeneous multicore systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 538–544, Sept 2012.

[71] R. P. Luijten and A. Doering. The dome embedded 64 bit microserver demonstrator. In *IC Design Technology (ICICDT), 2013 International Conference on*, pages 203–206, May 2013.

[72] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[73] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 317–328, Washington, DC, USA, 2012. IEEE Computer Society.

[74] I. Manousakis and D. S. Nikolopoulos. BTl: A framework for Measuring and Modeling Energy in Memory Hierarchies. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 139–46.

[75] J. Mars, L. Tang, and R. Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, July 2011.

[76] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM.

[77] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30.

[78] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded Socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1137–1142, New York, NY, USA, 2012. ACM.

[79] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Erik Hochapfel, and Sandrine Vaton. Energy-efficient FpgA implementation for Binomial Option Pricing

Using OpenCl. In *Proceedings of the Conference on Design, Automation \& Test in Europe*, DATE '14, pages 208:1–208:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[80] S. L. Moshier. Cephes Math Library, 2000. See http://www.moshier.net.

[81] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, Apr 2001.

[82] Sandeep Navada, Niket K. Choudhary, Salil V. Wadhavkar, and Eric Rotenberg. A unified View of Non-monotonic Core Selection and Application Steering in Heterogeneous Chip Multiprocessors. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 133–144, Piscataway, NJ, USA, 2013. IEEE Press.

[83] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, March 2010.

[84] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NumA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, 2014.

[85] NVIDIA. NVIDIA Tegra K1: A new era in mobile computing. *NVIDIA White Paper*, 2013.

[86] Zhonghong Ou, Bo Pang, Yang Deng, Jukka K. Nurminen, Antti Yla-Jaaski, and Pan Hui. Energy- and Cost-Efficiency Analysis of Arm-Based Clusters. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 115–123, 2012.

[87] M. Broadie P. Boyle and P Glasserman. Monte Carlo methods for security pricing. *J. Econ. Dynamics and Control*, 21:1267–321.

[88] Michael P Perrone and Tanaz Sowadagar. Cell Be software programming and toolkits. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[89] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y.

Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[90] Nikola Rajovic, Paul M. Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity cpus: Are mobile socs ready for hpc? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 40:1–40:12, New York, NY, USA, 2013. ACM.

[91] Dheeraj Reddy, David Koufaty, Paul Brett, and Scott Hahn. Bridging functional heterogeneity in multicore architectures. *SIGOPS Oper. Syst. Rev.*, 45(1):21–33, feb 2011.

[92] Dheeraj Reddy, David A. Koufaty, Paul Brett, and Scott Hahn. Bridging functional heterogeneity in multicore architectures. *Operating Systems Review*, 45(1):21–33, 2011.

[93] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[94] Cristina Rottondi. Greenpeace report "clicking green: A guide to building the green internet".

[95] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Trends in server energy proportionality. *Computer*, 44(9):69–72, Sept 2011.

[96] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging Core Specialization via Os Scheduling to Improve Performance on Asymmetric Multicore Systems. *ACM Trans. Comput. Syst.*, 30(2):6:1–6:38, apr 2012.

[97] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

[98] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, 71(1):114–131, January 2011.

[99] Lina Sawalha, Sonya Wolff, Monte P. Tull, and Ronald D. Barnes. Phase-Guided Scheduling on Single-Isa Heterogeneous Multicore Processors. In *Proceedings of the 2011 14th Euromicro Conference on Digital System Design*, DSD '11, pages 736–745, Washington, DC, USA, 2011. IEEE Computer Society.

[100] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, April 2009.

[101] Shuaiwen Song, Matthew Grove, and Kirk W. Cameron. An Iso-Energy-Efficient Approach to Scalable System Power-Performance Optimization. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 262–271, Washington, DC, USA, 2011. IEEE Computer Society.

[102] Shuaiwen Song, Chun-Yi Su, Rong Ge, Abhinav Vishnu, and Kirk W. Cameron. Iso-Energy-Efficiency: An Approach to Power-Constrained Parallel Computation. In *Proceedings of the 2011 IEEE International Parallel \& Distributed Processing Symposium*, IPDPS '11, pages 128–139, Washington, DC, USA, 2011. IEEE Computer Society.

[103] P. Stanley-Marbell and V. C. Cabezas. Performance, power, and thermal analysis of low-power processors for scale-out systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 863–870, May 2011.

[104] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[105] Bogdan Marius Tudor and Yong Meng Teo. On Understanding the Energy Consumption of Arm-based Multicore Servers. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 267–278, 2013.

[106] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through Performance Impact Estimation (Pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[107] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[108] Daniel Wong and Murali Annavaram. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 119–130, Washington, DC, USA, 2012. IEEE Computer Society.

[109] Biwei Xie, Xu Liu, Jianfeng Zhan, Zhen Jia, Yuqing Zhu, Lei Wang, and Lixin Zhang. Characterizing data analytics workloads on intel xeon phi. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 114–115, Oct 2015.

[110] OS Xilinx. Libraries document collection (ug 643).

[111] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.