Maximizing Heterogeneous Processor Performance under Power Constraints

Optimale prestatie op heterogene processors onder een vermogensbudget

Almutaz Adileh

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2017 - 2018

ISBN 978-94-6355-107-6
NUR 980, 987
Wettelijk depot: D/2018/10.500/25

Examination Committee

Prof. Luc Taerwe, *voorzitter*
Prodecaan Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Lieven Eeckhout, *promotor*
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Koen De Bosschere, *secretaris*
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Stijn Eyerman
Intel Labs
Belgium

Dr. Aamer Jaleel
Nvidia Research
USA

Prof. Jan Fostier
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Stefanos Kaxiras
Uppsala University, Department of Information Technology
Sweden

Reading Committee

Prof. Koen De Bosschere
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Stijn Eyerman
Intel Labs
Belgium

Dr. Aamer Jaleel
Nvidia Research
USA

Prof. Jan Fostier
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Stefanos Kaxiras
Uppsala University, Department of Information Technology
Sweden

Acknowledgements

I have delayed writing this part to the last minute. My PhD journey was a long one and writing this part did not seem happening at times. Perhaps I did not want to start writing it because I know that there is so much I have to say that cannot be conveyed in few words. Perhaps I was not able to find a proper way to thank all the great people who helped me persevere throughout this journey in parallel, without any order. These are few words to reflect my sincere thanks.

I would like to thank my parents, Amin Adileh and Fatima Abbasi. The two greatest humans I have met in my life. Time and time again, they happily put their lives on hold to pave the path for me and my siblings to find our way in life. After having a PhD, I may still be their least achieving kid. I wouldn't be writing these words without your unconditional support. There is no way I can repay you, but I hope to be a source of happiness for you as long as I live. During my PhD journey, Rana Dwayyat entered my life to bless it. She kept holding my hand and never let go. A pure heart with non-fading love. She gave me my first son Muhammad, who makes me want to be the best version of myself. I hope I will be a source of pride for you baba, and your best friend one day. No matter where we live, I pray that the future does not separate us.

I had the privilege of collaborating and rubbing shoulders with brilliant researchers. I will always be indebted to my advisor, Lieven Eeckhout. Had he not given me this chance and believed in me throughout my years at Ghent, I may have never reached the end of the tunnel. He is an inspiration to me as a researcher and as a great human. I cannot thank Stijn Eyerman and Aamer Jaleel enough for all the help they have provided me with. Their constant supply of ideas and help in formulating solutions was instrumental for the completion of this work.

I would like to thank the members of my examination committee. They provided me with invaluable feedback and suggestions to improve this work despite their packed schedules. I would like to thank them for taking time

to come to Ghent University and discuss the various aspects of this work. They have enriched my PhD examination experience and opened my eyes to various aspects worth of future exploration.

I met exceptional people and made formidable friendships throughout the various stages of my PhD. All of them helped by brainstorming, exchanging ideas, providing feedback, and being exceptional friends. Jennifer Sartor, Shoaib Akram, Sam Van den Steen, Sander De Pestel, Cecilia González-Álvarez, Kristof Du Bois, Wim Heirman, Trevor Carlson, Kenzo Van Craeynest, Maximilien Breughe, Xia Zhao, Ajeya Naithani, Kartik Lakshminarasimhan, Lu Wang, Yuxi Liu, Wenjie Liu, Josué Feliu, Pejman Lotfi-Kamran, Sotiria Fytraki, Stavros Volos, Onur Kocberber, Djordje Jevdjic, Cansu Kaynak, Javier Picorel, Mike Ferdman, Alisa Yurovsky, Evangelos Vlachos, Alexandros Daglis, Babak Falsafi, Boris Grot, David Lilja, Cristina Ghiurcuta, Nancy Chong, Qasem Ramadan, and Mohammad Murib. Thank you all.

Last but not least, I would like to thank all the people who constantly cheer for me. My brothers Muhammad and Maen and their families, my sister Lana and her family, my uncles and aunts, my cousins, my in-laws, and all my friends around the globe. I thank Allah for blessing me more than I deserve, with every breath I take and every decision I make. I will always strive to improve and I aspire to make this a step towards further success.

Gent, 9th April 2018
Almutaz Adileh

Summary

Technology scaling trends have forced processor designers into an era with new design constraints and challenges. The combination of Moore's law and Dennard scaling that enabled leaps of innovation in the computing industry over the past few decades has come to an end. Although the number of transistors available on a chip keeps growing as predicted by Moore's law, scaling them by Dennard's rules cannot guarantee the constant power density it once did, due to the leaky devices in recent technologies. Consequently, transistors have become abundant, but the active power consumption is expected to generate heat that far exceeds the ability to cool the processor. Therefore, the power and thermal characteristics of a processor have become a critical resource. In response, processor designers limit the chip's total power consumption by avoiding a large fraction of the processor from operating simultaneously, a phenomenon known as dark silicon. Maximizing performance in the era of dark silicon requires novel techniques that optimally exploit the available power budget.

Optimizing processor performance under power constraints has been an important area of research. Dynamic Voltage and Frequency Scaling (DVFS) is a well-known mechanism for managing power, energy and thermals in single-core and multi-core processors. Although DVFS can be used to improve performance under power constraints, the supply-voltage range over which dynamic scaling can be performed has shrunk over the years, reducing the opportunity for DVFS. Consequently, both academia and industry have proposed Heterogeneous Chip Multi-Processors (HCMPs) to combat the limitations of DVFS. HCMPs consist of high performance 'big' cores and power-efficient 'little' cores. Scheduling applications on the big and little cores of HCMPs is an intricate task. However, it is fundamental to successfully leveraging the potential of HMCPs to maximize performance under power constraints.

In this work, we propose novel scheduling policies to maximize the performance of power-limited HCMPs. This convoluted problem requires optimizing the power budget partitioning among the multiple applications that

may concurrently utilize the processor. It also requires leveraging the power budget allocated to individual applications by scheduling different phases on big and little cores during the course of the application's execution. To complicate the problem further, scheduling must care for HCMPs that consist of multiple core types each featuring several voltage-frequency operating points. Unfortunately, a wide body of work targeting power management using DVFS is not directly applicable to HCMPs, raising the priority of tailoring solutions to this important problem.

Our view of a power constraint deviates from the conservative definition that assumes the power limit must never be exceeded. We allow the processor's power consumption to temporarily exceed the power limit, as long as it is preserved over a technology-dependent period of time we refer to as the *power period*. The flexibility added by this view of the power constraint has been exploited by prior works to significantly improve the responsiveness of interactive applications. In this thesis, we leverage this flexibility to optimize the sustained performance of applications running on a power-limited HCMP. This objective distinguishes our work from all prior techniques that optimize for other metrics or follow the conservative definition of a power constraint. Our work is also the first to improve performance of HCMPs considering a generic number of core types and a generic number of voltage-frequency operating points per core type.

We conquer this multi-dimensional scheduling problem by dividing it into three dimensions, and by tackling each of them individually.

For single-threaded applications, we show that the proposed techniques that target sustained performance, e.g., sprint-and-rest, do push the boundaries of the power limit to gain extra performance, but fail to leverage the available heterogeneity. For single-threaded applications, sprint-and-rest sprints by activating the big core and then rests by turning the processor off. We analyze sprint-and-rest's behavior to show that performance can be significantly improved if the scheduler uses the heterogeneous core types. We therefore propose sprint-and-walk, a technique that utilizes the heterogeneity in the processor. Instead of turning the processor off to rest, sprint-and-walk estimates the total energy the processor is allowed to consume in a power period and controls its usage over the whole period. By running on the little core, which consumes power at a rate lower than the allowed limit, the processor accumulates energy credit. The processor then burns the accumulated credit by running on the big core, which normally dissipates power above the limit. This cycle of sprinting on the big core and walking on the little core repeats throughout the application execution. Our results show that sprint-and-walk improves performance over sprint-and-rest by 9% on average across all SPEC CPU2006 applications, and up to 19%, for a moderate power budget of 1.25 W.

The improvement increases as the power budget gets tighter. For a budget of 0.5 W, the average improvement of sprint-and-walk relative to sprint-and-rest is 43% on average and up to 76%. More importantly, we estimate the highest performance attainable through single-threaded phase scheduling through exhaustive search. Our evaluation shows that the performance achieved by sprint-and-walk remains within a few percent of an optimal scheduler across a range of power limits, scheduling granularities, and HCMP configurations.

To maximize the performance of HCMPs running a multi-programmed workload, optimal power budget partitioning among applications is needed. We show that current techniques for power budget partitioning lead to sub-optimal performance because they do not consider the power and performance profiles of each application. These techniques include both equal budget partitioning among applications and techniques that let applications greedily compete for the shared power budget. Even widely used ranking metrics, such as the performance per Watt or big-to-little performance ratio, can also be misleading. All these metrics fail to consider that the allocated power budget restricts an application's utilization of a big core, leading to incorrect assessment of which application benefits the most from the allocated budget. Contrary to intuition, our results show that in many cases, memory-intensive applications dissipate less power than compute-intensive ones, allowing them to utilize the big core for a longer time duration, leading to higher overall performance with the same power budget. To fill in the missing gaps, we formulate the problem as a Linear Programming (LP) optimization problem. By solving the LP problem, we devise a novel power budget partitioning strategy and a metric based on the big-to-little delta performance by delta power (DPe/DPo). We use DPe/DPo to perform an online ranking of applications to schedule on the big core type. We propose DPDP, a fast and scalable scheduler that partitions the power budget among applications and schedules their execution on the HCMP such that the overall system performance is maximized. Our evaluation with DPDP on a heterogeneous processor consisting of four big.LITTLE pairs shows that DPDP improves chip performance by 16% on average and up to 40% over a strategy that greedily and globally utilizes the power budget. DPDP also outperforms commonly used scheduling metrics and heuristics. We analyze the impact of DPDP on per-application performance and we propose a technique to enforce a user-defined tolerable slowdown. Our results show DPDP's ability to maximize performance while maintaining the desired latency requirements.

Finally, we consider maximizing the performance of HCMPs featuring an arbitrary number of core types (e.g., big, medium and little), each with multiple voltage-frequency operating points. The complexity of the scheduler's task explodes as it has a significantly wider set of options. The scheduler must

assign each application to an operating point at any core type, and migrate it to any other operating point on any core type when necessary. To find the points that maximize performance for a given power budget, the scheduler typically walks the performance-power curve starting at the lowest operating point on the little core. It keeps walking the curve until it reaches the first operating point that exceeds the power limit. To leverage the available power budget, the scheduler continuously migrates the application between this point and the last point it encountered that does not exceed the power limit. The two selected points could be on two different cores. We show that naively walking the default set of operating points leads the application to inefficient operating points that drain power without significant performance benefit. We call these points Power Holes (PH) as they drain the power budget for sub-optimal performance. Contrary to intuition, we show that even using a power-performance curve of Pareto-optimal operating points still degrades performance significantly. We propose PH-Sifter, a fast and scalable technique that sifts the default set of operating points and eliminates power holes. We show significant performance improvement for PH-Sifter compared to Pareto-sifting for three use cases: (i) maximizing performance for a single application, (ii) maximizing system throughput for multi-programmed workloads, and (iii) maximizing performance on a system in which a fraction of the power budget is reserved for a high-priority application. Our results show performance improvements of 13, 27, and 28 percent on average that reach up to 52, 91 percent, and 2.3x, for the three use cases, respectively.

Samenvatting

De continue miniaturisatie van transistors heeft geleid tot een reeks nieuwe uitdagingen in processorontwerp. De combinatie van Moore's wet en Dennard schaling, die de laatste paar decennia grote sprongen mogelijk maakte op vlak van innovatie in de computerindustrie, is uitgedoofd. Ofschoon Moore's wet nog steeds geldt en het aantal transistors op een chip nog steeds toeneemt, schaalt, door toenemende lekstromen in recente technologieën, de vermogensdichtheid niet meer proportioneel mee zoals Dennard voorspelde. Bijgevolg neemt het totale vermogenverbruik toe en wordt er ook meer warmte gegenereerd dan er kan afgevoerd worden. Dit zorgt ervoor dat de vermogen- en warmtekarakteristieken van een processor een cruciaal probleem zijn geworden. Tegenwoordig beperken processorarchitecten het totale vermogenverbruik door te vermijden dat de volledige processor continu in gebruik is. Doordat een deel van de transistors niet gebruikt wordt, staat deze oplossing bekend als *dark silicon*. Het maximaliseren van de prestatie in dit dark silicontijdperk vereist nieuwe technieken die het beschikbare vermogenbudget optimaal benutten.

Het optimaliseren van de prestatie van een processor, rekening houdend met een vermogenbudget, is een belangrijk onderzoeksgebied. Het dynamisch schalen van de voedingsspanning en frequentie – *Dynamic Voltage and Frequency Scaling (DVFS)* – is een welgekende techniek om vermogen, energie en warmte onder controle te houden in processors met één of meerdere rekenkernen (*cores*). Hoewel DVFS een krachtige techniek is om prestatie te verbeteren bij een beperkt vermogensbudget, neemt het nut ervan af door de steeds lager wordende voedingsspanning. Zowel de academische wereld als de industrie hebben heterogene processors met meerdere rekenkernen – *Heterogeneous Chip-Multiprocessor (HCMP)* – voorgesteld als oplossing. Een HCMP is opgebouwd uit krachtige, 'grote' rekenkernen en vermogensefficiënte, maar zwakkere, 'kleine' rekenkernen. Het kiezen van de meest geschikte rekenkern om een applicatie op uit te voeren is een ingewikkeld probleem. Deze keuze is echter fundamenteel voor het maximaliseren van de prestatie van een HCMP met een beperkt vermogensbudget.

De focus van dit onderzoek is het uitwerken van nieuwe technieken zodat de prestatie van applicaties die uitvoeren op een HCMP met een beperkt vermogensbudget wordt gemaximaliseerd. Dit ingewikkeld probleem vereist het verdelen van het vermogensbudget over meerdere applicaties die de processor tegelijkertijd kunnen gebruiken. Ook is het nodig om het gebruik van het vermogensbudget van één applicatie te optimaliseren gedurende zijn uitvoering. Het is immers mogelijk dat verschillende fasen tijdens de uitvoering beter worden uitgevoerd op verschillende soorten rekenkern. Dit probleem wordt nog ingewikkelder doordat de verschillende soorten rekenkernen in een HCMP verschillende spanning-frequentiedomeinen kunnen hebben. Helaas is een groot deel van het reeds uitgevoerd onderzoek met betrekking tot DVFS niet toepasbaar op een HCMP. Hierdoor wordt het nog belangrijker om nieuwe, aangepaste oplossingen te ontwikkelen voor dit probleem.

De belangrijkste focus van dit werk is het behalen van een constante prestatie binnen een beperkt vermogensbudget. Dit doel onderscheidt het onderzoek van eerdere technieken aangezien deze optimaliseren ten opzichte van andere (prestatie-gerelateerde) metrieken. Enkele voorbeelden van eerder werk zijn technieken die de reactiesnelheid verbeteren of deadlines proberen halen. Deze technieken hebben duidelijk een ander doel en zijn dus ook niet onmiddellijk toepasbaar op ons probleem. In dit werk wordt een vermogenslimiet beschouwd over een bepaald tijdsinterval. Dit wil zeggen dat gedurende het tijdsinterval het wel mogelijk is om deze vermogenslimiet tijdelijk te overschrijden, maar dat over het volledige tijdsinterval het gemiddeld vermogenverbruik onder de limiet moet blijven. Deze randvoorwaarde onderscheidt dit werk van eerder onderzoek waar men de vermogenslimiet conservatiever behandelt doordat deze nooit overschreden mag worden. Tevens is dit werk het eerste onderzoek waarbij de prestatie wordt verbeterd voor een HCMP met een generiek aantal soorten rekenkernen en een generiek aantal voltage-frequentiedomeinen per rekenkern.

We benaderen de oplossing van dit meerdimensionaal probleem door het op te splitsen in drie verschillende dimensies en deze onafhankelijk van elkaar op te lossen.

Voor enkeldradige (*single-threaded*) applicaties tonen we aan dat technieken die een constante prestatie beogen, bijvoorbeeld *sprint-en-rust*, erin slagen de vermogenlimiet goed te gebruiken en extra prestatie te behalen, maar dat ze er niet in slagen om de heterogeniteit van een HCMP te benutten. De sprint-en-rust techniek zal werk uitvoeren op een grote rekenkern en daarna de processor uitschakelen. Voor enkeldradige applicaties schatten wij de maximaal haalbare prestatie door de ontwerpruimte exhaustief te exploreren. Hiermee tonen we aan dat, door de heterogeniteit van een HCMP te negeren, sprint-en-rust slechts een fractie behaalt van de maximaal haalbare prestatie.

Daarom stellen we een nieuwe techniek voor, namelijk *sprint-en-wandel*. In plaats van de processor uit te schakelen om te rusten, zal sprint-en-wandel de totale energie die de processor mag gebruiken gedurende een periode schatten en het verbruik controleren. Door een applicatie op een kleine rekenkern uit te voeren, die minder vermogen verbruikt dan de toegelaten limiet, wordt er energiekrediet opgebouwd. Dit krediet kan dan vervolgens gebruikt worden om tijdelijk op een grote rekenkern uit te voeren die meer vermogen verbruikt dan de toegelaten limiet. Deze cyclus waarbij er 'gesprint' wordt door op de grote rekenkern uit te voeren en 'gewandeld' wordt door op de kleine rekenkern uit te voeren, wordt herhaald gedurende de uitvoering van de applicatie. Voor een vermogenbudget van 1.25 W verbetert sprint-en-wandel de prestatie van sprint-en-rust met gemiddeld 9% over alle SPEC CPU2006 applicaties en zien we een maximale verbetering van 19%. Deze verbetering wordt groter indien het vermogenbudget kleiner wordt. Voor een vermogenbudget van slechts 0.5 W bedraagt de gemiddelde prestatiewinst van sprint-en-wandel 43% ten opzichte van sprint-en-rust met een maximale winst van 76%. Bovendien is de behaalde prestatie met onze techniek slechts een paar procent verwijderd van het optimum, de maximaal haalbare prestatie die we bepalen via exhaustieve exploratie.

Om de prestatie van werklasten met meerdere applicaties te maximaliseren op een HCMP is het nodig om een vermogensbudget correct te verdelen. We tonen aan dat de huidige algoritmes voor partitionering van een vermogensbudget slechts een suboptimale prestatie behalen omdat ze geen rekening houden met een vermogen-prestatie profiel per applicatie. Sommige algoritmes proberen het vermogensbudget eerlijk te verdelen over alle applicaties. Andere algoritmes volgen een gulzige strategie waarbij applicaties zelf een deel van het vermogensbudget kunnen alloceren waardoor rekenintensieve applicaties bevoordeeld worden ten opzichte van geheugenintensieve applicaties. Ook vaak gebruikte metrieken zoals prestatie-per-watt of de verhouding in prestatie op de grote versus kleine rekenkern kunnen leiden tot foutieve conclusies aangezien ze niet echt geschikt zijn voor het optimaliseren van een vermogensbudget over een tijdsperiode. Onze resultaten voor geheugenintensieve applicaties tonen dat, tegen de verwachting in, deze applicaties in veel gevallen minder vermogen verbruiken als ze uitvoeren op een grote rekenkern dan een rekenintensieve applicatie. Hierdoor kunnen ze gedurende een langere tijd op een grote rekenkern uitvoeren en behalen ze dus een hogere prestatie onder een bepaald vermogensbudget. In ons werk pakken we deze problemen aan door een nieuwe metriek voor te stellen, DPe/DPo, die de verhouding van het verschil in prestatie op het verschil in vermogen kwantificeert voor verschillende rekenkernen en configuraties. Deze metriek sorteert applicaties op basis van hun prestatiewinst indien ze

een bepaald deel van het vermogen krijgen en laat toe het vermogensbudget optimaal te verdelen. We formuleren de metriek en het bijhorende algoritme om het vermogen optimaal te verdelen als een lineair optimalisatieprobleem en lossen dit op. We evalueren dit DPe/DPo algoritme met betrekking tot het maximaliseren van de prestatie onder een bepaald vermogensbudget voor een HCMP met vier rekenkernen. Gemiddeld genomen wordt de prestatie verbeterd met 16% en is de maximale prestatiewinst 40% ten opzichte van eerder voorgestelde algoritmes die het vermogensbudget globaal verdelen.

Ten slotte maximaliseren we de prestatie voor een HCMP met een generiek aantal soorten rekenkernen, bijvoorbeeld grote, gemiddelde en kleine rekenkernen, elk met meerdere voltage-frequentiedomeinen. De complexiteit van het probleem wordt hierdoor veel groter aangezien elke optie met elke andere optie gecombineerd kan worden. De 'scheduler' moet, voor elke applicatie en elke soort rekenkern, het meest geschikte spanning-frequentiedomein kiezen en ook in staat zijn om een applicatie op een andere rekenkern uit te voeren met een verschillende spanning en/of frequentie indien nodig. Voor het zoeken van de optimale configuraties zal de 'scheduler' de prestatie-vermogencurve overlopen startend in het laagste punt, de uitvoering op de kleine rekenkern, tot het op een punt komt waarbij het vermogensbudget overschreden wordt. Om zeker te zijn dat het volledige vermogensbudget benut wordt, zal de applicatie steeds gemigreerd worden tussen de twee configuraties juist onder en boven het vermogensbudget. Deze twee configuraties kunnen twee verschillende rekenkernen omvatten, maar ook dezelfde rekenkern met een lagere of hogere frequentie.

Het nadeel van deze naïeve techniek is dat het mogelijk is configuraties te selecteren die meer vermogen verbruiken zonder dat er prestatiewinst is. We noemen deze configuraties 'Power Holes' aangezien ze vermogen verbruiken zonder prestatiewinst op te leveren. Zelfs indien de prestatie-vermogencurve is opgebouwd uit enkel Pareto-optimale configuraties, is het nog steeds mogelijk suboptimale prestatie te bekomen. Dus stellen we 'PH-Sifter' voor, een snel en schaalbaar algoritme dat de verschillende configuraties 'zeeft' om 'Power Holes' te elimineren. Onze resultaten tonen aan dat er een beduidende prestatiewinst te behalen valt in minstens drie verschillende gevallen: (i) het maximaliseren van de prestatie voor één enkele applicatie, (ii) het maximaliseren van de prestatie voor werklasten met meerdere applicaties, en (iii) het maximaliseren van de prestatie indien een deel van het vermogensbudget gereserveerd wordt voor een applicatie met een hoge prioriteit. Voor deze drie gevallen zien we gemiddelde prestatiewinsten van respectievelijk 13, 27 en 28 procent en maximale prestatiewinsten van 52, 91 en 230 procent.

Contents

List of Figures

List of Tables

List of Abbreviations

ANTT	Average Normalized Turnaround Time
BIPS	Billion Instructions Per Second
CMPs	Chip-Multiprocessors
DVFS	Dynamic Voltage and Frequency Scaling
EAS	Energy-Aware Scheduler
GPU	Graphics Processing Unit
HCMPs	Heterogeneous Chip Multi-Processors
IPS	Instructions Per Second
ISA	Instruction Set Architecture
LLC	Last-Level Cache
LP	Linear Programming
MLP	Memory-Level Parallelism
PCM	Phase-Change Material
PH	Power Holes
PIE	Performance Impact Estimation
STP	System Throughput
TCO	Total Cost of Operation
TDP	Thermal Design Point

Chapter 1

Introduction

1.1 Motivation

For the past decades, the performance-driven computing industry has been empowered by two major pillars: Moore's law and Dennard scaling. Moore predicted that the number of transistors on a chip can be doubled almost every two years [2]. Due to its persistence, this observation is known as Moore's law. Dennard [3] proposed a method for scaling the device feature sizes, the supply voltage, and the operating frequency, such that the extra transistors integrated in the chip could operate faster and at lower voltage levels, while maintaining a steady chip power density. By virtue of Dennard scaling, chip manufacturers were able to provide higher performance with every generation while avoiding power-and thermal-induced problems. Unfortunately, recent technology advancement exposed new phenomena that hindered the traditional scaling as predicted by these two laws. Driving the computing industry beyond traditional scaling laws leaves computer architects with new problems and challenges to address.

The original Moore's law has slowed down due to the difficulty of shrinking transistors at the initial rate predicted by Moore. However, Moore's law is expected to hold in the foreseeable future, despite the slightly extended duration between technology nodes. On the other hand, Dennard scaling has already reached an end. Dennard scaling rules accounted only for the relation between the transistor dimensions, voltage, current and frequency. These elements affect the dynamic power of the processor. As transistor dimensions kept shrinking, the transistor gates became as thin as a few layers of atoms. At these small dimensions, leakage currents rose significantly making leakage power a major source of power dissipation in the processor. To scale down the supply voltage, an equal scaling of the threshold voltage

is required to allow scaling the operating frequency. Failing to scale down the supply voltage breaks Dennard scaling and results in an increase in power density. Scaling down the threshold voltage is physically limited and increases static power exponentially. Failing to manage the operating frequency as per Dennard scaling results in a performance degradation. In all cases, chip power densities are projected to keep increasing as transistor counts keep increasing.

The impact of the end of Dennard scaling became apparent with operating frequencies reaching a plateau, and the adoption of Chip-Multiprocessors (CMPs) to satisfy the market's appetite for performance. However, as the number of transistors continues to rise, power and thermal considerations pose serious threats to the performance improvements that can be expected from future technology generations. In fact, even today's processors cannot operate indefinitely at the maximum operating frequency, or cannot even turn on a significant fraction of their transistor real estate at once, a phenomenon known as *Dark Silicon* [4]. To combat power and energy challenges, power- and thermal-aware designs try to reduce the processor's power consumption, and optimally utilize the transistor real estate within the safe limits. Most of the prior power management work rely on dynamic voltage and frequency scaling (DVFS). However, DVFS suffers from the same threshold scaling problem. The inability to further scale down the threshold voltage has continuously shrunk the voltage ranges available for DVFS. Therefore, the power-performance operating points available through DVFS provide a limited solution to the power-related challenges.

Heterogeneous chip-multiprocessors (HCMPs) have been proposed as a solution that extends the power-performance operating options beyond DVFS [5]. HCMPs can also be used to cope with stringent power limits that force turning off parts of the chip. These processors integrate a mix of cores that vastly differ in their microarchitecture complexity, power and performance characteristics. The mix of cores ranges from high-performance but power-hungry cores to power-efficient but low-performance cores. In addition to extending the power-performance range over DVFS, the energy-efficient cores of HCMPs occupy less area and thus consume significantly less static power than high-performance cores with DVFS. Therefore, HCMPs provide more options to counter tight power limits and dark silicon. The usefulness of heterogeneous processors relies on intelligently scheduling applications to the appropriate core type, and migrating to another core type when necessary. **Our goal in this thesis is to propose novel scheduling techniques that optimize the performance of long-running applications on HCMPs in cases of stringent power budgets**. Going with the expected trends, we consider power budgets that limit the number of cores

that can be activated at any given point in time.

1.2 Key Challenges

Scheduling applications to core types, and partitioning the power budget among the applications (and their execution phases) is a complex optimization task with multiple dimensions to explore. The complexity increases with the number of applications, the number of different types of cores, and the voltage-frequency operating points per core. We deviate from prior works that conservatively define a power limit as a rate that must never be exceeded. Instead, we adopt a definition of a power limit that allows exceeding it momentarily as long as it is preserved over a time duration that is based on the processor's power and thermal characteristics. The same view on a power limit has been adopted by prior work and used to improve the application's responsiveness [6, 7]. This view of power opens another dimension of optimization possibilities for sustained performance. A scheduler may need to budget its power consumption properly over a whole time duration. Scheduling and power budget partitioning has to be fast, and has to scale with the number applications and operating points.

Despite the large body of DVFS and HCMP related work, they all explore only a subset of the problem dimensions. Techniques that were proposed for homogeneous machines with DVFS are not directly applicable to heterogeneous machines because they ignore heterogeneity among core types. Several techniques that optimize for performance under a power budget are either limited to DVFS, target single-threaded applications, follow non-scalable approaches, or assume the power limit cannot be exceeded momentarily during the execution. Due to the complexity of the problem and the vast space exploration required to find an optimal solution, the key challenge is to devise a fast and scalable solution to maximize performance of multiple applications running on a power-limited HCMP.

1.3 Key Contributions

To address the aforementioned challenge, we decompose the problem into its basic dimensions and investigate each of them separately. The combined contributions form our solution to performance maximization on power-limited HCMPs. First, we investigate solutions for optimal power management of single-threaded applications on a power-limited HCMP. Then, we add another dimension and solve the more general problem of maximizing

performance on an HCMP running a multi-programmed workload. Finally, we provide an optimal solution for maximizing performance on an HCMP comprised of any number of heterogeneous core types (e.g., big, medium, and little), where each core type can have multiple DVFS operating points. The main contributions of this work focus on maximizing performance of long-running applications on HCMPs under power constraints. HCMPs can be used to run other types of applications, e.g., applications with bursty behavior or strict performance requirements. We provide, as we see fit, further discussion on how to extend our proposed techniques to meet the performance demands of these types of applications.

1.3.1 Optimizing Single-Threaded Performance

Our goal in this contribution is to maximize sustained performance of a single-threaded application on a power-constrained HCMP. Most previously proposed power management techniques for single-threaded applications focus on improving either energy efficiency or application responsiveness. Moreover, DVFS is the backbone for most of these techniques. We show that the relevant techniques targeting sustained performance do not leverage the heterogeneity in the processor. Using exhaustive search, we measure the maximum performance that can be achieved by optimally using the HCMP cores under a power limit. We demonstrate that prior work degrades performance significantly under the same power budget.

We propose a technique that extends over prior proposals to successfully leverage HCMPs. We study the performance of our proposed technique in comparison to both prior work and the maximum performance under the same power limit. We show that our approach not only significantly improves performance over prior work for a wide range of power budgets, scheduling granularities and HCMP configurations, but also remains within a few percents of optimal performance.

1.3.2 Maximizing Throughput for Multi-Programmed Workloads

We look into techniques to maximize performance for multi-programmed workloads running on power-limited HCMPs. Apart from sharing cores and caches, the main resource among applications is the power budget. The problem here consists of two components. The first component is how to partition the power budget among the co-executing applications. The second component is which core type to assign to each application, and when to migrate between the big and little cores.

In this work, we make several contributions. First, we show that current scheduling and power management techniques are not suitable to maximize performance under a power limit. The metrics used to rank applications when partitioning the power budget are sub-optimal. We discuss the limitations of each technique and ranking metric. We show that these techniques do not consider the time period over which the power budget is calculated. Therefore, they fail to accurately estimate the performance of each application based on its allocated power budget. Second, we formulate the problem as a linear programming optimization. However, solving the linear program at scheduling time is too slow and not scalable to high core counts. We solve the linear program mathematically, and use the solution to provide an optimal scheduling strategy and a ranking metric. Our proposed strategy and ranking metric are fast and scalable, thus can be used at runtime. Finally, we propose a scheduler implementation that maximizes performance when given any workload mix, HCMP configuration, and power limit.

This work is published in:
A. Adileh, S. Eyerman, A. Jaleel, L. Eeckhout. Maximizing Heterogeneous Processor Performance under Power Constraints. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(3), p.29, 2016

1.3.3 Maximizing Performance on Generic HCMPs with DVFS

Core heterogeneity has been adopted in products relatively recently. Scheduling techniques tailored for HCMPs are under development, aiming to exploit all heterogeneous core types and their respective voltage-frequency operating points to improve energy efficiency. The techniques we propose so far in this thesis apply to HCMPs with multiple big and little cores but each core is limited to a single voltage-frequency setting. The optimization space significantly increases with multiple voltage-frequency operating points. The optimized scheduler must select the power budget to assign to each application, which operating point to use, when to migrate, and which other core and operating point to migrate to.

We contribute a comprehensive scheduling technique to optimally utilize the power budget on a generic HCMP where each core features a generic number of operating points. We show that current techniques that walk the power-performance curves to find points of maximum performance for a given power budget work for homogeneous architectures. However, walking naively from the little core's power-performance curves to the big core curves can result in landing the application on operating points that waste power and degrade performance. Surprisingly, we show that walking Pareto-optimal power-performance curves to reach the points of maximum

performance for a given power budget suffers from the same problem. We call the sub-optimal operating points "power holes", because reaching these points drains the limited power budget and prevents the scheduler from reaching an optimal solution.

Our approach walks a power-performance curve that contains only a subset of the operating points that optimally trade power for performance. To find the optimal subset of operating points, we propose a technique to prune the power holes on a heterogeneous architecture. Our technique profiles each application on a single voltage-frequency operating point on each core type. Using performance and power models, we generate the power-performance curves for each application on all core types. Our technique takes these curves and sifts the power holes per application to provide a set of optimal points. We show significant performance improvements compared to naive and Pareto-optimal curve walking.

This work is published in:
A. Adileh, S. Eyerman, A. Jaleel, L. Eeckhout. Mind The Power Holes: Sifting Operating Points in Power-Limited Heterogeneous Multicores. *IEEE Computer Architecture Letters (CAL)*, 16(1), pages 56-59, 2017

1.4 Other Research Activities

In addition to investigating novel scheduling techniques for power-limited heterogeneous processors, I contributed to other research activities targeting a diverse set of problems.

1.4.1 Cloud Workload Benchmarking and Characterization

Cloud computing emerged as a promising way to fully utilize the processors available in the datacenter. Several forms of computing services, spanning various levels of the stack, are being offered in public and private datacenters. Among the rush of applications populating datacenters, an important set of applications has emerged. The distinctive characteristics of these applications set them apart from important classes of workloads including desktop, scientific, parallel, and even server applications. These applications use a scale-out model of operation, in which processing a request from a user does not necessarily require stronger, more aggressive machines due to the need to access massive datasets. Instead, the request is split into multiple tasks each running on a different machine that has access to an independent shard of the dataset.

In the quest to propose processor designs tailored to enhance the performance and efficiency of datacenters running these scale-out applications, we noticed the lack of representative benchmarks and tools to characterize their behavior. I contributed to the creation of a new benchmark suite, called CloudSuite [8], to represent the modern scale-out workloads that run in datacenters. I significantly contributed to the release of the first version of CloudSuite with detailed instructions on how to install and fine-tune its applications to study their behavior in a research environment. Moreover, I organized and participated in giving a tutorial at ISCA 2012 on how to run CloudSuite on real machines as well as in a simulation environment. I also contributed to the creation and release of Simics images that include client, server, and back-end machines to simulate scale-out applications scaled up to 64 cores, along with the necessary datasets.

I also contributed to the analysis and characterization of scale-out applications on real machines. This work shows how scale-out applications utilize the various parts of the processor, and quantitatively demonstrates how contemporary server-class machines, designed to suite a generic set of workloads, over-provision many processor features making them inefficient at running scale-out applications. For example, the mismatch between server machines and scale-out application requirements include the provisioned pipeline width, the number of cache hierarchy levels, and Last-Level Cache (LLC) capacity, among others. The work also provides recommendations for designing an efficient processor to run scale-out applications.

This work won the best paper award at ASPLOS 2012:
M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware, *in Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pages 37-48, 2012

This work was also selected as an IEEE Micro Top Picks: M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, A Case for Specialized Processors for Scale-Out Workloads. *IEEE Micro. Top Picks*, 34(3), pages 31-42, May 2014

An extended version of this work also appeared in TOCS 2012: M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, Quantifying the Mismatch Between Emerging Scale-Out Applications and Modern Processors, *ACM Transactions on Computer Systems (TOCS)*, 30(4),

p.15, November 2012

1.4.2 Scale-Out Processors

Building on the characteristics of scale-out applications, this work optimizes the processor design targeting these workloads. In particular, due to the large instruction working sets of scale-out applications, the main benefit of shared LLCs comes from capturing the instruction working set as they spill from the private caches. However, due to their massive datasets, scale-out applications do not benefit significantly from the large on-chip caches. As the capacity of on-chip caches grows larger, their access latency increases. This latency is critical to performance as it impacts both data and instruction accesses. Therefore, contrary to expectation, as the LLC capacity increases, the negative impact of its access latency on performance outgrows its performance benefits, causing an overall performance degradation. Server processors for scale-out workloads benefit more from sizing LLCs modestly and replacing the area originally occupied by the large LLC with more cores. However, increasing the number of cores does not improve performance linearly either. The longer route to access even a modest non-uniform LLC incurs a higher access latency as the number of cores increases. We provide a processor design methodology that determines the number of cores and LLC size, such that performance density (i.e., performance per unit of chip area) is maximized. Each configuration of cores and LLC given by this methodology is called a 'pod'. Processors usually have enough transistors to integrate higher core counts and LLC capacity. A pod is replicated several times to populate the available area of the processor, and each pod acts as an independent server. Pods do not communicate through an on-chip network, further saving the area occupied by large on-chip networks for other purposes. Our results show significant improvement for scale-out applications over both conventional and tiled server architectures.

This work is published in:
P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, B. Falsafi, Scale-Out Processors, *in Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Pages 500-511, June 2012

1.4.3 Architectural Support for Probabilistic Branches

Branch predictors are subject to continuous fine-tuning due to the heavy branch misprediction penalty, especially in modern deep and aggressive

pipelines. We observe that numerous applications from important and emerging applications, including scientific, engineering, biology, financial and learning applications, use algorithms that draw random values from a distribution and steer control flow based on those values. Such probabilistic branches are challenging to predict because of their inherent probabilistic nature. As a result, probabilistic codes significantly suffer from branch mispredictions. We propose *Probabilistic Branch Support (PBS)*, a hardware/software cooperative technique that leverages the observation that the outcome of probabilistic branches needs to be correct only in a statistical sense. PBS stores the outcome and the probabilistic values that lead to the outcome of the current execution to direct the next execution of the probabilistic branch, thereby completely removing the penalty for mispredicted probabilistic branches. PBS relies on marking probabilistic branches in software for hardware to exploit. Our evaluation shows that PBS improves MPKI by 45% on average (and up to 99%) and IPC by 6.7% (up to 17%) over the TAGE-SC-L predictor. PBS requires 161 bytes of hardware overhead and introduces statistically negligible algorithmic inaccuracy.

This work is currently under review.

1.5 Structure and Overview

In this work we investigate scheduling techniques for maximizing the performance of HCMPs operating under a limited power budget.

In Chapter 2, we provide a motivational background to the significance of power limitations, the rise of dark silicon that allows only a fraction of the processor to be active at any point in time, and the promising concept of heterogeneous processors and the challenges associated with application scheduling on power-limited heterogeneous processors.

In Chapter 3, we investigate techniques for maximizing the performance of an HCMP running a single-threaded application via optimized scheduling of the application execution phases.

We then seek to optimize power budget partitioning among multiple competing applications in Chapter 4. We propose a fast and scalable technique to partition the budget between applications, schedule them on, and migrate them between the big and little cores, such that throughput is maximized.

In Chapter 5, we target a solution of the generalized problem where the HCMP consists of a generic number of big and little cores and each core features several voltage and frequency operating points. This generalized solution has an impact on single-threaded and multi-programmed applications,

in addition to cases where part of the budget is reserved for a time-critical task.

We provide a summary of our work in Chapter 6, in addition to potential directions for future exploration.

Chapter 2

Background

2.1 A Historic Perspective

The power consumption facet of computing systems has been a source of challenges for computer architects since the early computing machines. The problems and challenges associated with the power consumption of computing devices kept evolving with the evolution of the underlying technology.

Early computing systems, such as the ENIAC computer from the 1940's relied on power-hungry vacuum tubes and dissipated around 174 KW [9]. This power consumption rate was alarming especially with the low computing capabilities offered by the machine. The machine was spread over an area of several rooms, causing its power density (i.e., power/area) to be low. This implied that the major power-related problem for that machine was the high cost of computation. The low power density implies a relatively easy cooling task.

Technology evolved to solve the power cost-efficiency problem by adopting bipolar transistors. These transistors were relatively more power-efficient as early chips could perform the same computations as the ENIAC computers with only a few Watt of power. This technology allowed for powerful computational capabilities in a small-sized chip. Despite consuming significantly less power, the small chip size caused the power density to rise. To drive the demand for higher performance, more bipolar transistors were integrated on a single chip, raising its power consumption and density. The problem evolved from being mostly cost related to include cooling-related challenges. The continuous escalation of the power dissipation of bipolar devices led to the adoption of CMOS technology in the early 1990s.

The relatively low power of CMOS devices gave architects a temporary relief from power and heat issues. The transistors were efficient as they did not dissipate power unless they partook in a switching activity. However, their very large scale of integration and the continuous reduction of their feature sizes rekindled old problems. Power density started rising again, and the switched-off (static) power consumption started to get more significant due to non-negligible leakage currents at low feature sizes. This resulted in the rebirth of power and cooling challenges for this new technology.

Power consumption has become a primary design concern for processors, leading to plenty of research activity on energy- and thermal-aware processor designs for all processor domains. For mobile devices, power-aware designs aim to minimize energy consumption to extend battery life, and limit heat dissipation to acceptable levels for human interaction and manageable using only passive cooling. For desktop processors, energy consumption is one issue of concern, but cooling these processors as they strive to boost performance is a major challenge. For example, desktop processors reached a plateau in maximum operating frequency due to thermal concerns. Instead, performance of desktop processors relies on improving throughput using multiple cores. For server processors, power consumption is an even larger design concern given the emphasis on high performance and the impact power consumption has on total cost of operation (TCO). The extremely high power consumption rates translate into direct operating cost, which gets doubled when considering the power delivery and cooling requirement when deployed in a datacenter.

2.2 The End of Dennard Scaling

For the last few decades, improvements brought by every technology generation relied on two main pillars: Moore's law and Dennard scaling. Moore observed in the 1960's that the number of transistor integrated on a chip doubles every two years [2]. This observation persevered for the past few decades against numerous speculations about its end, but slowed down slightly in the latest technology generations. Dennard, on the other hand, introduced a method for scaling transistors [3] which allowed for the increase in transistor density and speed, while maintaining the power density unchanged across technology generations. However, Dennard scaling has ended in the last decade, causing paramount complications to processor architecture design.

Dennard scaling relied on scaling the power supply voltage and the operating frequency with the same factor as the device dimensions to guarantee steady power density. To scale down the supply voltage and increase switching

speed, the transistor threshold voltage has to be scaled down as well. The continuous shrinking of transistor dimensions uncovered the severity of leakage power. Transistor gates became very thin, featuring only few layers of atoms, and forming leakage paths that drain power even when the device is held to an off-state. The transistor ceased to operate as the nearly perfect switch it once was. This phenomenon raised the contribution of static power to the total processor power consumption. Per Dennard scaling, as supply voltage is scaled down, threshold voltage is scaled down as well, leading to exponentially elevated static power. We have reached a lose-lose situation: scaling the power supply to maintain power scalability does not work as leakage power increases with reduced threshold scaling, while not scaling the supply voltage increases power density as more transistors are integrated. Dennard's scaling rules are not applicable to this era of leaky transistors. The end of Dennard scaling poses serious questions regarding the best methods to drive performance with Moore's law.

The end of Dennard scaling has evident consequences on processor design. One clear example is halting the rise of operating frequency around 2004 [5]. Raising the frequency of operation and using deep complex pipelines to improve performance cannot be sustained without violating power and thermal envelopes. Instead, processor designers use the extra transistors to integrate multiple cores, and extract performance through parallelism. As the number of cores increases with Moore's law, parallelizing applications to drive performance becomes more challenging, and gets limited by Amdahl's law.

Increasing transistor counts while slowly scaling down the voltages is a serious problem. This trend has led to chips that feature more transistors than can be simultaneously powered on within safe thermal envelopes. Therefore, a significant part of current chips must be turned off. For example, early estimates from ARM co-founder and CTO Mike Muller, predicted that about only 9% of the chip real estate can be operated at once within the safe operating limits by 2019 [10]. A following study [4] predicted that chips need to power off around 50% of its transistors at 8 nm technology. More recent estimates [11] consider improvements in transistor technology by assuming studying 7 nm FinFets under a specific thermal design power to conclude that around 64% of the chip cannot be operated simultaneously under such technology assumptions. This phenomenon has been famously known as *dark silicon* [4], or the *utilization wall* [12].

To combat the rising power challenges, power-, energy-, and thermal-aware techniques have been proposed. Several tools are available to computer architects, including dynamic voltage and frequency scaling (DVFS), heterogeneous processing, and specialized units. In the following sections, we survey

the main concepts behind these techniques as they lay out the foundation for this thesis.

2.3 Power Management

In this section we focus on two important tools in the computer architect's toolbox to trade off power and performance, according to an application's requirements. These tools are dynamic voltage and frequency scaling and heterogeneous multicore processors. We describe the foundation of these techniques before going over notable proposals to apply them. We limit the survey to proposals that apply these techniques to optimize for various metrics without the assumption of a power limit. We devote Section 2.4 to discuss the implications of a power limit and optimizations under a power limit.

2.3.1 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is one of the key techniques for power and energy management of processors. The key idea behind DVFS stems from the basic equation describing dynamic power consumption: $P = ACV^2f$, where A is a term to factor in the transistors' switching activity, C is the load capacitance, V is the supply voltage, and f is the operating frequency. By reducing the supply voltage by a factor *k*, a quadratic reduction of k^2 in the total dynamic power is obtained. The reduction in supply voltage slows the switching of the transistor, requiring a proportional reduction in operating frequency. In total, DVFS can achieve a cubic reduction in dynamic power by scaling both voltage and frequency. However, scaling the frequency down leads to a degradation in performance. Therefore, power and energy management proposals apply DVFS only to applications or application phases with a relatively small performance degradation due to frequency scaling.

DVFS is an effective technique for the reduction and control of dynamic power. However, as feature sizes shrink with technology advancement, leakage power increases making static power a major source of power dissipation. Not only does this raise the demand for techniques tailored for static power, leakage concerns also impose a limit on DVFS effectiveness. The operating frequency is proportional to the difference between the supply voltage and the transistor threshold voltage. To maintain the range of operating frequencies, the threshold voltage must be scaled equally to the supply voltage. However, leakage power grows exponentially with the reduction

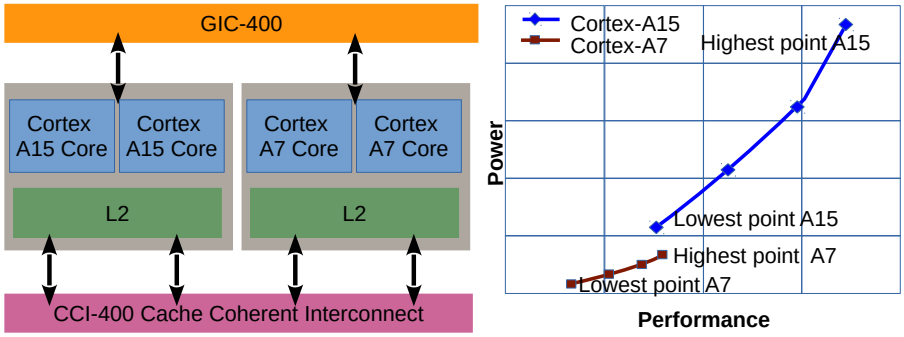


Figure 2.1: ARM's first big.LITTLE processor [1]. The Cortex-A15 is used as the big core and the Cortex-A7 is used as a little core. Heterogeneous architectures expand the range of operating points beyond DVFS.

in threshold voltage. Therefore, a trend in the range of DVFS voltages can be seen with each smaller technology node. The maximum supply voltage levels kept continuously shrinking but the minimum levels were only slightly reduced. Therefore, the range of DVFS operating values kept shrinking with technology advancement.

2.3.2 Heterogeneous Multicore Processors

Heterogeneous CMPs (HCMPs) emerged to counter the shrinking ranges of voltage-frequency operating points and the power limits of the processor. Heterogeneous computing comes in several forms. In this thesis we focus on heterogeneous architectures that mix multiple types of general-purpose cores that share the same instruction-set architecture (ISA) but have different microarchitectural implementations. We do not discuss other forms of heterogeneity in this thesis, such as mixing general-purpose cores with graphics processing units (GPUs) to handle the data-parallel part of applications, or mixing general-purpose cores with accelerators that excel at a very specific computation. In general, all these forms of heterogeneity share the same goal of supplementing general-purpose processors with more power-performance options for faster and more efficient execution.

HCMPs expand the range of power-performance operating points over CMPs. Figure 2.1 is taken from [1]. The left part of Figure 2.1 shows a heterogeneous processor featuring two types of cores that implement the same ISA. The first is a superscalar out-of-order core type that provides high performance, called the "big core". The second type, called the "little core", is an in-order core designed for low power consumption. The first

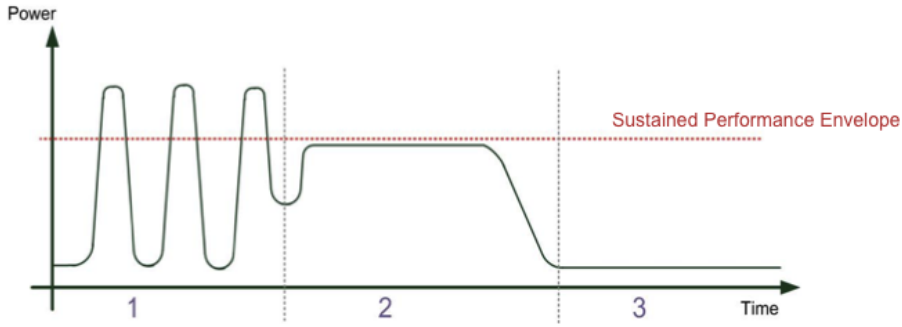


Figure 2.2: Three modes of using ARM's big.LITTLE processors.

ARM big.LITTLE architecture uses an ARM Cortex-A15 as the big core and a Cortex-A7 as the little core [1]. Numerous commercial products employ various flavors of single-ISA heterogeneity [13–16]. The right part of Figure 2.1 re-plots the example power-performance curves of the first ARM big.LITTLE processor taken from ARM's white paper [1]. The big core provides high-performance operating points but provides few options for low power. Little cores stretch the curve to provide lower power operating points due to their simpler pipeline and smaller chip area.

ARM proposed three use cases for its big.LITTLE products [17], as shown by Figure 2.2. The first use case is the case of bursty applications. The workload passes through bursts of high-performance requirements that benefit from the big core; periods that require only modest performance can use the little core. In the second case, the power budget allocated to the cores is limited, and the big.Little architecture is used to maximize the sustained performance under the power limit. This process may involve scheduling applications on the appropriate core types and migrating them when appropriate. In the third case, the application requires only modest performance. HCMPs offer the option to run them on the little cores to improve energy-efficiency, while turning off the big cores.

Reaping the benefits of HCMPs requires scheduling techniques to achieve the desired optimization goals. For example, to trade off performance and energy-efficiency for a single-threaded application, the scheduler has to identify phases of execution that can be run on the little core without degrading performance significantly, similar to DVFS. In a multicore processor running a multi-programmed workload, the scheduler has to identify the best application-to-core mapping to achieve the desired goal of optimizing performance or preserving energy, or any criterion for optimization of interest.

2.3.3 Scheduling Without Power Constraints

I. Optimizations Using DVFS

The main task of power- and energy-aware designs is to find the right strategy to apply DVFS. The techniques we briefly describe in this section all differ in their strategy. For example, they may differ in the domain of DVFS usage, e.g., at the system level or at the application phase level. They may also differ in the granularity at which DVFS is applied, e.g., core vs. chip-level. Despite all the differences, the general principle they all share and exploit is the same. DVFS techniques rely on finding periods of performance slack in the application run. During these periods, lowering the processor frequency can be more tolerable than during other parts of the execution where performance is critical.

At the system level, techniques like [18] target systems that mix real-time and non-real-time applications. The goal is to find slack by establishing a deadline for each application based on its level of interaction with the user. Based on the established deadlines, this technique estimates the voltage-frequency setting that maximizes energy-efficiency while meeting the deadline.

This thesis focuses on optimizing performance on heterogeneous processors based on the application characteristics. Applications (and their phases) are categorized into memory-intensive and compute-intensive categories. The general wisdom suggests that in memory-intensive applications (or phases) the processor is more likely to stall waiting for the lengthy memory operations. Therefore, scaling down the processor frequency during these phases leads to minimal performance loss, assuming the memory system operates on a separate frequency domain. During a memory access, computations that are independent of the access can run concurrently with it, while other computations that are dependent on the memory access must wait for its completion. Scaling down the frequency for the independent instructions only stretches their execution time, which overlaps with the memory access. Thus extending their execution time has negligible impact on overall performance. On the other hand, scaling down frequency for dependent instructions that do not run concurrently with the access, exposes their time and directly degrades performance.

Several approaches have been proposed to isolate memory-intensive parts of the code for single-threaded applications. Prior techniques include static compiler techniques, such as [19, 20]. These techniques fail to account for the underlying processor microarchitecture. Dynamic techniques that rely on performance monitors and runtime adjustment adapt more accurately at runtime. One notable line of work by Isci and Martonosi [21] splits the appli-

cation into power phases. They use a vector of performance counters to give each phase a fingerprint. Each counter in the vector gathers measurements related to a different component of the processor. The values in a vector change with the application phases, and an adaptation mechanism reacts to the change in power phases. The work is also extended to predict the next power phase based on currently collected performance counters [22].

Significant work has targeted CMPs exploiting the same performance slack principle. For example, Donald and Martonosi [23] establish the advantages of having a per-core DVFS autonomy compared to chip-wide policies. A chip-wide policy brings down all the core frequencies at the same time and does not suite the characteristics of individual applications in a multi-programmed workload. Plenty of research work has also been invested in understanding the impact of DVFS on performance of parallel multi-threaded application running on CMPs, and proposing methods for performance-energy adaptation for these applications [24–28].

II. Optimizations Using HCMPs

In general, techniques that optimize on HCMPs expose periods of slack, similar to the DVFS techniques. In addition to these techniques, we also review proposals that solve problems relevant only to HCMPs, along with proposals to optimize several other objectives.

HCMPs have the potential to improve several metrics over homogeneous processors. Kumar et al. [29] demonstrate the energy efficiency benefits of HCMPs over CMPs. They also show that HCMPs can even improve performance over CMPs under a constrained area by substituting a big core with several little cores using the same area [30]. Note that integrating more little cores benefits throughput but degrades per-application performance. Therefore, it may not be suitable for all mixes of multi-programmed workloads or parallel applications, as also indicated by Hill and Marty [31].

Several papers have proposed scheduling techniques that select the best application-to-core mapping for performance. Most notably, Van Craeynest et al. [32] provide a technique for dynamic scheduling to optimize HCMP performance when running multi-programmed workloads. Unlike static and offline analysis [33, 34], dynamic schedulers have the advantage of adapting the schedule to the change in the application behavior as it happens. Dynamic schedulers need a method to estimate the performance on one core type given its run on the other core type. Van Craeynest et al. [32] challenge prior wisdom by showing that memory intensity alone is a misleading metric and cannot accurately predict the application's potential

performance benefit due to using a big core. Therefore, schedulers should not use it to rank applications according to their anticipated benefit when run on a big core. They show that memory-intensive applications differ in the number of memory requests that can be sent to memory concurrently, known as memory-level parallelism (MLP). Memory-intensive applications with high MLP show significant performance gain as they migrate to a big core. They propose a method, called Performance Impact Estimation (PIE), that measures ILP and MLP on one core type, and plugs these statistics into big-to-little and little-to-big models to estimate performance on the other core type. Their experimental results report significant performance improvements compared to prior techniques.

Composite cores [35] is another notable scheduling work for HCMPs. Composite cores target minimizing energy given a tolerable performance slack. They show that a significant amount of energy can be saved by adjusting the schedule at extremely fine-grain intervals of 1 K instructions. The core migration overhead of current HCMPs is too high, forcing schedulers to operate at much coarser granularities. Composite cores build a non-traditional HCMP to allow low-overhead migration and fine-granularity scheduling. They use a shared front-end of the processor as it does a similar job in both core types. However, they deploy a heterogeneous backend consisting of one in-order (little) backend and an out-of-order (big) backend. This way, the migration between backends incurs minimal overhead and the proposed scheduling technique achieves high energy reduction. They follow the steps of PIE by relying on big-to-little and little-to-big performance models. Using a PID controller that takes the actual overall performance, the performance target, the performance of the last scheduling interval on one core type, and the model-predicted performance on the other core type, composite cores decide whether to run the following interval on the big or little core, such that the target performance degradation is not exceeded.

Other papers focus on meeting the quality-of-service requirements while using HCMPs for efficiency. In particular, Zhu et al. [36] classify applications based on their interaction with the end user using application statistics such as system calls and network usage. The applications that are classified as interactive applications get higher priority to run on the big core. Octopus-man [37] is another technique that tries to guarantee quality of service when needed, while running efficiently on the little core when there is performance slack. In particular, Octopus-man exploits the observation that online services have a peak of concurrent requests during the day and lower demand during the night. Therefore, it proposes using the big cores at peak request time, and using the little cores at times of low request levels. Several other works target performance and power goals. Van Craeynest et al. [38] focus

on scheduling applications on HCMPs such that the fairness is guaranteed among co-executing applications. The paper describes multiple formulas for fairness and proposes a scheduler that optimizes for these metrics on an HCMP.

2.4 Scheduling under Power Limits

So far, we briefly surveyed scheduling techniques that optimize for various metrics. In this section, we specifically focus on scheduling techniques under power limits. We provide the definition of a power limit that we assume throughout the thesis. Then we briefly go over the most recent schedulers that operate under power limits.

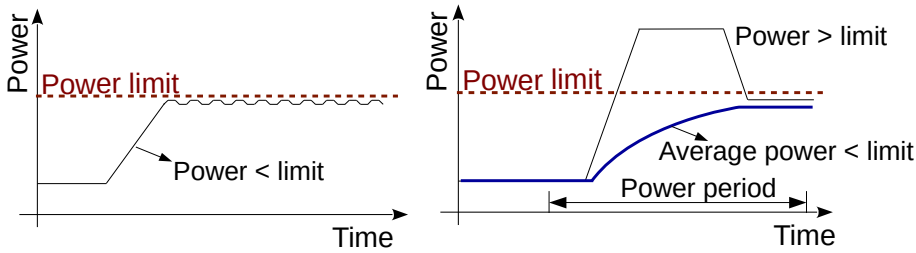
2.4.1 Scheduling vs Power Management

Scheduling refers to the activity of mapping an application to a particular core. In homogeneous CMPs, scheduling applications relies on factors such as the number of active tasks, the load per core, and the objective of the schedule. Power management is the activity of partitioning the power budget among the active tasks. This involves selecting the voltage-frequency operating point that guarantees high performance while remaining within the boundaries of the power budget.

These activities are distinct from each other, and modern OSes perform them through separate components. However, we use the two terms interchangeably throughout this thesis. As each chapter contributes to solving one part of the complex scheduling problem, the assumptions we make regarding the HCMP configuration and operation justify such usage of the terms. We assume that each application has access to the exact same set of core types. Each core type features the exact same set of voltage-frequency operating points. This implies that there is no competition on the core types among applications. Therefore, the selection of which core type an application uses (i.e., the schedule) and the voltage-frequency operating point it selects (i.e., power consumption) relies only on the portion of the power budget allocated to that application. An optimal schedule is a direct consequence of optimal power management.

2.4.2 Power Limits

Plenty of prior work assumes a basic definition of a power limit, whereby there is a predefined power consumption rate that should never be exceeded



(a) Basic power definition (b) General power definition

Figure 2.3: Conservative power definition generally used by power management techniques does not exploit the potential to temporarily exceed the limit to improve performance.

while the processor is running. This basic definition is illustrated in Figure 2.3a. This basic definition of a power limit applies only to a limited set of real-life scenarios. For example, when the power supply is provisioned to generate up to a specific power rating. Even if the processor is engineered to handle higher powers, a power limit is set. For practical scenarios, this definition is quite conservative as it prevents the processor from exploiting physically safe operating conditions to gain extra performance. For example, setting a maximum operating power for cooling purposes generally allows the specific cooling technique to dictate the rules for maintaining the power limit. Exceeding this power limit for a short time duration may not necessarily violate the cooling constraints of the chip.

We follow a more generalized definition of a power limit. A power limit is composed of two components. The first one is the power rate that must not be exceeded. The second component is a time period over which the power rate is calculated. In general, these values are defined by the processor manufacturer and are technology-specific. For example, manufacturers typically specify a power limit called the "thermal design point" (TDP), which is the maximum power a processor can consume without running into thermal problems. Despite the differences among major chip manufacturers on the definition of TDP, they agree that TDP is not the maximum power that must never be exceeded at any time during processor operation, similar to what is assumed by the conservative power definition. AMD for example views TDP as the maximum power that can be sustained when observed over a "thermally-significant period" [39]. This means that the processor can exceed this nominal value at runtime, as long as it does not exceed it for an extremely extended time duration that ends up burning the chip. Additionally, the processor must be allowed enough time after it exceeds the power limit to

cool down to an acceptable temperature that can be handled at TDP power. This more generic power definition is shown in Figure 2.3b.

There is not one single value for the power period over which power must be preserved. Techniques proposed by industrial and academic research to exceed TDP take thermal constraints into consideration. In these techniques, the time interval during which the processor is allowed to exceed TDP depends on the power rate above TDP the technique tries to achieve, and the thermal capacitance of the cooling infrastructure (e.g., heat sink and fan). These techniques are forced to lower their power consumption to the defined TDP for the remainder of the application execution. To safely operate at power rates that exceed the TDP, these techniques are required to execute the application below the TDP for a time duration that is sufficient to cool the processor down. In Turbo-boost 2.0 [7], the processor exceeds TDP by 1.2x-1.3x for 10s of seconds before cooling down. Similarly, in computational sprinting [6], the authors assume that the processor that can sustain 1 W with a single core can sprint for only a fraction of a second when activating 16 cores of 1 W each (i.e., 16x TDP). The processor then needs to cool down for a few seconds. They extend the duration of sprinting to 1 second by placing a block of phase change material as a heat sink, with around 20 seconds to cool down due to the material's high heat capacity. In our research, we look into a power period of 1 second. This means that we can exceed the power limit anytime during the power period, but the power consumption must adhere to the power limit when measured within 1 second of execution. This in no way affects the general conclusions of this thesis. The same techniques we propose apply to shorter or longer power periods. Our mathematical formulation assumes a generic finite time duration, as will be seen in the following chapters. We use 1 second only to simplify the calculations, reasoning, and mathematical derivation, and to shorten simulation time.

Note that our work focuses on power-limited processors. This means that our techniques aim to properly manage the power budget across the resources available on the processor chip, e.g., cores, network-on-chip, and caches. Power and thermal restrictions that are enforced at the processor level require managing how individual on-chip resources share the available power budget. On the contrary, power restrictions due to thermal constraints do not necessarily limit the power consumption of the memory system or other system-level components. Therefore, we focus on processor-level power management in this work. Managing the power budget across all system-level components, e.g., DRAM memory, processor and bus, is out of the scope of this thesis and is left as a future work.

2.5 Prior Work For Scheduling under a Power Limit

We went briefly over numerous scheduling techniques that optimize for various metrics by using both DVFS and HCMPs. In this section, we focus on scheduling techniques that optimize for performance under a power limit. Although a large body of research investigates the problem, no prior work looks at the problem from it numerous angles. We look closer at the most notable related techniques and show how they fail to consider one or more angles of the problem under consideration.

2.5.1 Responsiveness Techniques

Intel uses a technology called Turbo-boost to provide temporary performance boost via DVFS when needed. Turbo-boost 1.0 allows boosting the frequency of all the cores on chip as long as the power and temperature of the processor does not exceed the nominal TDP value [40]. Turbo-boost 2.0 [7] on the other hand allows temporarily reaching around 1.2x the TDP to significantly improve responsiveness at the initial phase of the application. The frequency is then decreased to ensure the sustained power remains under the TDP.

Computational sprinting [6] is similar in spirit to Turbo-boost 2.0 that is proposed to improve responsiveness of parallel applications. Instead of boosting frequency to levels that exceed TDP, this technique boosts performance by running a number of threads that utilizes all the cores of the processor. This operation resorts to turning off all the cores to abide by the safe thermal envelope. To increase the sprinting time of the application, and thus achieve higher performance, computational sprinting uses a block of phase-change material (PCM) on top of the processor due to its high heat capacity. For non-interactive applications, they propose to repeat the sprint operation indefinitely but turning off the processor for sufficient time to allow sprinting again afterwards. This sustained technique is called *sprint-and-rest* [41].

In comparison to these techniques, our work focuses on sustained performance rather than the initial performance burst. This means potentially preserving a burst of performance and power consumption to the parts of the application that improve sustained performance the most. Additionally, Turbo-boost relies on DVFS and computational sprinting relies on parallel cores. Both do not consider heterogeneous cores, and are not readily applicable to HCMPs. Finally, both techniques do not optimize performance of multi-programmed applications, and there is no clear approach to manage the several operating points available per core type. We provide in-depth comparison with these techniques when necessary in this thesis.

2.5.2 Multicore Sustained Performance

Techniques that try to maximize sustained performance under a power constraint are numerous [26, 42–46]. For example, Isci et al. [42] propose a global power management technique that selects the voltage-frequency operating point per application, such that the throughput of the CMP is maximized and the global power budget of the processor is respected. To find the best voltage and frequency setting per application, they perform a brute force approach by exploring all possible combinations. This scheme has high scheduling overhead and suffers from scalability problems. Other approaches try to find an optimal solution using linear programming solvers at runtime to find the best application to operating mode assignment, see for example [43]. However, the runtime overhead of this approach prohibits its usage as a practical solution especially with a large number of cores and operating points per core.

Other prior work thoroughly studies the scalability of power management schemes and proposes fast and scalable solutions to maximize performance under a power limit [26, 44–46]. However, all previously proposed works cannot be used as a solution to the scheduling problem we are considering in this thesis. All these approaches assume power management using only DVFS as the power adjustment knob. Therefore, these techniques are not directly applicable to HCMPs, in particular when each core has its own set of voltage-frequency operating points. This will become evident in Chapter 5, as we tackle the problems involved with the most generalized form of power management of HCMPs. Moreover, these approaches follow a basic definition of the power limit, and do not explore the benefits that can be achieved by spreading a power limit over a time period.

Chapter 3

Optimizing Performance for Single-Threaded Applications

3.1 Introduction

Heterogeneous chip-multiprocessors (HCMPs), e.g., ARM big.LITTLE, have been proposed for multiple purposes. The main use case for this architecture is to maximize energy efficiency at a tolerable performance degradation cost. For example, the application is scheduled to run on the little core when demand for performance is low, e.g., low user utilization, to save energy. When high performance is desired, e.g., heavy user interaction, the scheduler moves the application to the big core. Similarly for non-interactive applications, the scheduler identifies parts of the application that degrade the overall performance the least, and schedules them opportunistically on the little core [29, 35].

In this thesis, we focus on maximizing the performance of power-limited HCMPs. Power and thermal management techniques, e.g., Turbo-boost 2.0 [7] and computational sprinting [6], have been proposed to improve the performance of power-limited processors. However, these techniques mainly focus on improving the responsiveness of interactive applications. A more recent and relevant technique, called sprint-and-rest, was proposed to target the sustained performance of non-interactive applications [41], by extending the work on computational sprinting. Unfortunately, all these techniques were proposed for homogeneous chip-multiprocessors (CMPs) and lead to performance degradation when applied to heterogeneous processors.

Our goal in this chapter is to propose a scheduling technique that optimizes the utilization of the available power budget to maximize single-threaded

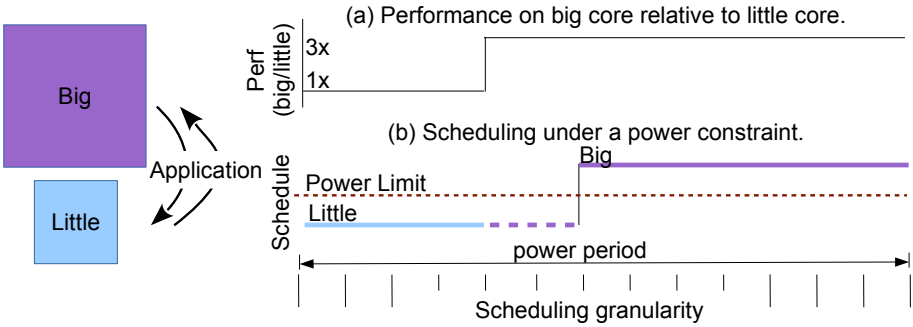


Figure 3.1: A scheduler needs to map each scheduling interval to the appropriate core type to maximize performance while remaining within the power limit.

performance. We show that sprint-and-rest does not exploit the diversity of core types introduced in HCMPs when optimizing the performance of single-threaded applications. We perform an exhaustive search to estimate the maximum performance that can be achieved by optimally scheduling the application phases on big and little cores under a power limit. Using this estimate, we demonstrate that exploiting heterogeneity leads to significant performance improvement over sprint-and-rest, under the same power budget.

We propose sprint-and-walk, a scheduling technique that extends sprint-and-rest to operate with heterogeneous cores. Sprint-and-walk manages to significantly improve performance over sprint-and-rest given the same power budget. Our results show that sprint-and-walk improves performance over sprint-and-rest by 9% on average across all SPEC CPU2006 applications, and reaches up to 19%, for a moderate power budget of 1.25 W. The improvement increases as the power budget gets lower. For a budget of 0.5 W, the average improvement of sprint-and-walk relative to sprint-and-rest equals 43% on average and reaches up to 76%. More importantly, we show that the performance of sprint-and-walk is within 3% of optimal performance. Sprint-and-walk maintains this near-optimal performance as we vary the given power budget, scheduling granularity, and HCMP configuration.

3.2 Problem Statement

We start by describing the assumptions we make about the processor under test and our definition of a power limit. We first assume the processor has no area constraints. This assumption allows us to use any microarchitecture

for both the big and little cores, and any desired mix of heterogeneous cores. We also assume the single-threaded application runs in isolation, eliminating the impact of interference through shared resources, e.g., the shared caches and the limited power budget. Both core types are always available, and the scheduler can migrate the application between the cores whenever it needs. Schedulers normally make their decisions once every scheduling interval. Figure 3.1 shows the configuration we use in this chapter.

We define a power limit similarly to Chapter 2. Under this definition, the power rate is estimated over a technology-dependent *power period*. This implies that the processor can dissipate power at a rate that exceeds the allowed limit at times, but needs to adhere to the limit over the totality of a power period. We assume the power limit has a value between the big and little core power ratings. A limit that exceeds the big core power allows sustained execution on the big core alone. On the contrary, a limit beneath the little core power prevents sustained execution even on the little core. Both cases are not interesting scheduling problems and will not be considered in this study.

Figure 3.1 sketches the single-threaded scheduling problem under consideration. The horizontal axis represents the application's execution during one power period. The period is divided into scheduling intervals, each of which can be scheduled either on the big or little cores. The curve in Figure 3.1(a) shows the performance gain for running the application on the big core relative to the little core for each scheduling interval. The curve in Figure 3.1(b) shows a schedule under a power limit. According to the shown schedule, the application runs initially on the little core, consuming power below the allowed rate. The scheduler runs on the big core towards the end as the later scheduling intervals seem to benefit more from executing on the big core than the earlier ones. The scheduler needs to estimate the fraction of time to run on the big core such that it does not exceed the power limit when averaged over the power period. Note that there are few scheduling intervals, represented by the dashed line in the schedule, that highly benefit from using the big core but have to be scheduled on the little core to keep the power consumption under the allowed limit.

Given the above description of the context, the scheduling problem becomes: **For each scheduling interval, which is more beneficial for overall performance: run on the big core to boost performance but burn excessive power, or run on the little core to save power that allows the application to utilize the big core at a later interval that boosts performance more?** The end solution is a mapping of each scheduling interval to a core type, such that power dissipation at the end of each power period remains below the power limit and performance is maximized.

Note that software schedulers usually make scheduling decisions at the granularity of an operating system time slice (e.g., a scheduling interval of 10 ms). Without loss of generality, we assume scheduling intervals based on a fixed number of instructions instead, to facilitate our analysis. A similar methodology has been used in prior works that propose hardware scheduling techniques that operate at a lower granularity than an operating system's time slice [35, 47, 48].

Comparing schedulers that take scheduling intervals of a fixed number of instructions as input can be done in different ways. Ideally for our case, a power period is divided into intervals of a fixed number of instructions, and schedulers migrate the application between the big and little cores up to the end of the period. The best schedule is the one that executes more instructions within the same time period. To reduce simulation time, a commonly used technique is to compare schedulers based on a representative task (i.e., fixed number of instructions) that is selected from the application. In this case, the best schedule is the one that finishes the task in the shortest time within the power limit, or the one with the highest performance assuming the repetitiveness of the same task to fill one power period. Because the selected tasks are highly representative of the application behavior, either of the two approaches would accurately estimate the performance of each scheduler under a given power budget.

We resort to comparing schedulers based on a fixed task as it significantly facilitates exploring HCMP scheduling techniques, and in particular the brute-force search method needed to assess the maximum performance attainable via scheduling. This exhaustive search is computationally challenging for a fixed representative task as we have seen. We expect exhaustive search over a much longer time duration (e.g., power period of 1 second) to be computationally intractable.

3.3 Scheduling under Power Constraints

We start by introducing sprint-and-rest, the most relevant approach for sustained performance maximization under a power limit. We demonstrate the shortcomings of sprint-and-rest when applied to HCMPs and propose sprint-and-walk to overcome these shortcomings. Finally, we describe the method we use to estimate the upper performance limit attainable via scheduling.

3.3.1 Sprint-and-rest

Computational sprinting [6] and Turbo-boost 2.0 [7] are techniques that target improving the application's responsiveness by starting it at the highest performance level. Computational sprinting targets multi-threaded applications and sprints by operating all the cores of a CMP concurrently, while Turbo-boost sprints by raising the clock frequency. In both cases, the application's responsiveness is boosted but the processor burns power at a rate that exceeds the safe limit. Therefore, the whole processor is either turned off completely or slowed down considerably to allow it to cool down to the safe operating limits.

Sprint-and-rest [41] is an extension to computational sprinting that targets non-interactive multi-threaded applications. During one power period, it starts sprinting on all cores and then turns off the processor to bring the power rate down to the safe limit. It repeats this sprinting and resting cycle throughout the application's execution. This technique has been shown to be more energy-efficient than sustained execution, i.e., turning on as many cores as allowed by the power limit without having to turn them off. The main benefit comes from eliminating the heavy cost of leakage power associated with turning on a partial set of cores, e.g., leakage power from the large caches and the network-on-chip.

Sprint-and-rest can be directly applied to HCMPs running single-threaded applications. During a power period, the application first runs on the big core (sprint) until the power budget is completely consumed. The processor is then turned off for the remainder of the power period (rest). This technique can run the application indefinitely under a power limit. However, it does not take advantage of the power-performance opportunities offered by the little core. Our analysis shows that switching to the little core instead of completely shutting down during the rest phase is more beneficial to performance under the same power limit.

Figure 3.2 plots the performance-power space for the baseline HCMP configuration (see configuration details in Section 3.4). The performance in billion instructions per second (BIPS) is shown on the horizontal axis, and the power in Watt is shown on the vertical axis. The performance and power values shown in the figure are averaged across all SPEC CPU2006 applications. We report the total power of the processor by summing the runtime dynamic power and leakage power, and assuming idle cores are power-gated. We study heterogeneous configurations composed of only two different core types (i.e., a big and a little core). We represent each core type with a single point on the figure, using its performance and power as coordinates. On average, the little core consumes 0.25 W for a performance of 0.75 BIPS, while

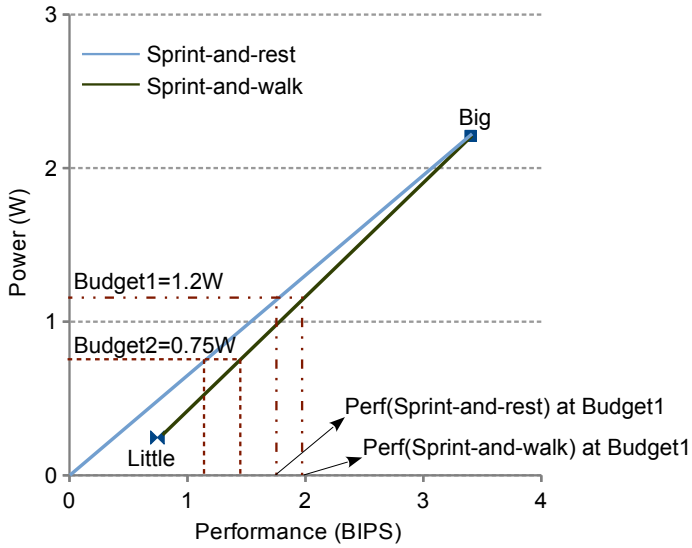


Figure 3.2: Sprint-and-walk yields higher performance than sprint-and-rest under any power budget.

the big core consumes 2.2 W for a performance of 3.4 BIPS. If the HCMP is given a power budget of 0.25 W, we expect it can sustain an application's execution on the little core. If the power budget is between 0.25 W and 2.2 W, the HCMP can conservatively execute the application on the little core without fully consuming the power budget. To exploit the whole power budget, the application can run on the little core for a fraction of the time period and migrate to the big core for the rest of the period. The fraction of time to run on each core type depends on the power budget and the scheduling technique. Figure 3.2 helps reason about the performance of a scheduling technique given a power budget.

Sprint-and-rest sprints on the big core and rests by turning the processor off. Therefore, its power-performance behavior on average is represented by the line connecting the big core point and the zero coordinate. Any point (x,y) on the line means that sprint-and-rest is expected to yield a performance "x" on average, when given a power budget "y" due to partially running on both core types. A technique that leverages heterogeneity by sprinting on the big core and migrating to the little core instead of turning off the processor have an average power-performance behavior that is represented by the line connecting the big core coordinate and the little core coordinate. The line is labeled as "sprint-to-walk" in the figure. A horizontal line represents a power budget. The figure shows two example budget lines at 0.75 W and 1.2 W. The intersection point between a power budget and the sprint-and-rest line

projects the expected performance on the horizontal axis.

It is clear that the intersection point of any power budget line with the sprint-and-walk line always happens further to the right, i.e., yielding higher performance, than the intersection with a sprint-and-rest line. For example, at a power budget of 0.75 W, sprint-and-rest degrades performance by around 24% relative to sprint-and-walk. The figure also shows that for loose power budgets, all scheduling techniques yield similar performance as they can run extensively on the big core. As the power budget gets tighter, the gap widens between the two techniques, showing a clear advantage for techniques that leverage heterogeneity. We expect the same conclusion to apply to CMPs that feature multiple pairs of big and little cores running multi-programmed workloads, and maintaining the LLC capacity provisioned per pair of cores.

3.3.2 Sprint-and-walk

We propose sprint-and-walk to utilize both core types in an HCMP, instead of shutting down the processor. Algorithm 1 describes how sprint-and-walk operates.

Sprint-and-walk is based on the concept of energy credit. A power limit **P** that a processor must preserve over a power period **T** implies that the processor has an energy budget **E = P x T** to spend over a period **T**. We are interested in power limits between the big and little core's power ratings. If the scheduler runs a scheduling interval on the little core, its energy consumption is lower than what sustained execution at the allowed limit would consume assuming the same time as need by the little core. On the other hand, the big core consumes more energy per scheduling interval relative to sustained execution at the allowed limit assuming the same time as needed by the big core. Therefore, running an interval on the little core allows the scheduler to accumulate energy credit equal to the difference between its consumption and the energy consumed during the same time at the allowed power limit. This energy credit can be used to run other future intervals on the big core. Running an interval on the big core burns more energy than the a run at the sustain power limit. The difference between the consumed budget and the allowed budget is taken from the previously accumulated credit.

In simple terms, Algorithm 1 runs the application initially on the little core to build up energy credit for a specific time duration. Then, it migrates to the big core and continues there until the accumulated energy credit is consumed. Running on the big core is the *sprint* part, while accumulating credit on the little core is the *walk* part. Repeating the sprint-and-walk cycle allows sustained execution of an application on a power-limited HCMP. We

Algorithm 1 Sprint-and-walk. Accumulate energy credit by running on little, then run on the big core until credit is dissipated.

```
Divide application into intervals of fixed instructions
power_consumed = 0, energy_credit = 0
credit_build_time = 0, threshold = 10 ms
remaining_time = One power period
while remaining_time >0 do
    take the next interval
    if energy_credit<= 0 OR credit_build_time<threshold then
        run on little
        remaining_time = remaining_time - time of interval on little
        if energy_credit >= 0 then
            credit_build_time += time of interval on little
        end if
        energy_credit+=interval time*(power_limit - interval power)
    end if
    if energy_credit >0 AND credit_build_time >= threshold then
        Run on Big
        remaining_time -= time of interval on Big
        energy_credit-=interval time*(interval power - power_limit)
        if energy_credit <= 0 then
            credit_build_time = 0
        end if
        remaining_time = remaining_time - time of interval on Big
    end if
end while
```

use 10 ms to build energy credit because it is frequent enough to sample the different phases of the application, and long enough to amortize the migration overhead. The algorithm preserves the power limit as it guarantees not running above the limit unless enough energy credit is available.

3.3.3 Optimal Performance: Oracle

As scheduling decisions are taken at the granularity of every scheduling interval, optimal scheduling requires complete knowledge of the application power and performance on both core types for every scheduling interval. With this knowledge, a scheduler has to rank the intervals according to their benefit from running on the big core. Assuming all the intervals start on the little core, the scheduler can proceed to move them to the big core, one interval at a time, until it estimates it would exceed the power limit. The resulting schedule would be optimal.

	Core 1	Core 2	Core 3	Core 4
Pipeline width	4	4	2	2
Type	Out-of-order	Out-of-order	Out-of-order	In-order
Frequency	2.6 GHz	1.5 GHz	1.5 GHz	1.5 GHz
Voltage	0.9 V	0.64 V	0.64 V	0.64 V
ROB size	168	168	32	-
L1 I-cache	32 KB			
L1 D-cache	32 KB			
Shared L2 cache	4 MB per pair			
Memory B/W	25.6 GB/s			

Table 3.1: Core configurations considered in this study.

	Core 1	Core 2	Core 3	Core 4
	Baseline	Config. 2	Config. 3	Config. 4
HCMP config.	core 1 (big)	core 1 (big)	core 2 (big)	core 3 (big)
	core 4 (little)	core 3 (little)	core 4 (little)	core 4 (little)
Power limit	1.25 W	1.5 W	0.6 W	0.4 W

Table 3.2: HCMP core mixes and power limits considered in this study. Table 3.1 details the core configurations.

We carry out a brute-force search through all possible schedules. For each application, we divide its representative code into *n* scheduling intervals. The default interval consists of 1 million instructions (see Section 3.4 for details on methodology). We search through all possible combinations where each interval could be scheduled on either the big or little cores. This means calculating the power and performance of 2^n schedules. The optimal schedule is the one that finishes the execution in the shortest time while remaining under the allowed power rate. We refer to this approach as the *oracle* throughout this study.

3.4 Evaluation Methodology

We analyze all SPEC CPU2006 applications using all their inputs sets, for a total of 55 benchmarks. We use SimPoint [49] to find a representative code

segment of 750 Million instructions. We use Sniper 6.0 [50] with its most detailed cycle-level core modes to carry out the performance evaluation for all applications across all HCMP configurations under test. Moreover, we use McPAT [51] to model power consumption assuming a 22 nm technology. We report the total power of the processor including leakage power and runtime dynamic power, assuming idle cores are power-gated.

The baseline configuration we use in this study and carry throughout the thesis deploys a 4-wide aggressive out-of-order core running at 2.6 GHz and 0.9 V as the big core. For the little core, we use a 2-wide in-order core running at 1.5 GHz and 0.65 V. We carry out experiments on HCMPs composed of other core mixes. Table 3.1 provides an overview of all core types we use in this chapter. Table 3.2 shows the four HCMP configurations and their respective power limits.

As mentioned in Section 3.2, we study only power limits that fall between the big and little power ratings. To select an appropriate limit for our HCMP configurations, we run each application on all four core types we consider in this study. We measure the power rating for each application on each core type. For each HCMP consisting of two different core types we calculate the average power between the big and the little core power ratings. Finally, we take all the average power ratings for each application and calculate the median of these values as the power budget. Without lack of generality, we assume this power budget must be consumed over a power period of 1 second. Table 3.2 shows the power budgets we use in this chapter. Our baseline configuration experiments assume a budget of 1.25 W. A similar value has been assumed in prior power-limited design works such as computational sprinting [6].

We evaluate HCMPs that feature two core types integrated on a chip without any special micro-architectural techniques aiming at lower migration overheads, such as the ones used in [35, 48]. Therefore, we assume that scheduling decisions are made at a relatively coarser granularity than the 1 thousand instructions used in these prior studies. We assume scheduling intervals of 1 million instructions. At an estimated migration overhead of 20 microseconds [1], we expect the switching overhead to be around 2% if migration happens at every scheduling interval. This migration overhead is considered sufficiently low not to impact the general findings of this study, and can thus be ignored. For completeness, we study the impact of shrinking this scheduling interval to 10 thousand instructions and expanding it to 10 million instructions where the scheduling overhead is negligible.

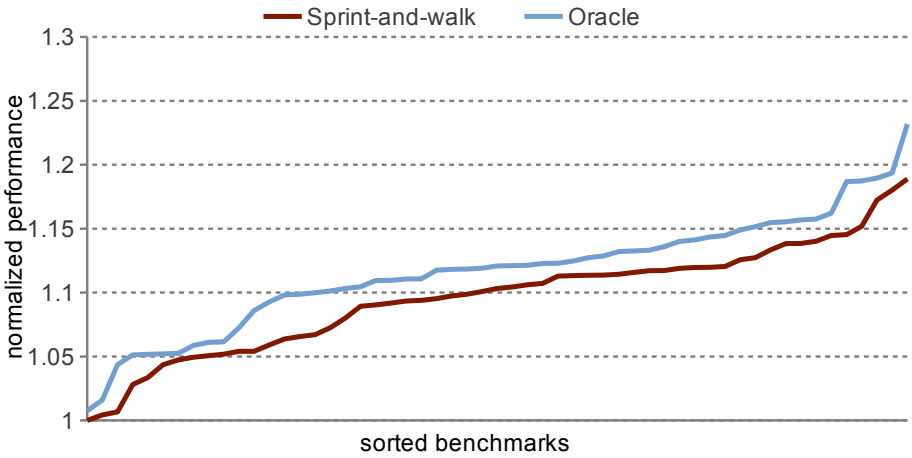


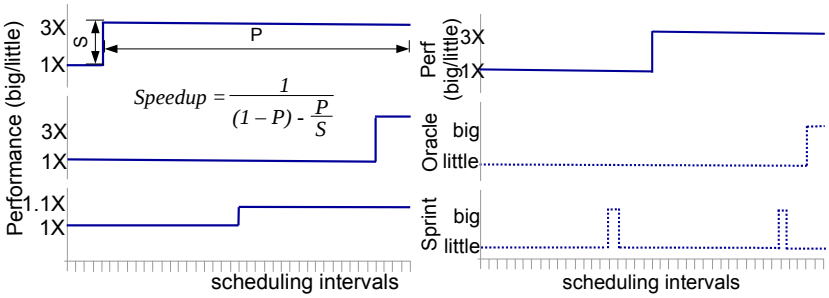
Figure 3.3: Performance for oracle and sprint-and-walk normalized to sprint-and-rest. Both methods improve performance significantly over sprint-and-rest. Sprint-and-walk achieves near-optimal performance.

3.5 Experimental Results

In this section we show the performance improvement of both sprint-and-walk and oracle over sprint-and-rest. First, we show performance results assuming the baseline HCMP and power limit, and then we discuss the factors that impact these results. We perform a design space exploration to account for the impact of diverse core configurations, power limits, and scheduling granularities on performance.

3.5.1 Potential Performance Improvement

Figure 3.3 shows the performance of oracle and sprint-and-walk normalized to sprint-and-rest for the baseline configuration. Both sprint-and-walk and oracle improve performance significantly over sprint-and-rest. Moreover, there is a small gap between the performance curve for sprint-and-walk and oracle for all the applications. The similarity between the two curves demonstrates the significant resemblance between the two methods across all applications. Sprint-and-walk improves performance by 9% on average and up to 19%, over sprint-and-rest, while oracle yields an average improvement of 11% that reaches up to 23%. As previously demonstrated in Figure 3.2, sprint-and-walk is expected to yield even higher performance improvements over sprint-and-rest as the power budget decreases. The following subsections discuss potential factors that lead sprint-and-walk and oracle to yield similar performance.



(a) Weak phase behavior (b) Impact of the power limit

Figure 3.4: Reasons for limited performance benefits for oracle scheduling.

I. Applications with Weak Phase Behavior

We observe that several applications show weak or no phase behavior. This corroborates findings from prior studies such as [52]. Figure 3.4a shows several cases of weak phase behavior. The vertical axis show the performance improvement of the big core over the little core for every scheduling interval. We include Amdahl's speedup law in the figure to show two kinds of phase behavior. The top and middle lines show cases where the application does not see a change in it behavior for almost the entire period. This implies that regardless of which intervals the oracle selects, the performance benefit beyond sprint-and-walk is limited. In Amdahl's terms, the limited benefit of oracle is due to the insignificant fraction of enhancement. For the bottom line case, there are two considerable phases of execution, the speedup ratio 'S' in Amdahl's equation is quite low. Therefore, the speedup of oracle over sprint-and-walk remains limited by Amdahl's law.

II. Power Limit Reduces Speedup

For applications that exhibit significant phase variation, oracle may still see limited improvement over sprint-and-walk due to the imposed power limit. Figure 3.4b demonstrates such a case. The application has two main phases that differ considerably from each other. However, the power limit is too strict, forcing both oracle and sprint-and-walk to run extensively on the little core for sustained operation. If the power limit is too loose (not shown), both schedulers can afford to run extensively on the big core during the power period. In either case, the performance of sprint-and-walk remains close to oracle.

3.5.2 Sensitivity Study

The power limit has a major impact on the performance improvement for both sprint-and-walk and oracle over sprint-and-rest and it may also affect the gap between oracle and sprint-and-walk. We show the impact of varying the power limit above and below the baseline limit. We also examine the impact of the selected scheduling granularity and HCMP configuration.

I. Power Limit

We explore power limits around the median point of 1.25 W, ranging from 0.5 W to 1.5 W by increments of 0.25 W. Figure 3.5 shows sorted performance for oracle and sprint-and-walk normalized to sprint-and-rest, under various power limits. The figure conveys a similar message to the one observed in the baseline scenario. Both oracle and sprint-and-walk improve performance significantly over sprint-and-rest, especially for tight power limits. As predicted by Figure 3.2, with the increase in the power budget, all the methods converge due to their ability to schedule more intervals on the on the big core. For power budgets of 0.5 W, 0.75 W, 1 W, 1.5 W, sprint-and-walk improves performance by an average of 42%, 24%, 15%, 7% and up to 76%, 44%, 28%, 40%, respectively. For the same budgets, oracle improves performance by an average of 46%, 27%, 17%, 9% and up to 78%,48%, 32%, 45%, respectively. When the budget exceeds the power consumption of the big core, all schedules have the ability to run on the big core throughout execution. Sprint-and-walk starts by running initially on the little core to accumulate energy credit, this may causes a few percents of performance degradation compared to the other two methods when the application can sustain big core execution within the power budget.

The curves in Figure 3.5 as well as the performance improvement numbers show that the resemblance between sprint-and-walk and oracle persists across the different power budgets. This implies that sprint-and-walk provides near-optimal performance across all applications despite the change in power budgets.

II. Granularity

So far, we picked intervals of 1M instructions as our baseline scheduling granularity. Increasing the granularity over 1M lowers the overheads associated with making scheduling decisions and migrating the application from one core type to another. However, this comes at the expense of lowering the accuracy at which finer granularities can track the application phases,

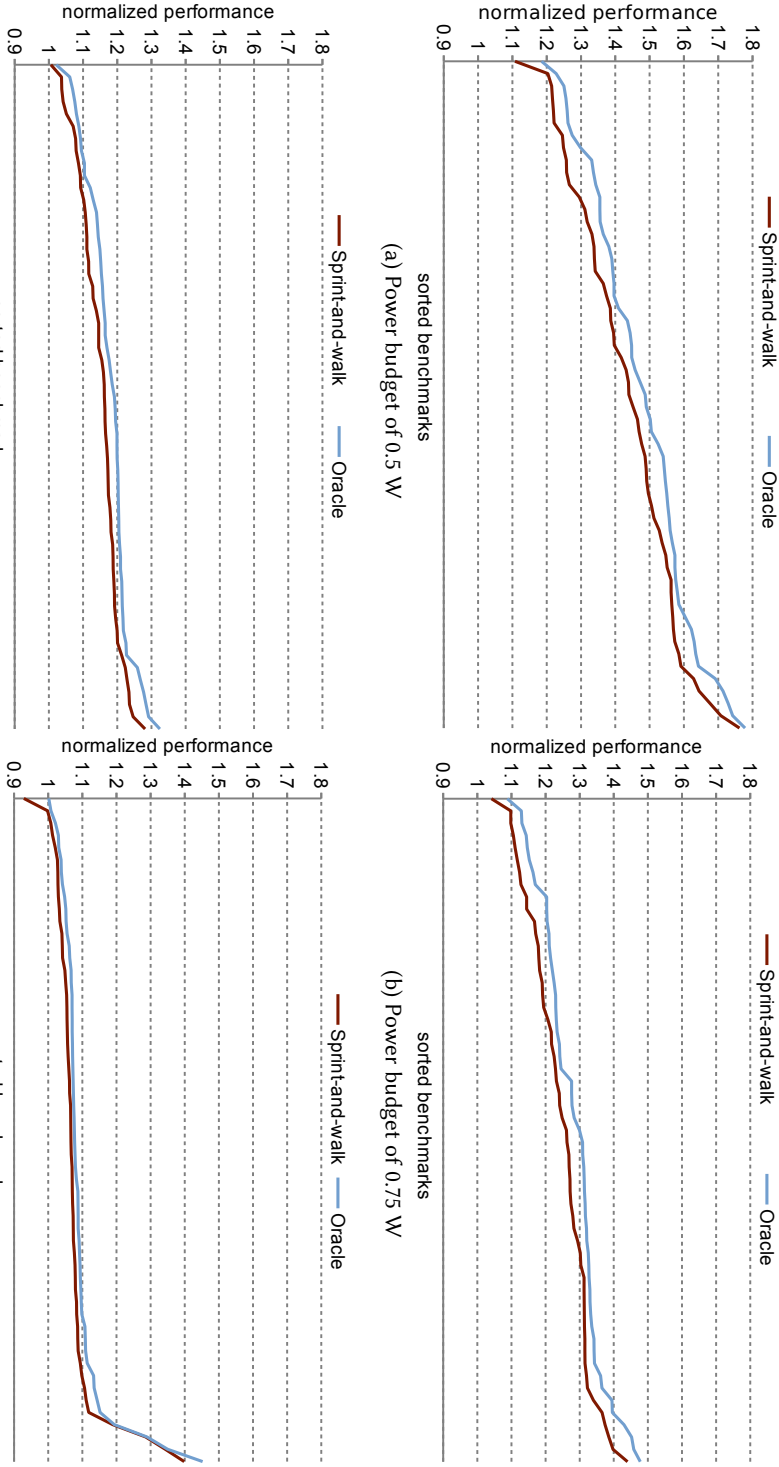


Figure 3.5: Normalized performance for sprint-and-walk versus oracle, relative to sprint-and-rest, for power budgets: (a) 0.5 W, (b) 0.75 W, (c) 1.0 W, and (d) 1.5 W.

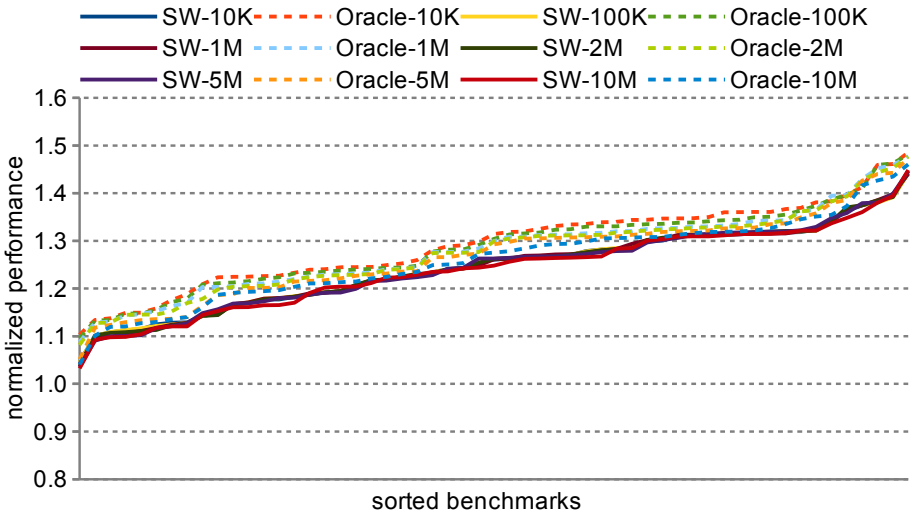


Figure 3.6: Performance for oracle and sprint-and-walk normalized to sprint-and-rest for various scheduling granularities.

which could lead to a higher performance at the same power limit. In this section, we experiment with 10 K, 100 K, 1 M, 2 M, 5 M, and 10 M instruction granularities.

Figure 3.6 shows the normalized performance for oracle and sprint-and-walk over sprint-and-rest at a power budget of 0.75 W for the various scheduling granularities. For each granularity we show two curves, one for sprint-and-walk and one for oracle. As the curves are close to each other, we show the oracle curves in dashed format. All the curves are normalized to the same performance achieved by sprint-and-rest, which is measured mathematically, i.e., the switches from the big to the little core happens immediately when the budget finishes. This corresponds to switching at the finest granularity. We select 0.75 W because the gap between sprint-and-walk and oracle for both the average and maximum performance is slightly wider than other budgets, making it an interesting case to study. Other budgets behave similarly as we have seen in Section 3.5.2(I.) and lead to similar conclusions.

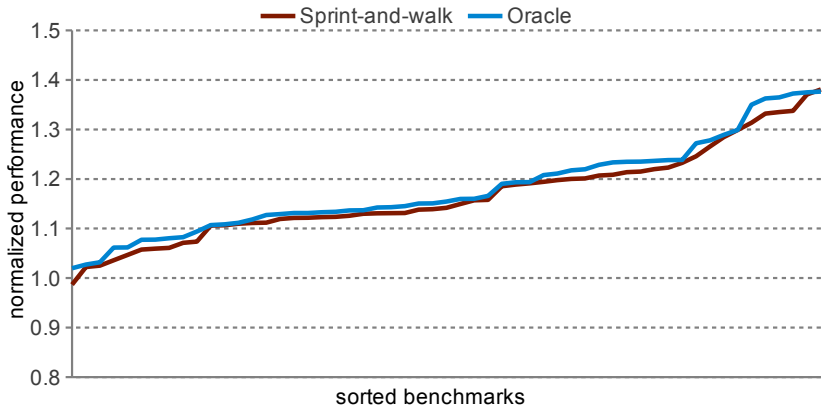
The figure shows that the finer the granularity, the more accurately oracle tracks the application phases, leading to higher performance improvement over sprint-and-rest. However, the performance improvement is insignificant. On average, oracle's performance improvement goes from 25% at a granularity of 10 M instructions to 27% at 1 M instructions and 29% at 10 K instructions. This result corroborates prior works [35, 48] that also suggest that a higher improvement is expected at granularities beneath 1 K instructions. However, operating even at the fine granularities considered in this

work requires a significant change to the underlying core microarchitecture to elude the high migration overhead. On the other hand, sprint-and-walk does not show the same sensitivity to the change in scheduling granularity, as its algorithm has a limited freedom since it migrates to the big core only at when enough credit is accumulated. On average, sprint-and-rest improves performance by 23% at a granularity of 10 M instructions, but saturates at 24% for all the finer granularities. The take-away message is that even when ignoring the migration overheads of an oracle at fine granularities, the gap with sprint-and-walk remains insignificant.

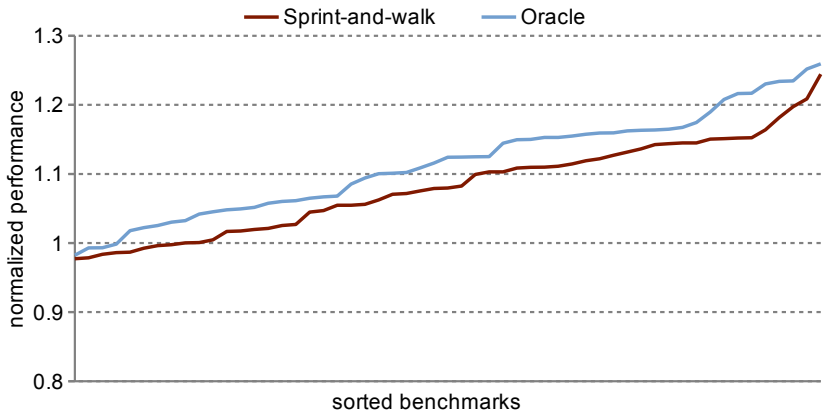
III. HCMP Configuration

Our baseline HCMP pairs a 4-wide aggressive out-of-order core with a 2-wide in-order core. These cores are at two opposite ends of the power-performance spectrum. We investigate whether such a configuration biases the conclusions we have seen so far. For example, if the big core consumes too much power whenever used, the scheduler is forced to run more frequently on the little core, leaving little room for optimization. Similarly, the wimpy little core can degrade performance significantly, overshadowing the scheduler's attempts to improve performance on the big core. We explore the results for three other HCMP mixes that vary the core complexities and voltage-frequency operating points as shown in Table 3.2. In the table, configuration 1 is the baseline configuration. Configuration 2 mixes a 4-wide out-of-order core running at 2.6 GHz (big) with a 2-wide out-of-order core running at 1.5 GHz (little). We reduce the frequency of the 4-wide core to 1.5 GHz (big) and mix it with the in-order core (little) to form configuration 3. Finally, configuration 4 uses the 2-wide out-of-order core running at 1.5 GHz (big) and the in-order core (little).

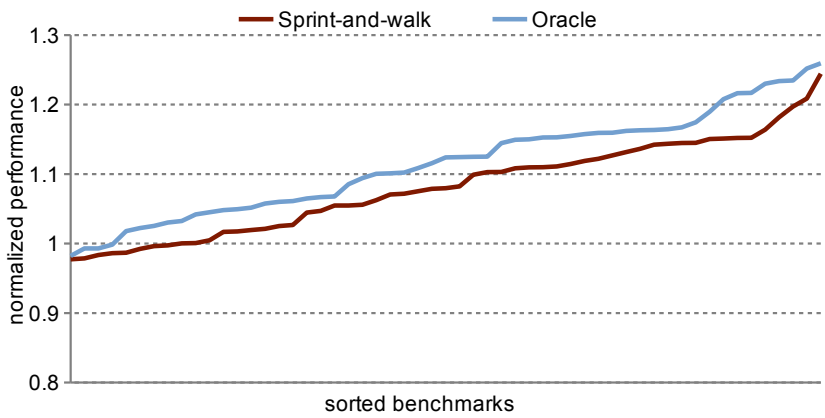
Figure 3.7 shows performance for oracle and sprint-and-walk relative to sprint-and-rest for the three HCMP configurations. The curves in Figure 3.7 resemble the ones shown for the baseline configuration. On average, oracle improves performance by 17% for configuration 2, 11% for configuration 3, and 9% for configuration 4. Sprint-and-walk improves performance of the same configurations by 16%, 8%, 6%, respectively. This shows again a gap of 1-3% between sprint-and-walk and oracle on average. Interestingly, we notice that three to four cases in configurations 3 and 4 benefit more from sprint-and-rest more than both techniques that use the little core. Both configurations have a relatively weak big core operating at a relatively low frequency, making its power consumption relatively low. For those specific applications, the little core's performance is considerably lower than the average but their power consumption remains close to the average. This



(a) HMCP configuration 2: core1 big, core3 little



(b) HMCP configuration 3: core2 big, core4 little



(c) HMCP configuration 4: core3 big, core4 little

Figure 3.7: Performance for oracle and sprint-and-walk normalized to sprint-and-rest for the various HCMP configurations from Table 3.2.

makes the little core a less efficient option than turning off the processor. Note that the various HCMP configurations assume different power budgets suitable for each configuration. The figures are intended to show the near-optimal performance of sprint-and-walk remains across different HCMP configurations. The results cannot be used to make conclusions on which configuration provides better performance.

3.6 Summary

In this chapter we focused on maximizing single-threaded performance of HCMPs under a power constraint. Power management techniques for single-threaded applications, such as Turbo-boost 2.0, and multi-threaded applications, such as computational sprinting, focus on responsiveness. Sprint-and-rest was proposed for sustained performance under a power budget. When applying sprint-and-rest to power-limited HCMPs, the processor sprints on the big core until it consumes its power budget, after which it rests by turning off the processor, and repeats the cycle throughout the application execution.

We show that sprint-and-rest does not handle the limited power budget properly when deployed on a heterogeneous processor because it fails to utilize the available heterogeneity. We propose a new method, called sprint-and-walk, that extends sprint-and-rest and utilizes both core types of the HCMP. We show that sprint-and-walk significantly improves performance over sprint-and-rest when given the same power budget. More importantly, we show that sprint-and-walk achieves near-optimal performance. To estimate the maximum performance attainable via scheduling under a power limit, we exhaustively search all possible schedules. Our experiments show that sprint-and-walk improves performance over sprint-and-rest by 9% on average and up to 19%. Oracle improves performance by 11% on average and up to 23%. Moreover, the gap between the two techniques is insignificant across all SPEC CPU2006 applications. Further more, our extensive analysis shows that sprint-and-walk is a robust solution that remains within a few percents of optimal performance as we vary the power budgets available to the processor, the scheduling granularities, and the HCMP core configurations.

Chapter 4

Optimizing Performance for Multi-Programmed Workloads

4.1 Introduction

In the previous chapters we introduced Heterogeneous chip-multiprocessors (HCMPs) as way to extend the shrinking range of operating points available through DVFS. In the era of dark silicon, HCMPs provide a selection of core types that allows the processor to better cope with stringent power limits, and gain extra performance under a given power budget. Both academia and industry have proposed heterogeneous chip multi-processors that consist of multiple high-performance but power-hungry 'big' cores and multiple power-efficient but low-performance 'little' cores. Recent commercial HCMP offerings include Samsung's Exynos series starting from Exynos 5 [13] till the latest Exynos 9 [53], NVIDIA's Tegra-4 [14]/Tegra-K1 [54]/Tegra-X1 [55], MediaTek's Helio X20 [16], and Intel's QuickIA [15]. In Chapter 3, we focus on utilizing HCMPs to maximize the performance for single-threaded applications under power constraints. In this chapter we target optimizing the performance of HCMPs when running a multi-programmed workload under a power constraint.

Performance and power consumption of an HCMP is a function of the application to core mapping, with time spent on the big core being the determining factor. As a result, significant research work has focused on a dynamic scheduler that selects the appropriate core type to optimize per-

formance [32, 34, 38, 56–58] or energy efficiency [33, 35, 59]. Unfortunately, none of these prior works take power constraints into account.

We focus on HCMPs with a constrained power budget, i.e., the processor cannot consume more than a fixed power budget over a specific time period (e.g., n Watt per m seconds), which is dictated by design parameters. Under such power budget constraints, applications can be executed on the big core only when sufficient power budget is available. Otherwise, the application must be executed on the little core[1]. Consequently, application performance on such systems directly depends on effectively consuming the available power budget (which is a function of the application's power consumption on the big core).

Intuitively, the power budget should be distributed among concurrently executing applications based on utility, i.e., the ability for an application to execute a large fraction of the defined time period on the big core. If an application can execute a larger fraction of the power period on the big core, it should be given a larger share of the power budget compared to an application that can run less on the big core and thus benefit less from running on the big core. With this in mind, we make the following contributions in this chapter:

- We formulate the performance optimization challenge on power-constrained HCMPs as a linear programming problem. We show that the optimal solution is a schedule where each application runs on either a big or a small core and exactly one application runs partially on both.
- We show that to obtain optimal performance on power-constrained HCMPs, big core resources should be given to applications with the highest Delta Performance / Delta Power (DPe/DPo), i.e., the ratio of the performance delta and the power delta between the big versus little core.
- We propose DPDP power budget partitioning, a novel policy that dynamically ranks and schedules applications to big and little cores based on the DPe/DPo metric. Our proposal uses the insight of the linear program solution to design a scalable power budget partitioning policy, that is proven to be optimal in an offline scenario.
- A surprising (perhaps counterintuitive) finding is that memory-intensive applications tend to be preferred (over compute-intensive applications) to run on the big core in power-constrained environments.

1. In our setup, we assume the power consumption of the little core never exceeds the power constraints, similar to the sustained workloads case in [17].

> Because memory-intensive applications consume less power on the big core than compute-intensive applications, they can run a longer fraction of time on the big core before having to migrate to the little core. Therefore, in many cases, they better leverage the power budget to improve performance than compute-intensive applications.

Our evaluations with DPDP on a 4-core heterogeneous processor consisting of big.LITTLE pairs show that DPDP improves chip performance by 16% on average and up to 40% over a strategy that greedily and globally optimizes the power budget. We demonstrate that DPDP outperforms schedulers based on commonly used heuristics such as performance ratio and performance per Watt. We also show that DPDP is scalable to different core counts, core types, and power budgets. Moreover, we analyze the impact of DPDP on per-application performance and we propose a technique to enforce a user-defined tolerable slowdown. Our results show DPDP's ability to maximize performance while maintaining the desired latency requirements.

4.2 Motivation

4.2.1 Implications of Power Limits on HCMP Scheduling

We define a power constraint as the maximum power consumption averaged over a certain time interval, similar to the way we define it throughout this thesis. This means that power consumption can temporarily exceed this limit, as long as it is followed by a lower power phase to ensure that the average is within the limit. This is different from prior work [26, 42, 44–46], which typically assumes a strict power limit at every moment in time. This definition is motivated by recent work on thermal management [6, 7]: heating because of high power consumption happens gradually and has a certain delay (thermal time constant). As a result, chip temperature is determined by the average power consumption over this time period, rather than the instantaneous power consumption. We conservatively set the power period to one second, but our technique can handle any time period setting (as long as it is long enough compared to the core migration time).

In our HCMP setup, this power constraint definition means that we can execute more programs on the big cores than the power budget allows, followed by a migration to the little cores to compensate for the overconsumption. Therefore, HCMP power management should consider both the performance and power characteristics of each program on each core type.

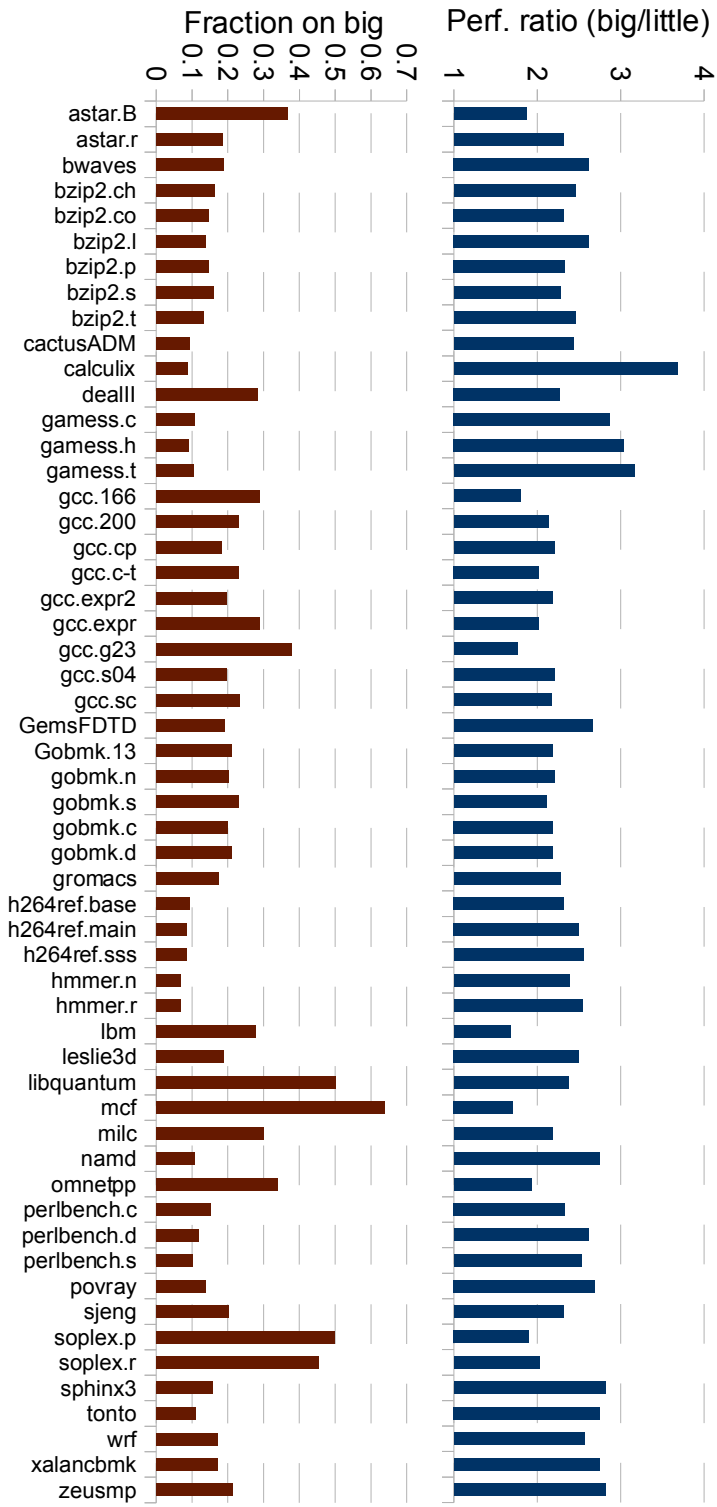


Figure 4.1: The big/little performance ratio (top graph) and fraction of time each application is allowed on the big core based on a 1 W per 1 s power budget (bottom graph).

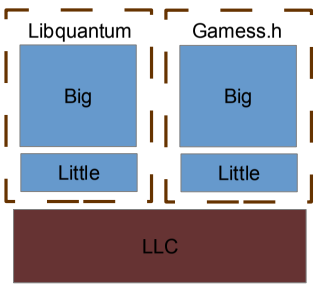
Assuming no power constraints, Figure 4.1 (top graph) shows the performance advantage for the SPEC CPU2006 benchmarks when running on the big core relative to the little core[2]. Under no constraints, applications can observe anywhere from 2 to 4× better performance on a big core relative to a little core. However, on a power-constrained HCMP, the budget limits how long the application can execute on the big core. Once the power budget is depleted, the application must be executed on the little core. Assuming a power budget of 1 Watt to spend over 1 second per application, Figure 4.1 (bottom graph) illustrates the fraction of the total execution time each application can execute on the big core. Under power constraints, we observe that applications can spend as little as 10% of the total execution time on the big core (e.g., hmmer), or as much as 60% of the total execution time on the big core (e.g., mcf). The varying behavior among workloads is primarily due to the difference in power consumption on the big core. In general, we find that memory-intensive applications tend to have lower power consumption on the big core since they spend a large fraction of the execution time stalled waiting for memory, which enables them to spend more time on the big core for a given budget.

4.2.2 Power Budget Partitioning

Based on the observations from the previous section, we now show how prior proposals are unsuitable for power-limited HCMPs environments. Figure 4.2 shows an example heterogeneous multicore featuring two big and two little cores, concurrently running two applications: gamess.h and libquantum. The power consumption of both applications on each core type is provided as well. For this example, we assume a power budget of 2 Watt over a period of 1 second (1 Watt per big.LITTLE pair).

The gamess.h benchmark is a compute-intensive workload that significantly benefits from the big core (3× performance), but if it may only consume 1 W, it can be run on the big core for 0.09 seconds only, and for the remaining 0.91 seconds, it has to run on the little core, because of its relatively high power consumption on the big core. Although the memory-intensive libquantum does not benefit from the big core as much (2.3× performance), its relatively low power consumption allows it to run for 0.5 seconds on the big core for the same 1 W power consumption. Overall, libquantum achieves about 40% higher performance than gamess.h (both relative to little core) when both are given the same 1 W per 1 s power budget.

2. Throughout Section 4.2, we assume an out-of-order little core. In Section 4.6 we show results for both in-order and out-of-order little cores (see Section 4.5 for our experimental setup).



	Libquan.	Gamess
Power Big (W)	1.34	3.35
Power Little (W)	0.56	0.75
Perf. ratio	2.37	3.04

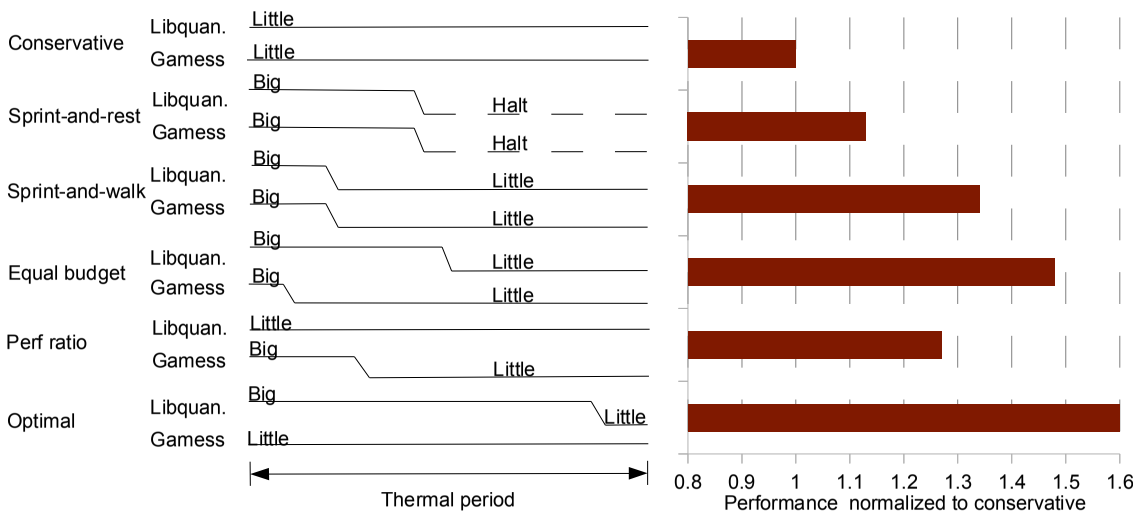


Figure 4.2: Performance gain for several budget partitioning approaches normalized to running all applications on the little cores.

Figure 4.2 also shows the performance (drawn to scale) for several HCMP scheduling approaches. While not all of these approaches explicitly partition the power budget, the application to core mapping indirectly partitions the power budget based on which and when applications execute on a big core:

- The **Conservative approach** interprets the power budget as a strict power limit (total power cannot exceed 2 Watts at any time). If the total power consumption of executing one or more applications on the big core exceeds the power budget (as is the case in our example), the applications can only execute on the little core. Consequently, this approach does not utilize the available power budget and hence yields suboptimal performance. This approach is taken by most DVFS-based CMP power capping studies [26, 42, 44–46].

- **Sprint-and-rest** is similar to computational sprinting for long-running applications [6, 41]. Here, we execute all applications on the big core to obtain the highest performance, and as soon as we have consumed the available budget, the HCMP is turned off to cool down.

- **Sprint-and-walk** follows our definition in Chapter 3. For multiple cores, this approach is similar to sprint-and-rest but after sprinting both applications on the big core, we move both of them to the little cores such that the total budget is still preserved. It is clear that the fraction spent on the big core will shrink compared to sprint-and-rest to provision for the run to continue on the little cores. This is the HCMP scheduling variant of Intel's Turbo-boost 2.0 [7], which increases the frequencies of all cores if there is thermal headroom.

- **Equal budget partitioning** divides the power budget equally among the applications (each getting 1 W per 1 s). Here each application spends a different fraction of the time on the big core based on its power rates on the big and little cores.

- **Performance ratio** ranks the applications by their big-to-little performance ratio. We always run the lowest ranked application on the little core while the highest ranked application gets the remainder of the budget (which allows it to run a fraction of the time on the big core). This is a common approach for scheduling in HCMPs.

- **Optimal** system performance is achieved by favoring libquantum over gamess.h, i.e., run gamess always on little, and give the remaining budget to libquantum to run on the big core.

The suboptimal performance observed for the various scheduling policies is mainly due to being application-unaware. Both sprint-and-rest and sprint-and-walk let all the applications greedily compete for the budget: the applications with higher power consumption rates deplete most of the budget leaving the lower power applications with a smaller fraction of the budget despite being better at utilizing it. Similarly, although the performance ratio approach tries to optimize where to allocate its budget, ignoring the power limits restricts the time spent on the big core, leading to a wrong prediction of which application would benefit the most from the given budget. Although equal budget partitioning provides an equal chance for both applications, it fails to reach optimal performance because the budget given to gamess.h is depleted quickly, not benefiting its total performance significantly. However, when prioritizing libquantum, its memory-intensive nature leads to lower power consumption that results in an overall higher utilization of the big core, and this leads to higher overall system performance. The bottom line is that **application awareness is essential to partition the available power budget among co-running applications to maximize overall system performance**.

Maximizing performance in power-constrained HCMPs mandates optimally tuning the fraction of time each application gets on the big core, which comes down to searching through an infinite number of possible fraction allocations. This analysis clearly motivates the need for a new optimal and scalable mechanism for partitioning the available power budget across concurrently executing applications. To that end, the next section formulates the power budget partitioning problem using linear programming, which yields a practical, yet well-performing algorithm.

4.3 Power Budget Partitioning using Linear Programming

As shown in the previous section, partitioning the power budget across applications to optimize performance is not straightforward. A partitioning policy should take into account both the performance gain of an application on the big core, as well as the fraction of time it can spend on the big core, which is determined by its power consumption. Instead of trying out various heuristics, we take a more rigorous approach by formulating the problem statement using linear programming. Note that the power manager itself does not need to solve a linear program during runtime. Instead, the key insight from the mathematical formulation leads to a solution that enables a low-overhead scalable power manager to dynamically find the optimal

schedule and power distribution among the applications in the large design space.

4.3.1 Linear Programming Formulation

To formulate power budget partitioning as a linear programming problem, we denote performance as S and power consumption as P (in Watt). The performance of each application is expressed as its IPS (instructions per second) divided by its IPS when run on the big core in isolation (i.e., its weighted IPS), such that the sum of the performance of all applications in the workload equals system throughput (STP) [60]. $S_{L,i}$ and $P_{L,i}$ denote performance and power, respectively, for application i on the little core, whereas $S_{B,i}$ and $P_{B,i}$ denote performance and power on the big core. f_i denotes the fraction of the power period application i executes on the big core; by consequence, $1 - f_i$ then is the fraction of time it runs on the little core. P_{budget} is the available power budget. Our objective is to find f_i for each application i so that system throughput is maximized while remaining within the power budget. We only consider solutions where each application either runs on the big or the little core (no idle periods), because we find a sprint-and-rest scheme to be always suboptimal for our configuration. Initially, we assume that each application in the mix has one pair of a big and little cores allocated to it. This allows us to focus on the power budget partitioning problem based on the application and core characteristics while ignoring any further restrictions that result from applications competing over a limited number of big or little cores. We show how our approach can be extended to other configurations in Section 4.6.3 (asymmetric configuration). This optimization problem can be written as a linear programming problem as shown in Equation 4.1:

$$\begin{aligned} &\text{maximize } \sum_{i=1}^{n} f_i S_{B,i} + (1 - f_i) S_{L,i} \\ &\text{subject to } 0 \leq f_i \leq 1, \forall i \\ &\qquad \sum_{i=1}^{n} f_i P_{B,i} + (1 - f_i) P_{L,i} \leq P_{budget} \end{aligned} \tag{4.1}$$

It is clear that the set of fractions f_i that meet the constraints to form a correct solution is infinite. However, an interesting characteristic of linear programming is that an optimal solution is at one of the intersection points of the constraint equations. In the case of n applications, finding a solution could be cumbersome though because a comprehensive search to find and evaluate the intersection points is still needed. Nevertheless, we will show how we circumvent this obstacle by exploiting an important characteristic of the solution space as we describe next.

4.3.2 The Solution Space

To ease the discussion, we first consider two applications, and then generalize our findings to more applications. For two applications, the problem can be rewritten as:

$$\begin{aligned} &\text{maximize } f_1 S_{B,1} + (1-f_1)S_{L,1} + f_2 S_{B,2} + (1-f_2)S_{L,2} \\ &\text{subject to } 0 \leq f_1, f_2 \leq 1 \\ &\qquad f_1 P_{B,1} + (1-f_1)P_{L,1} + f_2 P_{B,2} + (1-f_2)P_{L,2} \\ &\qquad\qquad \leq P_{budget} \end{aligned} \tag{4.2}$$

The solution space of this optimization problem is shown in Figure 4.3 on the left. f_1 and f_2 need to be inside the square between 0 and 1, and the power budget restricts the solutions to the left of the line cutting the square. Due to the nature of linear programs, the optimal solution is one of the two intersections of the budget line and the square (indicated by the dots). This means that there are only two possibly optimal solutions: either program 1 *or* program 2 runs on the big core as long as possible, and if any budget is left over, the other program can run on the big core for a fraction of the time only.

A similar argument can be made for multiple applications in n dimensions: the optimal solution is always on one of the edges of the unit hypercube, meaning that only one fraction is a real number between 0 and 1, and all other fractions are either 0 or 1. To illustrate this, the right part of Figure 4.3 shows six possible solutions in three dimensions: all solutions have two fractions either 0 or 1, and one fraction in between 0 and 1. This implies that all applications run either on the big core or the little core all of the time, and *one (and only one!)* application migrates between big and little (because its fraction is in between 0 and 1). Finding a solution thus boils down to **finding which applications to always run on the big core (if any), which applications to always run on the little core, and finding the one application that should migrate between core types**.

4.3.3 Delta Performance / Delta Power

We have shown that using linear programming optimization, an infinite solution space can be reduced to prioritizing which applications to run on the big core at the availability of a power budget. However, searching comprehensively through all possible solutions is still not a feasible approach for a dynamic power manager.

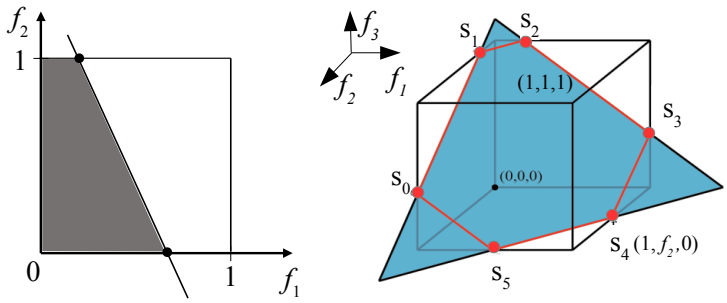


Figure 4.3: Graphical representation of the solution space for two-program (left) and three-program (right) combinations. The diagonal line/plane represents the power budget. The shaded area indicates the solution space, the dots are potential optimal solutions.

The question now is how to rank the applications such that the top-ranked applications run on the big core, and the bottom-ranked applications run on the small core; the application at the boundary then needs to switch between the big and small cores. To derive a mathematically sound ranking metric, we analytically solve the linear program. We first do the analysis for two applications, and then generalize our finding to more applications.

Using the problem defined in Equation 4.2 for two applications, we note that the optimum is achieved when the budget is completely consumed, making the second restriction an equation instead of an inequality. We solve this equation for f_2, and replace f_2 in the maximization function with that expression. This yields a linear function in f_1:

$$\begin{aligned} \text{maximize} \quad & \alpha f_1 + \beta, \\ \text{with} \quad & \alpha = \frac{S_{B,1}-S_{L,1}}{P_{B,1}-P_{L,1}} - \frac{S_{B,2}-S_{L,2}}{P_{B,2}-P_{L,2}}. \end{aligned} \tag{4.3}$$

Maximizing this function depends on the sign of α: if α is positive, f_1 should be as large as possible; if α is negative, f_1 should be as small as possible. The sign of α is determined by the *Delta Performance by Delta Power* ratio (DPe/DPo): if the difference in performance between the big and little core divided by the difference in power consumption between the big and little core for program 1 is larger than for program 2, the sign is positive, and vice versa. Hence, if the delta performance delta power ratio of program 1 is larger than for program 2, program 1 should execute on the big core as long as possible, and if it is smaller, program 2 should run on the big core.

Applying the same solution method for three programs yields the following result (with $DPDP_i$ the delta performance delta power ratio of application i

between big and little core, and β a constant term):

$$\text{maximize} \quad (DPDP_1 - DPDP_3)\, f_1 + (DPDP_2 - DPDP_3)\, f_2 + \beta \tag{4.4}$$

This means that if $DPDP_1$ is larger than $DPDP_3$, f_1 should be maximized, and similarly for f_2. If $DPDP_1$ is smaller than $DPDP_3$, then f_1 should be minimal, and similarly for f_2. If both $DPDP_1$ and $DPDP_2$ are larger than $DPDP_3$, then the largest of $DPDP_1$ and $DPDP_2$ will determine which fraction yields the largest performance benefit: if $DPDP_1$ is larger than $DPDP_2$, the term with f_1 will be larger than the term with f_2, so maximizing f_1 yields the largest performance benefit, and vice versa for f_2. In conclusion, the ideal scheduling policy is to select the program with the largest DPDP to run on the big core, and if budget is left, select the second largest DPDP, and so on. A similar analysis for four programs gives the same conclusion.

These insights provide us with the **foundation for an optimal schedule**: rank the programs based on DPe/DPo, and calculate the fraction of time the highest ranked program can run on the big core, assuming all other programs execute on the little cores. If that fraction is smaller than 1, the optimal schedule is found. If it is 1, calculate how long the program ranked second can execute on the big core, given that the first program runs on the big core all the time, and the other programs execute on the little core. Then continue this iterative process until the budget is fully consumed. This is a linear method in the number of programs, which makes it a scalable solution.

4.4 DPDP Budget Partitioning

The mathematically derived optimal power management foundations described in the previous section assume that performance and power consumption is known for all applications for both the big and little cores; moreover, it is assumed to be constant over the power period. In reality, this is not the case: performance and power is unknown (or needs to be measured or predicted across core types), and applications go through phase changes during execution. In this section we discuss the implementation details of our power manager, called DPDP, which leverages the key insights described in the previous section to optimize performance within a tight power budget in a low-overhead and scalable way. DPDP requires hardware support to independently operate (and deactivate) individual cores in the processor, in addition to the ability to measure the performance and power consumption of each core in the processor as the applications run.

DPDP power budget partitioning involves four phases: (i) profiling, (ii) a ranking and partitioning phase, (iii) a monitoring and repartitioning phase

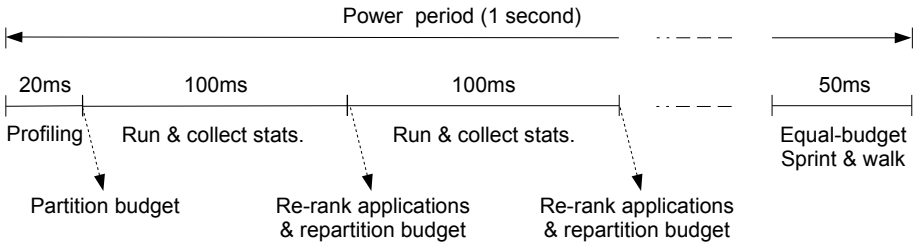


Figure 4.4: The four phases of the DPDP power manager.

to adapt to application phase changes, and (iv) sprint-and-walk to make up for profiling inaccuracies and to ensure that we do not exceed the power budget. Figure 4.4 shows how these phases are distributed along the power period.

Phase #1: Initial profiling. This phase is done only once, when the applications start. Profiling is done by executing each application for a short duration on each core type and measuring its performance and power consumption. To set the duration of the profiling phase, we need to make a compromise between profiling accuracy and overhead. A longer profiling phase has a better chance of capturing accurate power and performance measurements for each application. However, it allows applications to inefficiently consume part of the power budget, reducing the potential performance gain. We set our profiling duration to 10 ms on each core type for a total overhead of 2% for a power period of 1 second (2 times 10 ms). For applications that have no fine-grained phase behavior, this duration could be reduced without losing accuracy. We profile all co-running applications in parallel to reduce the overhead and to capture the effect of interference in shared resources. We start by running half of them on the big cores and the other half on the little cores, and migrating them after 10 ms.

Phase #2: Ranking applications and partitioning the budget. As discussed in Section 4.3, the optimal schedule requires the applications to run either on the big core or the little core, except for one application that runs partially on both core types. Using the statistics gathered for each application in the profiling phase, our scheme ranks the applications based on their respective DPe/DPo metrics, and uses this ranking to determine the schedule for each application. Algorithm 2 summarizes the classification and partitioning phase. The algorithm starts with the highest ranked application and assumes all the other applications run on the little cores. If the remaining budget permits, the scheduler allocates a big core to this application and allocates the required power budget for that core, then it updates the remaining budget statistics. The scheduler repeats the same procedure iteratively for

Algorithm 2 Determining the fraction of time on the big core each application gets during a power period.

```
Start with list of applications ranked by DPe/DPo
consumed_budget = ∑(power of all apps on little core)
while consumed_budget < available_budget do
    Take the next highest ranked application a
    if available_budget − consumed_budget ≥ P_{B,a} − P_{L,a} then
        Schedule application a on big all time
        consumed_budget = consumed_budget − P_{L,a} + P_{B,a}
    else
        Fraction_big (a) = (available_budget − consumed_budget) / (P_{B,a} − P_{L,a})
        Budget fully consumed, end while loop
    end if
end while
Schedule the rest of the applications on little core
```

the remaining applications in rank order. Once an application cannot fully execute on the big core, the scheduler calculates the fraction of time the application is permitted to run on the big core, and schedules the remaining applications on the little cores.

Phase #3: Statistics collection and budget repartitioning. To cope with changes in the application phase behavior, our scheme continuously accumulates power and performance statistics for each application based on its allocated core type. Every 100 ms, our scheme repeats Phase #2 using the updated performance and power values, in addition to the total power consumed up to this point. This enhances the accuracy of the measured statistics and ensures the adaptability of our power budget partitioning scheme to changes in workload behavior.

Phase #4: Sprint-and-walk at the end of the power period. In the last 50 ms, we determine the leftover budget. We equally divide this budget among the applications, and execute all of them on the little cores for 10 ms. We then determine how much power is 'saved' by running on the little core compared to the allocated budget. We then 'burn' this excess power by running the applications on the big cores, until it is completely burned. After that we again execute on the little core, saving budget, and then burn the saved power on the big core. This is repeated until the end of the power period. We call this *dynamic sprint-and-walk*: the fraction of time to run on the big core is dynamically determined by saving and burning the power budget. This step is required for two reasons. The first is to ensure that the execution remains within the power limit at the end of the power period. The second reason is that we can use this phase as the profiling phase for the next power period. During Phase #3, most of the applications run on a

	Big	Little
Type	Out-of-order	In-order
Frequency	2.6 GHz	1.5 GHz
Voltage	0.9 V	0.64 V
Pipeline width	4	2
ROB size	168	-
L1 I-cache	32 KB	32 KB
L1 D-cache	32 KB	32 KB
Shared L2 cache	4 MB per pair	
Memory bandwidth	25.6 GB/s	

Table 4.1: Big and little core configurations.

single core type for the whole duration. In Phase #4 on the other hand, each application runs on both the little and big core for some time, generating profile information for the next power period.

The overhead of the scheduler is minimal. The main overhead incurred by the scheduler is to rank the n applications, which has a complexity of $O(n \log n)$. Considering this overhead is incurred at most once per 100 ms (which is an adjustable design knob), the scheduler has an unnoticeable impact on performance. The scheduler described in this section can be implemented in software as part of the operating system scheduler. Optionally, it can be implemented as part of a power management unit in hardware. Similar to the software implementation, a hardware implementation needs to rank the applications based on their DPe/DPo measurements. Moreover, by continuously monitoring an application's power and performance statistics in Phase #4, as described above, profiling overhead is incurred only at the beginning of the application run.

4.5 Experimental Setup

We use the Sniper 6.0 [50] simulation infrastructure (using its most detailed cycle-level core model) to carry out the experiments in this work. We simulate heterogeneous multicore systems that consist of two core types, big and little, see Table 4.1. The big core is an aggressive four-wide out-of-order core running at 2.6 GHz, while the little core is a two-wide in-order core running at 1.5 GHz. The last-level cache is shared by all cores. There is

4 MB of LLC per pair of big and little cores. Throughout the evaluation experiments we assume that the processor relies on a cache hierarchy that does not use data or instruction prefetching techniques. We aim to study the power budget partitioning techniques based on application characteristics, regardless of prefetching benefits. Such techniques may make a memory-intensive application less memory-bound. However, their impact would still leave a significant distinction in power and performance characteristics across applications, which can be leveraged by DPDP. Therefore, we expect our results and conclusions to hold in the presence of such techniques.

We use the in-order little core configuration throughout Section 4.6. We consider a two-wide out-of-order little core in one of the sensitivity studies, to resemble recent low-power microarchitectures, such as Intel's Silvermont [61]. We evaluate scheduling 4 applications on processors consisting of 4 pairs of big and little cores. We also demonstrate the applicability of our method to architectures having fewer big cores than little cores.

We use McPAT 1.3 [51] to estimate the power consumption of our schedules, assuming a 22 nm chip technology. We report total power consumption as the sum of the leakage power and the runtime dynamic power, assuming clock gating for unused structures in the active cores. Idle cores are power-gated. We set the power budget for each big-little pair at 1 Watt for each period of 1 second, i.e., 4 pairs of big and little cores are given 4 W every second. This budget assumption is reasonable for the sake of our analysis as it falls between the big core and little core power ratings and allows sufficient room for optimization. A similar power budget has been assumed in prior work [6]. Moreover, we provide a sensitivity study to show the benefit of DPDP as we vary the assumed baseline power budget. Our simulation infrastructure accounts for the overheads associated with migrating applications between cores. This includes 20 μs required for saving and restoring architectural state [1] and for powering on the other core (because our scheduler knows when to migrate, powering on the other core could also be done slightly before the transition time). We also model the impact of cache warmup (on top of the 20 μs mentioned above). Overall, our power manager suffers minimal overhead because it switches between cores at most once every 100 ms in phase #3, and less than five times in phase #4.

To evaluate our scheme we use all 26 SPEC CPU2006 benchmarks and consider all of their reference inputs resulting in 55 benchmark-input combinations. We use PinPoint [62] to generate representative regions of 10 billion instructions, and we simulate 1 second of execution. We consider 75 randomly chosen combinations of 4 benchmarks. We evaluate performance using total system throughput (STP), which reflects the overall achieved throughput of the system compared to a reference single big core. We also

consider user-perceived performance by evaluating the average normalized turnaround time (ANTT) [60].

4.6 Results and Discussion

We now demonstrate the effectiveness of DPDP power budget partitioning. We consider the following five schemes and evaluate their effectiveness at improving performance within the power budget of 1 Watt per 1 second per application.

- *Global sprint-and-walk.* Our first scheduler considers a global power budget (i.e., 4 Watts per 1 second for four applications), and greedily optimizes performance within the given power budget. It starts by executing all applications on the little cores for 10 ms. It then calculates the 'saved' budget compared to the total budget, which it then burns by executing all applications on the big cores. The 'saved' budget equals the available budget (0.01 W per 10 ms per application) minus the amount of energy consumed during the 10 ms time interval. Once the available budget is burned, all applications migrate back to the little cores, saving budget again for the next 10 ms, which can then be burned on the big cores, etc.

- *Equal-budget sprint-and-walk.* This scheduler is similar to the previous one, except that we now partition the overall power budget across the co-running applications, and optimize the power budget for each application individually, i.e., we assign 1 Watt per 1 second for each application. Similarly to the previous scheduler, all applications start running on the little cores for 10 ms. For each application, we calculate the saved budget relative to the available budget, and we greedily run the application on the big core until the saved budget is consumed. Once an application's power budget is consumed, it migrates back to the little core for another 10 ms to again build up its power budget, and the scheme repeats.

- *Budget partitioning using performance ratio.* This scheduler is similar to DPDP as described in Section 4.4, but instead of using DPe/DPo as the ranking metric, we use performance ratio between big and little cores. In other words, applications that speed up more on the big core are given a larger share of the budget and thus higher priority to run on the big core, as long as the power budget is not exceeded.

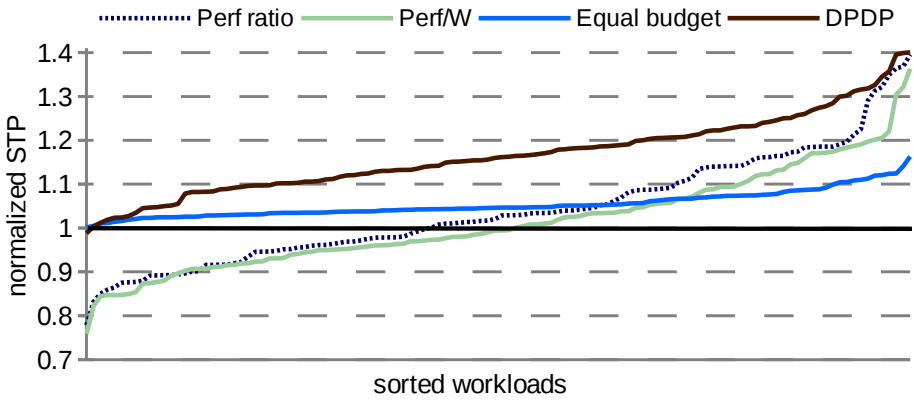


Figure 4.5: Comparing the various power budget partitioning schemes relative to global sprint-and-walk for mixes of four applications.

- *Budget partitioning using performance per Watt.* Here, we rank the applications based on the performance per Watt on the big core. Performance per Watt is a commonly used metric for expressing power efficiency, and intuitively, it makes sense to run applications with the highest performance per Watt ratio on the big cores.
- *Budget partitioning using DPe/DPo.* This is the DPDP scheduler, as described in Section 4.4.

We normalize all of the results to the global sprint-and-walk scheme, because this scheme is the natural translation of Intel's Turbo-boost [7], originally designed for DVFS, to HCMPs. The graphs in this section show how each of the schemes perform compared to the baseline scheme using an S-curve, showing the sorted relative performance difference for all workload combinations.

4.6.1 DPDP Results

Figures 4.5 quantifies the performance improvements achieved by DPDP for mixes of four applications. The graph clearly shows that DPDP outperforms the other power budget partitioning schemes. DPDP improves performance by 16% on average and up to 40% over global sprint-and-walk for mixes of four applications. The performance improvement of DPDP stems from optimal budget partitioning. DPDP selects the applications that achieve the highest raise in performance given the available budget, the period over which power is calculated, and the performance characteristics of the application on both core types. The other alternatives, as explain in Section 4.2, fail to consider one or more aspects of performance maximization under a power limit.

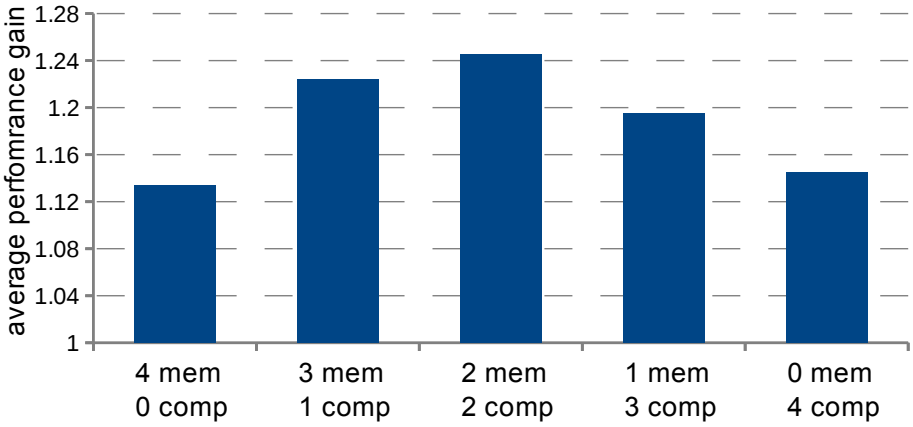


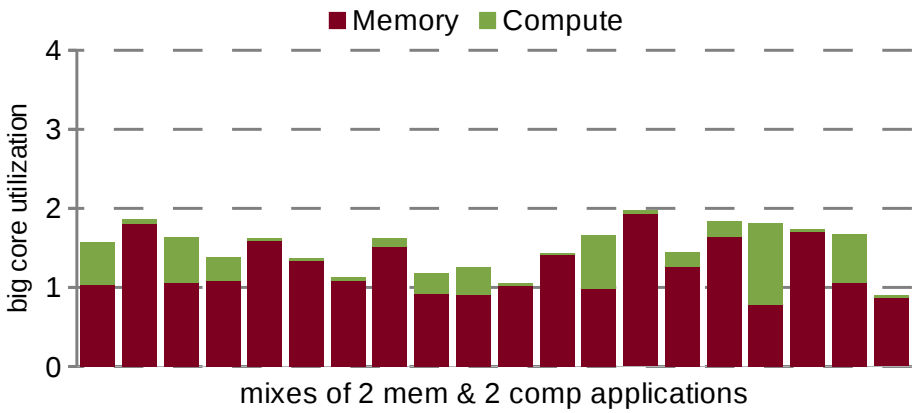
Figure 4.6: Average STP improvement for DPDP versus global sprint-and-walk for different classes of compute and memory-intensive four-application mixes.

Figure 4.5 also demonstrates DPDP's robustness: DPDP improves overall performance for all workload mixes. Although equal budget partitioning consistently improves performance, for most mixes, the improvement is limited to less than 5% on average. The other two budget partitioning schemes are less robust, and do not consistently improve performance. In fact, about half of application mixes observe a performance degradation for the schemes based on the *performance ratio* and *performance per Watt* metrics. This clearly demonstrates the effectiveness of the DPe/DPo metric for application scheduling and power budget partitioning.

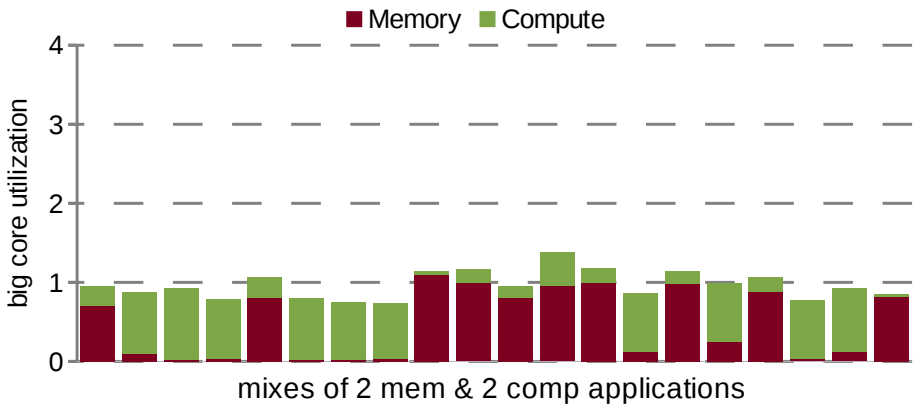
Figure 4.6 shows the average performance improvement for DPDP over global sprint-and-walk for different mixes of compute and memory-intensive applications. We classify applications as memory-intensive if they spend at least 25% of their execution time waiting for main memory. We consider workload mixes with zero to up to four memory- and compute-intensive applications. The performance gain for DPDP over global sprint-and-walk peaks for mixes with 2 compute and 2 memory-intensive applications. This is as expected: the larger the difference is between the applications' big-versus-little characteristics, the larger the impact of power budget partitioning is on performance.

4.6.2 Big Core Utilization

To gain more insight into the performance benefits achieved through DPDP, we now investigate which applications get to run on the big core more



(a) Breakdown for the DPe/DPo metric.



(b) Breakdown for the performance ratio metric.

Figure 4.7: Big core usage. For most cases, DPe/DPo favors memory-intensive applications, achieving 56% higher big core utilization than performance ratio.

frequently. Figure 4.7 breaks down the time spent on the big cores by application type (memory versus compute-intensive) for DPDP versus budget partitioning using performance ratio. For a mix of four applications, the highest utilization of the available 4 big cores equals 4. All the mixes shown in the figure use two memory and two compute-intensive applications.

Two observations can be made from the figure. First, DPDP leads to a higher big core utilization compared to budget partitioning using the performance ratio metric, compare Figure 4.7(a) versus (b). DPDP improves the big core utilization by about 56% over budget partitioning based on the performance ratio metric on average across the workload mixes. This suggests that DPDP is better able at effectively utilizing big core resources, which explains the

observed performance benefits.

Second and more interestingly, DPDP tends to favor memory-intensive applications by allocating a larger fraction of the power budget to them than to compute-intensive applications, although not uniformly so — it is a function of the DPe/DPo ratio. This observation suggests that memory-intensive applications are better at utilizing the available budget than their compute-intensive counterparts. This is counter-intuitive, as memory-intensive applications usually show a smaller performance benefit from running on a big core compared to compute-intensive applications. In fact, [33, 34, 56, 57, 59] propose scheduling compute-intensive applications on a big core to optimize performance (in the absence of a power limit). Van Craeynest et al. [32] show that memory-intensive applications could benefit from running on a big core by exploiting more memory-level parallelism, which explains the fact that the performance ratio metric also selects the memory-intensive applications for some mixes. However, we find that memory-intensive applications have another benefit under power constraints. Due to the fact that they wait more for main memory, they can more extensively leverage clock-gating, which reduces the big core's power consumption. This in its turn increases the time that they can spend on the big core, which leads to an overall increase in system throughput under a power constraint.

4.6.3 Sensitivity Analysis

We now explore the sensitivity of DPDP with respect to the available power budget, the core types available in the HCMP, and asymmetry in the HCMP configuration.

I. Available Power Budget

The available power budget has a considerable impact on the performance gain that can be achieved through power budget partitioning. Figure 4.8 shows the impact of varying the power budget on the achieved gain. Decreasing the budget to 0.75 Watt per 1 second slightly decreases the average performance gain to 13.5%. Similarly, increasing the budget to 1.5 Watt per 1 second shows smaller gains compared to the nominal 1 Watt per 1 second power budget. A much larger budget (2 Watt per 1 second), on the other hand, shows an insignificant performance gain. This is to be expected: for a power budget in-between the power ratings of the big and little cores, proper power budget partitioning is expected to provide significant performance gains. Once the budget becomes either too constrained or too abundant relative

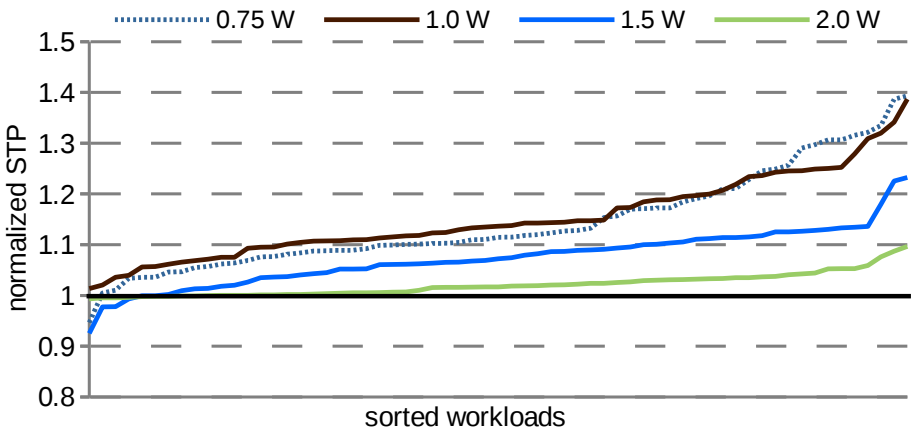


Figure 4.8: Normalized STP across different power budgets.

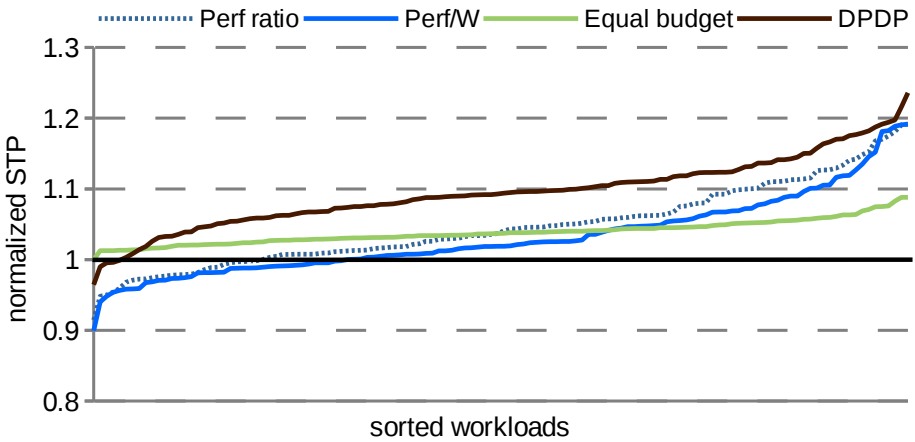


Figure 4.9: Normalized STP assuming out-of-order little cores.

to the little and big core's power consumption, budget partitioning becomes less valuable. For constrained cases, most of the applications would have to run on the little cores anyways, making it close to a conservative approach. For abundant budgets on the other hand, most of the applications are able to run on the big cores, limiting the opportunity for budget partitioning.

II. Core Type

We now set the little core to be an out-of-order core instead of an in-order core (frequency settings, cache hierarchy, and other structures remain the same), see Figure 4.9. DPDP still yields a significant performance improvement over a global sprint-and-walk approach. DPDP improves performance by

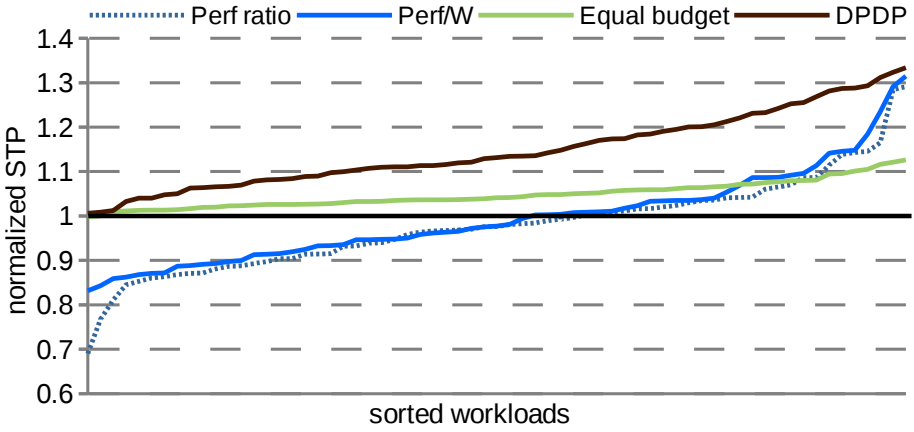


Figure 4.10: Normalized STP for the various partitioning policies assuming a CMP configuration of 2 big and 4 little cores.

9% on average over global sprint-and-walk, and up to 26%. Note that the performance gain for an out-of-order little core is lower than the gain seen for the in-order configuration. This relatively lower performance gain happens for two reasons. First, the less powerful in-order little core provides relatively lower performance compared to the out-of-order little core, increasing the opportunity for power budget partitioning. Second, the in-order little core consumes less power than the out-of-order little core, which increases the fraction of time allowed on a big core for our budget partitioning scheme.

III. Asymmetric HCMP Configuration

In the previous results, we assume as many big and little cores as there are applications. However, the DPDP scheduler also applies to configurations with fewer big cores than little cores. The only change is that the partitioning algorithm (Algorithm 2) also halts if all big cores are used before the budget runs out. Figure 4.10 shows the results for an HCMP configuration consisting of four little cores and only two big cores. The general performance improvement over all other techniques is again clear. DPDP still shows a significant 14% improvement on average over global sprint-and-walk that reaches up to 33%. However, these gains are lower than our baseline results. Clearly, as the number of available big cores decreases, more applications are forced to stay on the little cores even when sufficient power budget is available.

4.6.4 Exploiting Application Phase Behavior

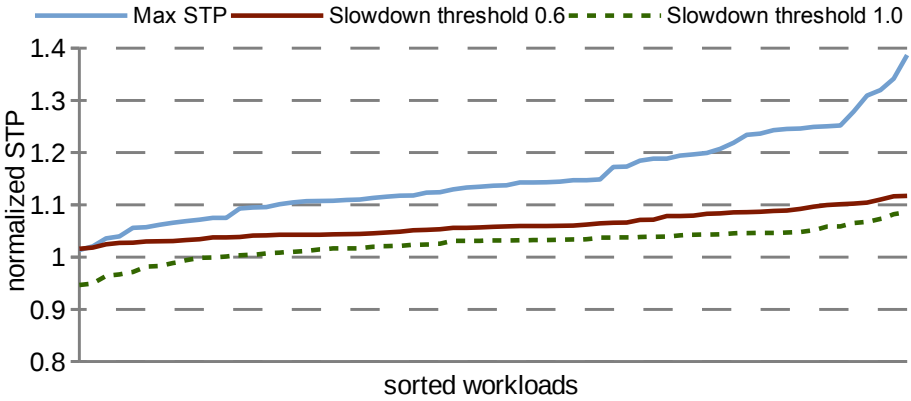
In DPDP, there is one application that runs on the big core for a fraction of the power period, and on the little core for the remaining fraction. This is done by first running on the little core, after which we switch to the big core, and the cycle keeps repeating every 100 ms throughout the application execution. This behavior is similar to the sprint-and-walk method we tested in Chapter 3. Our prior analysis shows that this method remains within a few percents of an optimal power manager even when we vary the power budget, the scheduling granularity, and the HCMP configuration. Therefore, we decided not to implement a phase-aware scheduler to further improve the performance of the single application that migrates between the big and the little core.

4.6.5 Per-Application Performance Considerations

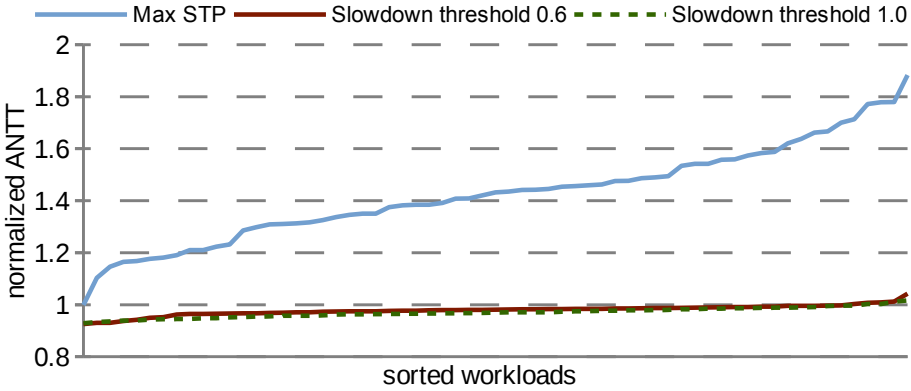
DPDP is designed to optimize system throughput under a power limit. To maximize throughput, DPDP favors applications with higher DPe/DPo values, giving them higher power budgets to run more on the big core. It is expected that applications with low DPe/DPo could suffer a slowdown compared to techniques that distribute power equally or that even greedily optimize the global power budget (e.g., global sprint-and-walk). In this section, we show how to extend DPDP with the capability to balance between the maximum throughput requirement and the maximum performance degradation any single application suffers.

To avoid slowing down low-ranked applications too much, we allocate more power to those applications once a significant per-application performance degradation is detected. Our approach tries to control the degree of similarity by which applications progress in their execution (i.e., equal-progress fairness) [38]. To assess the progress of each application, we calculate the slowdown of each application using DPDP compared to always running on the big core. The similarity of slowdowns among applications indicates the fairness of the distribution, and reveals whether one application is suffering a relatively significant slowdown.

We use the ratio of the smallest slowdown to the highest slowdown among all applications to represent the equality of progress. We call this ratio the *progress index*. The closer this ratio to one, the fairer the power distribution and the progress of applications. The closer it is to zero, the higher the focus is on system throughput (i.e., lower regard to per application slowdown). We provide the user with a knob to specify a slowdown threshold. At runtime, if the progress index drops below the slowdown threshold, DPDP focuses on



(a) System throughput for different per-application performance support thresholds



(b) ANTT for different per-application performance support thresholds

Figure 4.11: STP (higher is better) and ANTT (lower is better) for different per-application performance support thresholds. A min to max slowdown point of 0.6 improves both STP (6%) and ANTT (3%).

improving per-application performance. Otherwise, it continues to maximize system throughput.

DPDP is a flexible power manager, thus integrating this knob is straightforward. Every 100 ms, DPDP reranks the applications based on their updated performance and power consumption. To provide per-application performance support, we calculate the progress index, in addition to the DPe/DPo metric every 100 ms. If the index drops below the specified threshold, DPDP reranks the applications based on their respective slowdowns. This ensures that applications with slower progress get more time on the big core over the next 100 ms. When the progress index exceeds the slowdown threshold, DPDP resumes operating for maximum system throughput using the DPe/DPo metric.

Figure 4.11 shows the potential for adjusting the slowdown threshold to strike a sweet spot between system throughput and per-application performance. We use the average normalized turnaround time (ANTT) to assess per-application performance. Equation 4.5 describes how ANTT is calculated. For all the co-executing applications, ANTT averages the ratio of each application's performance when running in isolation (i.e., $S_{sp,i}$ in the equation) over its performance when running as part of the multi-programmed workload (i.e., $S_{mp,i}$ in the equation). ANTT incorporates equal progress by heavily penalizing slowly running applications [60].

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{S_{sp,i}}{S_{mp,i}} \tag{4.5}$$

The higher ANTT normalized to global sprint-and-walk, the worse the impact of DPDP on the perceived per-application performance. When operating DPDP in full support of per-application performance (i.e., setting the slowdown threshold to 1), system throughput sees either insignificant gain or even a slight degradation for several mixes, as the dotted line in Figure 4.11 (a) shows. On the other hand, when operating in maximum system throughput mode (i.e., original DPDP where applications are always ranked using DPe/DPo), ANTT tends to increase by an average of 40%, as Figure 4.11 (b) shows. By properly adjusting the slowdown threshold, DPDP is able to achieve similar ANTT to the mode of full per-application performance support (Figure 4.11 (b)). ANTT gets slightly reduced compared to global sprint-and-walk as can be seen for several mixes in the figure (mixes below 1). Figure 4.11 (b) also shows that proper threshold tuning improves system throughput by 6% on average and up to 12%. More importantly, the results show the flexibility of DPDP to adapt to various system requirements.

4.7 Related Work

We now discuss related work in power and thermal management, as well as recent work in scheduling for HCMPs.

4.7.1 Power and Thermal Management

Brooks et al.[63] discuss thermal constraints in microprocessors. They propose dynamic thermal management schemes for single-core processors using DVFS and fetch throttling. Donald et al. [23] study dynamic thermal management for homogeneous multicores, and several papers [26, 42, 44–46] propose schemes for maximizing the performance of homogeneous multicore

processor under strict power limits using per-core DVFS. None of these DVFS works are directly applicable to HCMPs and neither do they consider the potential gains offered by temporarily exceeding the power cap.

Intel's Turboboost 1.0 [40] increases the frequency when few cores are active, whereas Turboboost 2.0 [7] allows for increasing the frequency beyond the TDP for short periods of time to improve responsiveness. Computational sprinting [6, 41] is a technique to improve the responsiveness of interactive applications by temporarily using more cores than the TDP allows, followed by an idle cool-down period. Our technique targets improving sustained chip throughput, rather than improving interactive responsiveness. Raghavan et al. [64] show that computational sprinting can also be beneficial for sustained performance if enabling more cores leads to a better energy-efficiency. In a heterogeneous multicore setup, we find that a sprint-and-rest scheme (run on the big core, and then idle; the second technique in Figure 4.2) never outperforms a sprint-and-walk scheme (run on the big core, followed by running on the little core), because running on the little core is always more energy-efficient than running on the big core. Fan et al. [65] describe an architecture to sprint data analytics applications at a rack level. They use game theory to optimize system throughput of the whole rack given individual chip thermal limits and the rack-level power limit. An agent can sprint a chip by activating additional cores and raising their frequency. Their technique is not intended to partition the budget among multiple applications sharing the same chip. Our approach on the other hand takes a single chip running multiple applications concurrently. We improve system throughput by correctly selecting which applications to sprint on the big cores given a specific power budget.

Muthukaruppan et al. [66] and Zhu et al. [36] propose power and thermal management on HCMPs to improve energy efficiency while meeting QoS requirements. Here we focus on optimal power management with maximizing total system throughput as a main objective. Paul et al. [67] propose a technique to coordinate power and thermal management to improve performance and energy efficiency in systems consisting of both CPUs and GPUs.

4.7.2 Scheduling for Heterogeneous Multicores

Kumar et al. [29] advocate single-ISA heterogeneous multicores to reduce power consumption. They show that a heterogeneous multicore is superior to DVFS in terms of energy-efficiency. A recent study by Lukefahr et al. [47] confirms that heterogeneity indeed outperforms DVFS for low-power systems.

Many proposals advocate scheduling compute-intensive applications on the big cores, because they show the highest performance improvement [33, 34, 56, 57, 59]. Van Craeynest et al. [32] show that memory-intensive applications can also show important performance gains on big cores if they are able to exploit more memory-level parallelism. All of these proposals optimize for performance or energy-efficiency, without considering power constraints. Our analysis shows that under power constraints, memory-intensive applications have another benefit: due to their lower power consumption, they can execute longer on the big core, which increases their overall performance, despite of their lower performance improvement on the big cores.

4.8 Summary

Power and thermal constraints are becoming the main limiting factor in extracting high performance in modern processors. HCMPs provide flexibility to improve performance under power limits: if power headroom is available, applications can execute on big, powerful cores, while executing on little, energy-efficient cores cools down the chip and builds up new headroom. This paper explores mechanisms to maximize the performance of an HCMP under power limits.

We show that global greedy scheduling or equal budget partitioning schemes do not lead to optimal performance, because some applications can use the budget more efficiently than others. Previously proposed scheduling schemes for performance and energy efficiency also do not reach optimal performance, because they ignore the fraction of time that applications can make use of the big core as implied by the power budget. Using linear programming, we deduce that ranking applications by their delta performance delta power ratio (DPe/DPo) leads to the theoretically optimal schedule.

We propose and evaluate a scheduler that uses the DPe/DPo metric, and show that it indeed outperforms the other schedulers by a significant margin. Our experimental results with 4 big.LITTLE pairs demonstrate that DPDP outperforms global greedy scheduling, a natural translation of Intel's Turbo-boost to HCMPs, by 16% on average and up to 40%. An interesting observation is that under power constraints, it is beneficial to favor memory-intensive applications to run on the big core, whereas prior work advocates scheduling compute-intensive applications on the big core (in the absence of power constraints). The reason for this counterintuitive result is that memory-intensive applications usually consume less power on the big core, allowing them to run on the big core for a longer period of time, thereby improving overall performance within the power budget.

We also show that DPDP yields significant improvement as we change the available power budget and the little core microarchitecture. DPDP's performance improvement remains significant across a spectrum of power budgets. However, we notice that when the power budget gets extremely loose or extremely tight, the performance of all the scheduling metrics converge as the opportunity for optimization shrinks. We also show that DPDP yields high performance improvement as we substitute the in-order little core with an out-of-order one. However, DPDP's performance improvement over other techniques gets lower with an out-of-order core. The in-order little core consumes less power and yields lower performance than the out-of-order core. Hence, it has a bigger power and performance gap with the big core. Therefore, the impact of optimizing the power budget and mapping applications to core types has a higher influence on the fraction of time spent on the big core and the overall performance of the processor.

We demonstrate DPDP's flexibility to adapt to operating scenarios of interest. We show how to use DPDP in cases of asymmetric HCMP configurations. Similar to symmetric HCMPs, DPDP yields significant performance improvement over other techniques on asymmetric HCMPs. DPDP improves performance by increasing the big core utilization. On HCMPs with fewer big cores, DPDP's performance improvement over other techniques may get lower. DPDP can be flexibly applied to cases where the per-application perceived latency is of interest. We show how to extend DPDP with a knob that lets a user determine the level of tolerance to per-application incurred latency in favor of maximizing the system throughput. Our results show that it is possible to strike a sweet spot by which DPDP significantly improves STP over the global sprint-and-walk and at the same time slightly improving the latency per application compared to approaches that greedily compete for the power budget.

Chapter 5

Optimizing Performance on HCMPs with DVFS

5.1 Introduction

One of the main benefits brought by heterogeneous multicore processors is to expand the set of power-performance tradeoff points in CMPs beyond DVFS. However, this added flexibility significantly complicates mining for optimal operating points that achieve the power and performance targets. In this chapter, we seek to optimize the performance of a generic, power-limited HCMP configuration. We target HCMPs that feature a generic number of core types, and each core type can run on several Voltage-Frequency (V-F) operating points.

We maintain our view of a power limit as a consumption rate that can be exceeded instantaneously as long as it is preserved over a time period. To maximize performance of one application on one core type, a traditional power manager uses the default V-F operating points to generate a performance-power curve for the application. It then walks the curve to find the point of maximum performance that does not exceed the limit [68]. A power target in-between two points can be achieved by alternating between the points. We show in this work that the same approach should not be applied to power-limited HCMPs. This approach leads to sub-optimal power-performance tradeoff points.

We observe that when mixing different core types in HCMPs, naively relying on the default set of operating points leads to sub-optimal results. Finding the highest performing point below the power limit, and alternating with another point (e.g., immediately) above the limit can waste significant power

and performance. We show that, contrary to intuition, even when filtering the default set of operating points to keep only Pareto-optimal points, this approach still leads to significant performance degradation. Using a brute-force approach to find the points of maximum performance involves high overhead. This is especially true with a high number of core types, operating points per core, and concurrent applications. In particular, optimizing performance for multiple applications sharing a power budget explodes the search space as it requires both optimal budget partitioning among applications and optimal per-application operating point selection based on the assigned budget.

In this work we show that several points in the default set of operating points drain power without significant performance benefit when they are used in power-limited HCMPs; hence, the name *power holes*. Surprisingly, Pareto-optimal points suffer from a similar problem. Sifting power holes is key to optimizing performance for both cases of a single application and multiple concurrent applications. Moreover, sifting must be performed at runtime as it is application-specific. We propose PH-Sifter, a fast and scalable technique for sifting operating points, and keeping only the set of points that optimally use power to gain performance. We show significant performance improvements for PH-Sifter compared to Pareto-sifting for three use cases: (i) maximizing performance for a single application, (ii) maximizing system throughput for multi-programmed workloads, and (iii) maximizing performance of a system in which part of the power budget is reserved for a high-priority application. Our results show performance improvements of 13%, 27%, and 28% on average that reach up to 52%, 91%, and 2.3x, respectively, for the three use cases.

5.2 Background and Motivation

This work targets the performance of power-limited HCMPs with multiple V-F operating points per core type. Because the V-F operating points come in discrete values, power managers need to search for the highest performing point that does not violate the power limit [68]. Conservatively selecting a single operating point does not guarantee maximum performance under a power limit. First, the difference between that point's power rating and the allowed budget is wasted, degrading performance. More importantly, higher performance could be achieved within the same limit when migrating the application between two points; one point that does not exceed the limit and another point that exceeds it but yields a higher performance. One approach to find the optimal operating points is to perform a brute force search for all

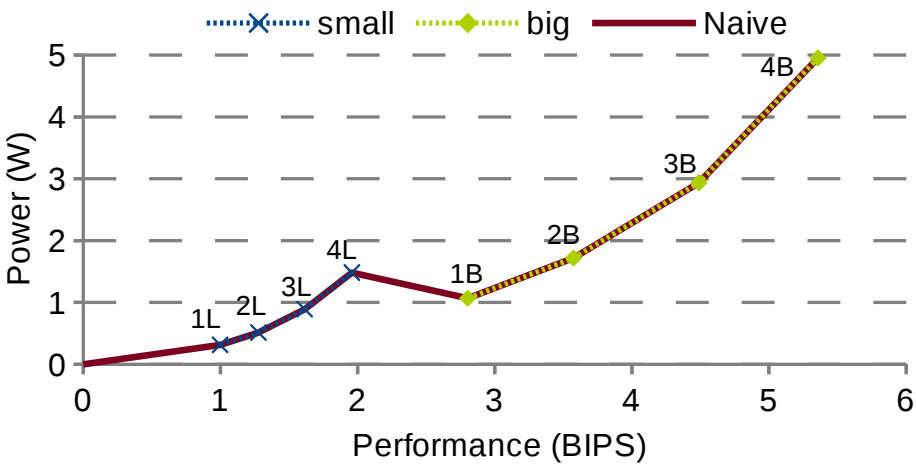


Figure 5.1: Naive performance-power curve walking.

the pairs of operating points. However, this method suffers from significant overhead for a single application, and the overhead explodes as the number of operating points and concurrent applications increases.

Figure 5.1 shows the performance versus power consumption for two core types, each with four operating points, for an example benchmark, bzip2.liberty from SPEC CPU2006. Each point represents one V-F setting of the core (see Section 5.5 for details regarding the experimental setup). Figure 5.1 also shows the naive approach to maximize performance under a power limit. It starts by walking the default operating points from the lowest performance point on the little core to the highest point that does not exceed the power limit. Then it opportunistically alternates between this point and the next higher performance point to leverage the whole power budget, while keeping the average power below the limit.

For example, assuming a power limit of 1 W, this naive approach schedules the application on the third point of the little core (3L) and switches to the fourth point (4L) as much as the budget allows. This approach results in sub-optimal performance because alternating between point 3L of the little core and 1B of the big core better trades power for performance.

A method that considers only Pareto-optimal performance-power points filters point 4L, as Figure 5.2 shows. An operating point is considered Pareto-optimal if there exists no other operating point that yields better performance at lower power. Point 4L of the little core is a non-Pareto-optimal point, and can therefore be discarded. All other operating points are Pareto-optimal. Pareto curves are usually used to show the spectrum of efficient operating points in HCMP products [16]. Surprisingly, walking the Pareto-optimal

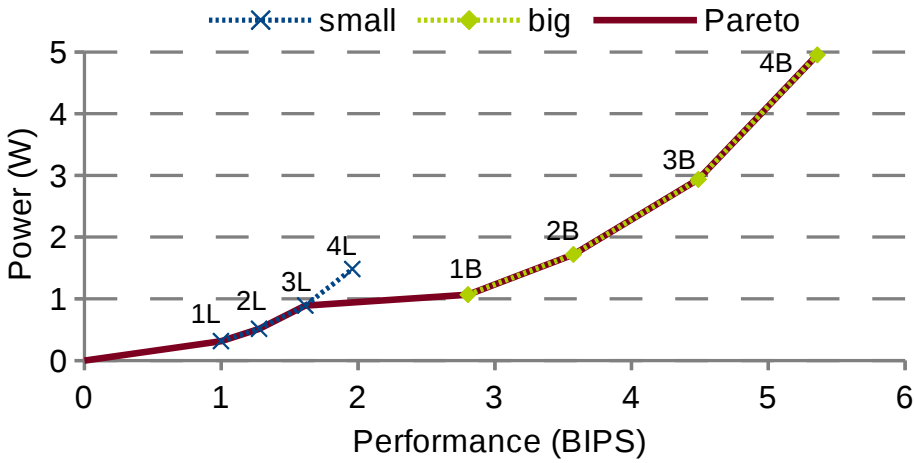


Figure 5.2: Pareto-sifting.

curve to find the optimal operating points within the power budget [69] still leads to power holes that waste power and degrade performance.

Consider the same application and the same 1 W power budget. An approach that uses Pareto-optimal points alternates between points 3L (little) and point 1B (big). Assuming no migration overhead in this example, the application runs 62% of the time on the big core and 38% on the little core, reaching an average performance of 2.35 BIPS. The Pareto approach improves over the naive approach because the line between points 3L and 1B crosses the 1 W budget line further to the right on the x-axis. However, there are still opportunities to improve performance. For example, point 2L uses less power than 3L and allows the application to utilize the big core at point (1B) up to 88% of the time, for a higher performance of 2.62 BIPS. As we show in Section 3.3.2, this latter choice is still not the optimal. This calls for a feasible approach to identify optimal operating points that maximize performance under power limits.

5.3 PH-Sifter

Based on the discussion in Section 5.2, the problem of filtering operating points boils down to selecting the next point that yields the highest performance for power, starting from a lower operating point. In this section, we propose PH-Sifter, a fast operating point sifting technique. PH-Sifter relies on Delta Performance / Delta Power (DPe/DPo) to rank the relative efficiency of optimal operating points. DPe/DPo is the reciprocal of the slope between

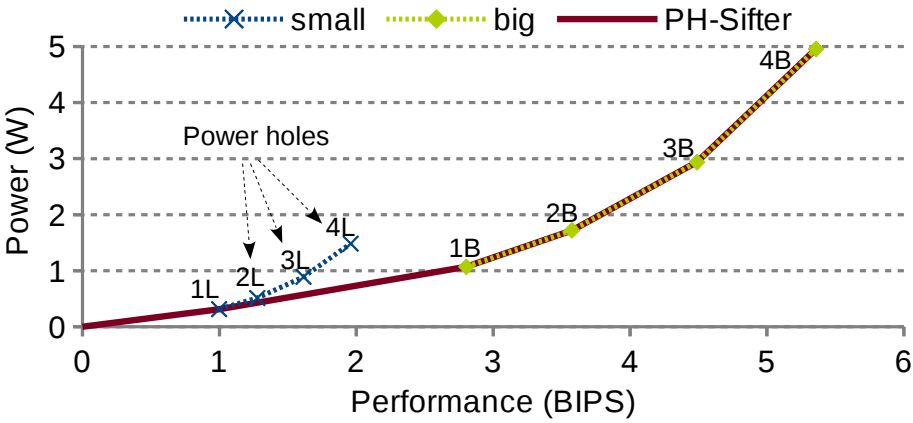


Figure 5.3: PH-Sifter.

two operating points. Hence, the flatter the slope, the higher DPe/DPo. In Figure 5.1, the slope between the third and fourth points of the little core (3L vs 4L) is steeper than the slope between the third point of the little core and the first point of the big core (3L vs 1B).

Now consider the line between the lowest operating point on the little core and the lowest point on the big core as in Figure 5.3. Its slope is lower (i.e., its DPe/DPo is higher) than all other lines originating from the lowest operating point of the little core to any other operating point. Intuitively, alternating an application between the lowest point of the little core (1L) and the lowest point of the big core (1B) results in a (virtual) operating point on this line. It is clear that any other operating point on the little core results in a worse power/performance (virtual) operating point. As a result, all operating points of the little core can be pruned except for the lowest point.

A set of operating points can be proven optimal for performance optimization under a power limit *if (and only if)* DPe/DPo is a monotonically decreasing function from the lowest to the highest power-performance operating point. Suppose a set of operating points are not monotonically decreasing. This means that there is at least one operating point 'x' that breaks monotonicity and its performance rating falls between the performance of two operating points (a and b), such that 'a' is followed by 'x' which is then followed by 'b' when walking the operating point power performance curve. Since this point breaks the decreasing monotonicity, DPe/DPo(a,x) is smaller than DPe/DPo(x,b). Moreover, DPe/DPo(a,b) is greater than DPe/DPo(a,x) but smaller than DPe/DPo(x,b). If the latter condition is not satisfied, this means that 'x' forms a monotonically decreasing curve or it is just an impossible point to exist, and both are cases of no interest. Because DPe/DPo(a,b) is

greater than DPe/DPo(a,x), alternating between points 'a' and 'b' always yields a better performance for any power budget ranging between the power ratings of points 'a' and 'b'. Therefore, the set of operating points cannot be optimal because point 'x' is not needed.

Figure 5.3 visualizes this. The Pareto frontier in Figure 5.2 is not a monotonically decreasing function: the slope between the second and third points of the little core (2L vs 3L) is steeper than between the second point of the little core and the first point of the big core (2L vs 1B). In turn, the slope between (2L vs 1B) is steeper than that between the first point on the little core and the first point on the big core (1L vs 1B), making the second, third and fourth points on the little core (2L, 3L and 4L) sub-optimal and classified as power holes as shown in the monotonically decreasing curve in Figure 5.3.

Put differently, the monotonicity of the DPe/DPo metric means that the curve of optimal operating points should be convex, i.e., no point should be above any line connecting two other points. For the power manager, this means that the **next operating point to be considered is the one with the highest DPe/DPo relative to the current operating point, and all intermediate operating points are *power holes* that should be pruned.**

5.3.1 PH-Sifter Algorithm

Algorithm 3 describes PH-Sifter. The algorithm starts at the lowest operating point on the little core. It then chooses the next optimal point as the one with the highest DPe/DPo value. All the intermediate points between the two selected points are considered power holes and are filtered out. From the newly chosen point, the algorithm again selects the point with the highest DPe/DPo value from the set of operating points that were not filtered out already. The algorithm proceeds until it reaches the highest operating point on the big core. The algorithm has a complexity of $O(np)$ to prune p operating points for n applications. The filtering algorithm can be also parallelized to prune the n applications simultaneously.

Excluding intermediate points at each step of Algorithm 3 results in the convex shape that guarantees optimality of the sifted set. For any potential budget, the two points in the set with power values just above and below the budget form a line that crosses the budget line furthest to the right (i.e., highest in performance).

Consider Figure 5.3 as an example. Starting from point zero, point 1L has the highest DPe/DPo. Index $current\_point$ is set to point 1L and it is included in the optimal set. Next, from 1L, point 1B has the highest DPe/DPo, so it is included in the optimal set, $current\_point$ is set to 1B, and intermediate points

Algorithm 3 PH-Sifter: excluding power hole operating points.

```
init_points = all operating points form all core types
sifted_points = NULL
Sort (ascending) init_points by performance values
current_point = init_points[0]
Push init_points[0] to sifted_points, pop it from init_points
while current_point is not last element in init_points do
    Calculate DPe/DPo from current_point to all init_points
    Take highest DPe/DPo as highest_point
    Remove all intermediate points between current_point and highest_point
    from init_points
    Push highest_point to sifted_points, pop it from init_points
    current_point = highest_point
end while
output sifted_points
```

(2L, 3L, 4L) are filtered out. Repeat this iterative process until $current\_point$ reaches 4B.

Note that Algorithm 3 is not restricted to two core types (i.e., big and little types) only. PH-Sifter operates under the assumption that the application can be scheduled on any number of available core types, each featuring any number of operating points. The initial set of points, i.e., $init\_points$ in Algorithm 3, considers all the operating points on all core types. The algorithm takes all the available operating points and sifts them based on the DPe/DPo metric. The core type corresponding to each operating point is not needed by the algorithm to perform the sifting process. Therefore, the results and use cases we show in this chapter can be easily generalized to any number of core types and operating points.

5.3.2 Multiple Concurrent Applications

As the number of available cores and concurrent applications increases, operating point sifting becomes more important. Algorithm 4 shows how to maximize the performance for multiple concurrent applications. In principle, the algorithm walks the power-performance curves for all the applications at the same time starting with the lowest operating point on the little core. At each step, the algorithm selects the application with the highest DPe/DPo value and provides it with the budget necessary to move to the next higher operating point. The algorithm updates the new operating point of that application and repeats the procedure until the power budget is distributed among the applications.

Algorithm 4 Maximizing throughput for multiple applications.

Sift each application (PH-Sifter) to get $sifted\_points$ per app.
$current\_point[App_i] = sifted\_points[App_i][0]$
$next\_point[App_i] = sifted\_points[App_i][1]$
$used\_budget$ = $\sum$(power of all apps on $current\_point$)
while $used\_budget < available\_budget$ **do**
  Calculate DPe/DPo per app between $current\_point$ & $next\_point$
  Sort the results & take the highest ranked application a
  if $available\_budget - used\_budget \geq next\_point[a].power - current\_point[a].power$ **then**
    Schedule application a on next_point[a]
    $used\_budget = used\_budget + next\_point[a].power - current\_point[a].power$
    $current\_point[a] = next\_point[a]$
    $next\_point[a] = next\_point[a] + 1$
  else
    $fraction_{next\_point}(a) = \frac{available\_budget - used\_budget}{next\_point[a].power - current\_point[a].power}$
    Budget fully consumed, end while loop
  end if
end while
Schedule the last application on $next\_point$ for $fraction_{next\_point}$ and on $current\_point$ for the rest of the time
Schedule the rest on their $current\_point$

As system throughput sums the performance gain achieved by each application, each step in Algorithm 4 allocates power to the application with the highest potential gain, and consequently the highest contribution to system throughput. Allocating part of the budget to another application at any step lowers throughput. The optimality of both algorithms can be mathematically proven.

Note that Algorithm 4 would fail to maximize performance without properly-sifted operating points. **Without PH-Sifter, a power manager may have to resort to an expensive non-scalable brute-force approach to optimally divide the power budget and guarantee maximum performance for this case.** For n applications, Algorithm 4 has a complexity of $O(n)$ because it needs to find the application with the highest DPe/DPo at each step. The total number of steps depend on the power budget.

5.4 Power Management Scheme

So far we have described the algorithms for sifting power holes of each application using PH-Sifter, and power budget partitioning through a step by step walking of the curve and allocating power to the applications with the highest DPe/DPo. In this section we briefly describe the power manager we

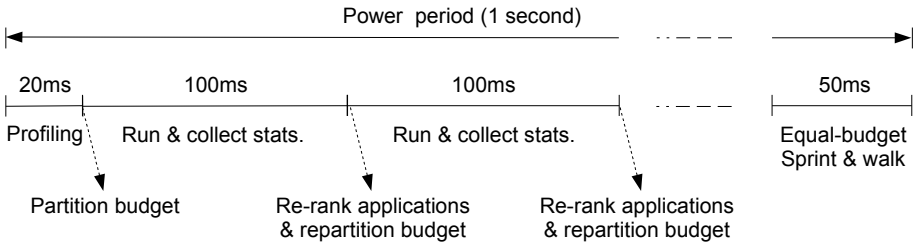


Figure 5.4: The four phases of the power manager.

implemented for evaluating the impact of these algorithms. We use a similar framework to the one used in Section 4.4. We use the same framework to compare both PH-Sifter and Pareto-sifting.

The power manager has four phases distributed over the power period of 1 second as shown in Figure 5.4. In the first phase, the manager profiles all the applications on the middle V-F operating point of each core type for 10 ms. The manager assigns half of the applications to one core type, and the other half to the other core type in the first 10 ms. The manager reverses the profiling assignment in the following 10 ms. In the second phase, the power manager uses the profiled middle operating point for each application on each core type, and estimates the power and performance of the remaining operating points using the models taken from [70].

In principle, the model divides the execution time into two parts, pipelined and non-pipelined. The pipelined part represents the fraction of the execution that is spent on the processor and that scales (i.e., expands or shrinks) with the change in operating frequency. The non-pipelined part represents the time the processor spends waiting for a memory access. The frequency domain of the memory system is independent from the processor. Therefore, the non-pipelined time does not scale with the processor operating frequency. Equation 5.1 shows how performance is scaled based on this classification of time. $T(V_n, f_n)$ is the total execution time at a nominal voltage and frequency setting, and $T(V, f)$ is the execution time we would like to predict when running at another voltage-frequency operating point (V, f).

$$\begin{aligned} \mathrm{T}(V_n, f_n) &= T_{pipelined}(V_n, f_n) + T_{non-pipelined} \\ \mathrm{T}(V, f) &= \frac{T_{pipelined}(V_n, f_n)}{f/f_n} + T_{non-pipelined} \end{aligned} \tag{5.1}$$

We rely on the leading-loads model [25] to estimate the pipelined and non-pipelined times. As equation 5.1 shows, we scale the first term linearly with frequency while keeping the second term unchanged. In practice, we

track the non-pipelined time and consider the rest of the execution time as pipelined time. The leading-loads model shows that when estimating the non-pipelined time that results from a load that misses in the last level cache, we only need to track the memory access time of the first missing load until it returns from memory. This applies whether the access happens in isolation or with multiple overlapping memory accesses. Similarly, we also measure the non-pipelined time that results from instruction cache misses. We model both cases in our power management technique.

Once performance numbers are estimated at the target frequency, we use the estimated time to estimate energy and power at the target voltage and frequency. We assume a clock-gated processor throughout this thesis. Equation 5.2 describes how to calculate the total energy of the processor as we scale the operating voltage and frequency settings.

$$\mathsf{E}(V, f) = E_d(V_n, f_n).\frac{V^2}{V_n^2} + P_s(V)T(V, f) \tag{5.2}$$

The first term in the equation shows that to scale the dynamic energy that is measured at a nominal frequency and voltage setting to a target voltage setting, multiply it by $\frac{V^2}{V_n^2}$. The second term simply scales the static power by the total execution time assuming the new operating frequency. Further details on the power and performance estimation models can be found in [70].

After predicting the power and performance numbers at all the operating points, the power manager sifts the points per application and partitions the power budget. We compare between PH-Sifter and Pareto-sifting. For a single-threaded application, the method to walk the resulting curve is identical. For power budget partitioning among multi-programmed workloads, we use the same incremental DPe/DPo method once assuming a Pareto-optimal curve and another time using the PH-Sifter curve.

In phase three, the manager has already partitioned the power budget and set the schedule and operating point selection. The application runs on the assigned setting. To adapt to changes in program behavior, the manager keep accumulating power and performance statistics at runtime during this phase. Every 100 ms it regenerates the curves per application, sifts the newly generated curves and repartitions the power budget. Phase continues until the final 50 ms of the execution. The fourth phase starts at the final 50 ms of the power period. In this phase, the applications go through equal budget sprint-and-walk, between the highest operating point on the big core to the lowest performance point on the little core, to make sure the power budget is respected at the end of the power period.

The power management scheme can be implemented either in software as part of the operating system scheduler or in hardware as part of a power

Table 5.1: Voltage-frequency settings used in the experiments.

Frequency (GHz)	Voltage (V)
1.5	0.64
2.0	0.74
2.6	0.88
3.2	1.1

management unit. In hardware, we require n comparators to prune n applications in parallel. The same comparators can then be reused to guide the walking of the curves as described by Algorithm 4 in at most $O(\log n)$ comparisons each time. The total number of repetition is a function of the power budget. Whether implemented in hardware of in software, the algorithm scales linearly with the number of available operating points per application since the pruning step takes at most $O(np)$ steps as previously discussed ($O(p)$ if done in parallel). Moreover, considering this overhead is incurred at most once per 100 ms, the overhead of our power management technique is expected to have an unnoticeable impact on performance.

5.5 Experimental Setup

This work is a direct progress from the previous chapter. We therefore maintain a similar simulation methodology to what has been described before. We use Sniper 6.0 [50] to perform the necessary simulation experiments for this work. We simulate HCMPs consisting of two core types, a big and a little core. Similar to the assumptions of the previous chapter, the big core is aggressive four-wide out-of-order, while the little core is two-wide in-order. We assume an LLC of 4 MB per pair of big and little cores that is shared by all the cores. We consider the voltage-frequency operating points shown in Table 5.1. Extra intermediate frequency settings are generated by interpolating between the shown operating points. Similar to previous chapters, we carry out our experiments while assuming a cache hierarchy that does not use data or instruction prefetching techniques. We aim to study the power budget partitioning techniques based on application characteristics, regardless of prefetching benefits.

We use all SPEC CPU2006 benchmarks using all the reference inputs. For the experiments involving multi-programmed workloads, we randomly generate 75 mixes of four applications and run each mix on a processor with four

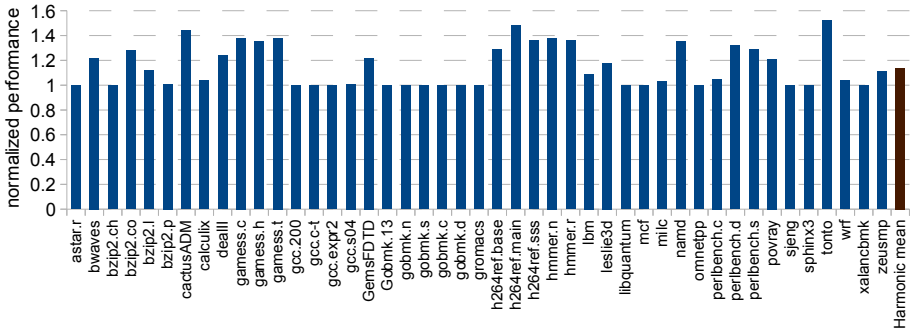


Figure 5.5: PH-Sifter performance gain over Pareto-sifting for a single application with four points per core type.

big/little pairs. We use McPAT 1.3 [51] to estimate total power consumption of the processor, including both dynamic and leakage power, assuming 22 nm chip technology. Similar to previous chapters, we assume idle cores are power-gated and unused structures in the active cores are clock-gated. We assume 20 μs [1] core migration overhead, and account for all related overheads, such as cache warmup.

5.6 Evaluation

Operating point sifting is necessary for any optimization targeting the performance of power-limited HCMPs with multiple V-F operating points per core. We evaluate PH-Sifter under three scenarios. The first case shows the performance improvement of a single application using PH-Sifter. The second case extends the first to multiple concurrent applications. Finally, we look into a case with one high-priority application that requires a fixed fraction of the power budget, leaving the remaining budget to other lower-priority applications.

5.6.1 Maximizing Performance for a Single Task

Figure 5.5 shows the performance gain of PH-Sifter normalized to Pareto-sifting for the SPEC CPU 2006 applications. Applications run on a pair of big/little cores each with four operating points, assuming a power budget of 1 W. The results show an average performance improvement of 13%, up to a maximum of 52%. The average is affected by applications that do not benefit from pruning under the 1 W budget assumption. For these applications, the budget falls between optimal points that exist in both PH-Sifter and

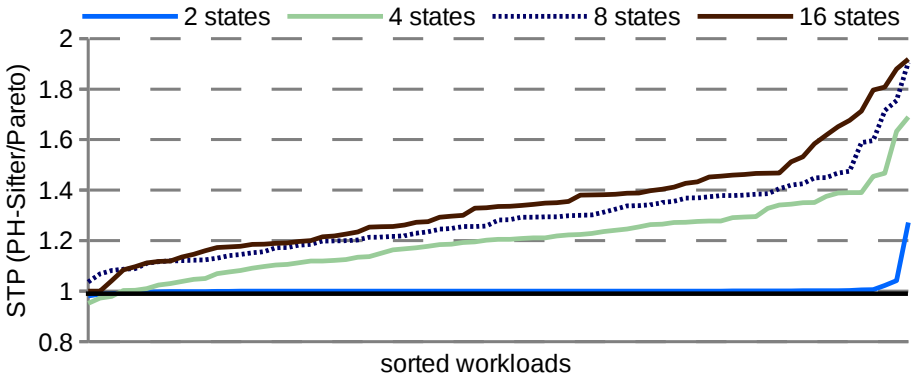


Figure 5.6: Comparing PH-Sifter vs Pareto-sifting for different numbers of V-F operating points per core type, using multi-programmed workloads.

Pareto-sifting. If the budget for these applications changes, due to sharing the budget with other applications or due to a fluctuation in the power consumption of other components (e.g., GPUs), PH-Sifter will similarly show high performance benefits.

5.6.2 Maximizing Performance for Concurrent Tasks

We use Algorithm 4 to maximize the performance for multi-programmed workloads. Figure 5.6 shows the performance gain of PH-Sifter over Pareto-sifting with 2, 4, 8 and 16 operating points per core type. Each point on the horizontal axis represents a mix of four applications, for a total of 75 mixes. The workloads are sorted based on their normalized system throughput (STP, on the vertical axis) [60].

The figure shows that PH-Sifter outperforms Pareto-sifting for different numbers of operating points. This shows the impact of power holes on Pareto-sifting performance. Additionally, we note higher improvement as the number of operating points increases. The intuition is that as the number of operating points increases, more power holes need to be sifted. We notice that Pareto-sifting fails to identify a larger number of non-optimal points causing the gap with PH-Sifter to widen. The lowest gains are for two points per core, with only a few workloads showing noticeable gains. For these experiments, we use only the lowest and highest points per core type. We find that Pareto-sifting in most cases correctly identifies the high point on the little core as a power hole, similar to PH-Sifter (Figure 5.3, considering only the lowest and highest points).

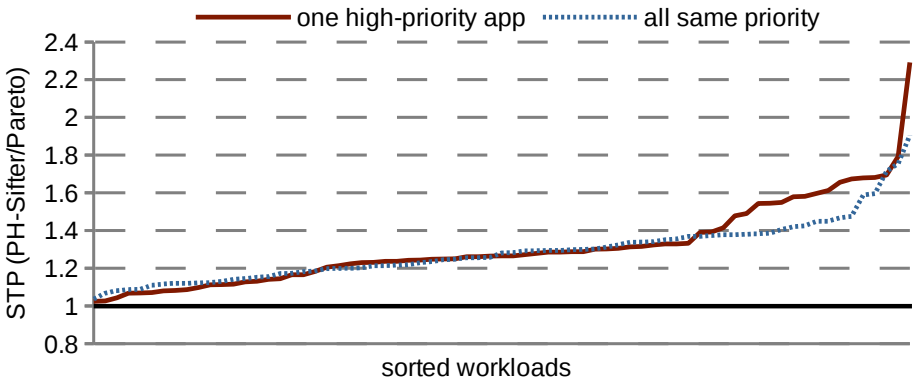


Figure 5.7: Performance gain of PH-Sifter over Pareto-sifting while provisioning for a high-priority application.

Our results show that proper sifting of operating points significantly impacts performance. An average performance gain of 19%, 27%, and 32% is achieved over Pareto-sifting, that reaches up to 69%, 91%, and 92% for 4, 8, and 16 points per core, respectively.

5.6.3 Provisioning for QoS

We consider a scenario where a fixed fraction of the power budget is reserved solely for a high-priority application to meet its performance target, while the remaining budget is divided among the remaining lower-priority applications. This scenario could be further generalized to one where the high-priority application consumes a variable fraction of the power budget, while the remaining budget goes to the other applications.

Our experiment assumes 25% of the power budget is allocated to the high-priority application. Figure 5.7 shows the results of PH-Sifter compared to Pareto-sifting assuming eight operating points per core type. The figure shows again the result of the case without a high-priority application, to show the similarity. Although the same power budget is allocated to the latency-sensitive application, PH-Sifter significantly improves system performance by an average of 28% and up to 2.3x compared to Pareto-sifting.

This case could be viewed as a mix of the two previously shown use cases. The application risks not meeting its quality target using only 25% of the power budget if the operating points are not properly sifted. Similarly, the maximum system performance of the remaining applications would suffer significantly if the power holes are used per application.

5.7 Discussion

The main goal of this thesis is to propose power management techniques that maximize performance for single-threaded applications and multi-programmed workloads under power constraints. In this section, we provide a preliminary discussion of how the proposed techniques can be further extended to multi-threaded applications and to other cases of interest where power-efficiency is enforced when applications have specific performance requirements.

5.7.1 Multi-threaded Applications

The characteristics of multi-threaded applications distinguishes them from single-threaded applications and workloads consisting of a group of single-threaded applications. In particular, the performance of multi-threaded applications is not necessarily the sum or average of its individual threads. Moreover, adjusting the voltage-frequency operating points does not change performance and power consumption of multi-threaded applications in the same manner as they would for single-threaded applications. Synchronization and communication among threads impact the overall performance of multi-threaded applications. Therefore, power and performance characteristics of individual threads cannot be representative of overall application performance.

Prior work [27] reveals two categories of multi-threaded applications. In the first category, all the threads are equally critical. Equally progressing all threads is necessary for overall application performance. The second category consists of one or more threads that are more critical than others. Improving performance of critical threads significantly impacts application performance, while improving a non-critical thread performance has negligible impact on total performance. Other related works [28, 70] seek to predict the energy and performance impact of changing the voltage and frequency settings on multi-threaded applications. We use wisdom from prior work to layout a strategy for extending our work to maximize performance of multi-threaded applications under power constraints.

For multi-threaded applications whose threads are equally critical, allocating a power budget to improve the performance of any thread is expected to provide similar performance impact regardless of the selected thread. However, boosting the performance of the same thread all the time may reduce its criticality. For example, the thread may finish its tasks or reach a barrier ahead of other threads. Therefore, we expect that the best strategy

for this category of multi-threaded application is to ensure equal progress of all threads. This requires allocating the power budget equally across all threads. When extra power can be allocated to one or few threads only, the scheduler needs to ensure that this power budget is allocated to a different set of threads at each scheduling interval, e.g., in a round-robin fashion.

When one or more threads in the application are more critical than others, prior work suggests that investing a fraction of the power budget to improve the performance of a non-critical application would yield negligible performance improvement for the total application performance. On the other hand, improving the most critical thread leads to the highest performance improvement. Therefore, we propose evaluating the criticality of all threads of the application, e.g., building a criticality stack [27], and use criticality information for power budget partitioning.

In cases when the big and little cores have only one voltage-frequency operating point, the power manager allocates enough power for the most critical thread to run on the big core. Any remaining power is then allocated to the second most critical thread, and so on. The rationale behind such a scheme is that the highest little-to-big delta performance can be achieved by boosting the performance of the most critical thread. A non-critical thread has a delta performance to delta power ratio of approximately zero because improving its performance does not impact total performance. Therefore, we propose using per-thread criticality information as a proxy for the delta performance by delta power metric.

For processor featuring multiple core types, each with multiple voltage-frequency operating points, we propose following the same steps used for a single-threaded application when dealing with multi-threaded applications. First, the voltage-frequency operating points for each thread are filtered out based on their individual power-performance characteristics (i.e., not considering the overall performance of the application). This leaves the power manager with the optimal set of points for each thread. The second step that walks the curves by taking the application with the highest DPe/DPo step each round cannot be used in the same manner as for single-threaded applications. The problem here, again, is that allocating the power budget to the thread with the highest DPe/DPo step may not necessarily improve total application performance because that thread may be non-critical. Instead, we propose advancing the most critical threads, based on the criticality stack of the application.

When walking the curves of each thread, the power manager has to determine how much of the power budget to allocate to the most critical thread, relative to the second most critical and other lower-ranked threads. This

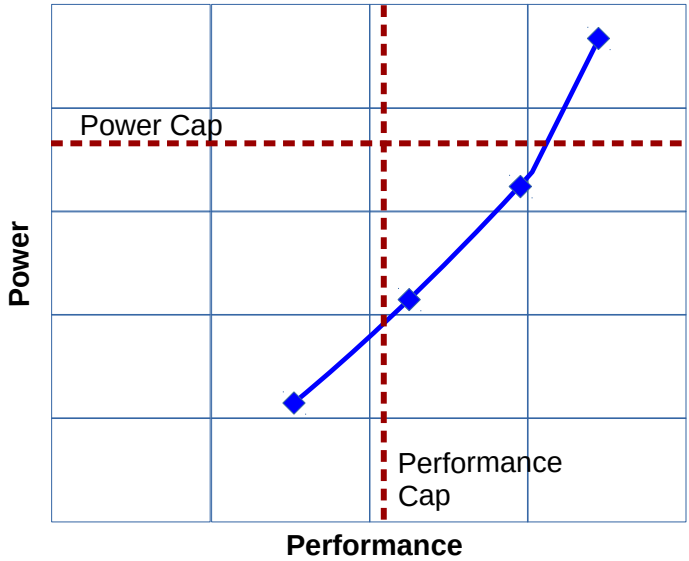


Figure 5.8: Optimizing under a performance cap vs. optimizing under a power cap.

decision basically determines how many steps each thread can walk higher on its power-performance curve of filtered operating points. We propose a heuristic to advance threads by a number of steps that is proportional to the relative criticality of threads. For example, if the criticality stack suggests that the most critical thread is three times more critical than the second ranked thread, it is allocated a proportionally higher fraction of the power budget. A more elaborate scheme is beyond the scope of this thesis and will be considered for future work.

5.7.2 Performance Capping

Our optimization techniques maximize performance within a given power constraint. Several mobile applications may demand a different but closely-related optimization. In particular, the computation capabilities available to several mobile applications are surplus to their needs. We thus see an opportunity to make another optimization: Given the performance demand of an application, how to minimize its power consumption.

This optimization problem can be viewed as the reciprocal of the optimization problem addressed in this thesis. Figure 5.8 shows the relationship between the two optimization targets. A horizontal line cutting through the performance-power curve representing the processor capabilities represents

the power constraints, under which we optimize performance. A vertical line represents a specific performance requirement beyond which the application does not benefit. Here we briefly discuss how our proposed techniques can be leveraged to minimize power consumption.

For the case of one single-threaded application and one operating point per core type, sprint-and-walk can be applied with the performance goal in mind. This means that the application first starts on the big core to gain a performance headroom. Then the application migrates to the little core until the performance can no longer tolerate running on the little core without violating the performance requirement. Then it migrates again to the big core to gain another performance headroom for a specific time interval.

The techniques proposed in this thesis are necessary for the new optimization target in the general case where multi-programmed workloads run on multiple core types each with multiple voltage-frequency operating points. In particular, to minimize power consumption of the processor given a performance cap, generating a per-application curve similar to the one shown in Figure 5.8 is necessary. Obtaining such a curve relies on pruning power holes using the same PH-Sifter techniques described by Algorithm 3. This allows each application to reach to the two optimal performance points between which it needs to migrate for minimal power given a performance cap. To minimize the power consumption given a cap on system performance, we take all the pruned curves for each application and walk them by reversing the walk described by Algorithm 4. The inverse walk means that we start all the applications on the highest operating point. At each step we walk the application with the lowest DPe/DPo value (i.e., the biggest slope). Taking the application with lowest DPe/DPo step means that our approach selects the step that drops power the most without impacting performance significantly. We repeat these steps until we reach the point where dropping the performance of any of the applications further leads to potential violation of the performance requirement. At this point, one application can still be migrated among two performance points as much as the performance requirement allows. Taking the highest steps of power reduction at each step guarantees that we minimize the power consumption while operating at the desired level of system performance.

5.7.3 Bursty Applications

The techniques proposed in this thesis are mostly oriented towards improving system throughput (or sustained performance for single-threaded applications) given a restricted power budget. However, HCMPs may operate in environments where applications have a bursty behavior. In particular,

an application sees a sudden request that requires a timely response. This translates into an immediate surge of computational demand on the processor side to react to the user's request within a pre-defined quality-of-service time limit.

Such an environment and workload behavior are not the direct target of most of the optimizations proposed in this thesis. In other words, we do not seek to minimize the response time of the processor for time-critical applications. However, we would like to emphasize the benefits of our proposed power management techniques in cases of close relation to these environments and applications. In particular, this chapter has demonstrated two cases where our techniques are necessary for optimizing the power management under strict performance requirements.

First, for any application, pruning the power holes as described in Algorithm 3 is required to ensure no power is lost unnecessarily. For example, if the desired performance can be achieved by a power hole operating point on either the big or little cores, a scheduling technique could be easily deceived to use such a point as the best operating point that meets the performance requirement while minimizing power consumption. However, we have shown that such a point wastes power because other points could achieve the same performance at a lower power budget. Pruning is necessary to guarantee that the application is operating between the optimal operating points that guarantee the lowest power consumption.

We have shown the advantages of our techniques when co-locating a time-critical application with multiple applications with no strict performance demand (see Section 5.6.3). For a power-limited processor, this case mandates that enough power is allocated for the time-critical application to meet its target performance requirement. The rest of the power budget is then partitioned among the other applications. The benefit of our proposed techniques are two-fold in this case. First, by sifting the sub-optimal power holes we guarantee that the time-critical application meets its performance target with minimal power consumption. In other words, when compared to techniques that do not filter power holes, PH-Sifter can free more power budget that can be allocated to the other non-critical applications. Second, with any remaining power budget, our approach maximizes system throughput of the remaining tasks that run concurrently on the processor.

5.8 Related Work

Techniques for maximizing performance under a power constraint in the presence of multiple V-F operating points have be proposed by academic

and industrial research. Although these prior technique suite homogeneous CMPs with multiple V-F operating points, they are not readily applicable to HCMPs. For example, Isci et al. [42] relies on brute-force search to find the optimal V-F operating point assignment for each application such that performance is maximized within a power limit. Such an approach does not scale as the number of applications and V-F operating points increase. With the possibility to temporarily exceed the power limit by migrating between two operating points above and below the power limit, the number of possibilities increases, making this approach impractical. Ma et al. [26] improve the power management scalability by dividing the cores on the chip into groups based on the applications they are executing. They provide a scheme to allocate a "frequency" budget to each group of cores based on the power efficiency of the applications using performance per Watt as a ranking metric. We have shown in the previous chapter the significant performance degradation that can be caused by partitioning the power budget based on this misleading metric. More importantly, this technique does not provide a scalable solution to use when each application can run on multiple core types each featuring several V-F operating points. More recently, Su et al. [68] show how prior work that search incrementally for the V-F setting that maximizes performance under a power limit requires significant time overhead to reach the correct V-F setting. They propose a practical method based on performance counters that are implemented in the processor to predict the power and performance of each V-F operating point to fasten the search process. However, as we have demonstrated, after forming the power-performance curve, this method walks the curve naively and falls into the inefficient power holes.

Scheduling application on HCMPs and maximizing their efficiency is also an on-going effort in industry. Despite implementation differences among vendors, power management in recent systems rely on multiple components that act independently. The OS scheduler in one hand is responsible for selecting the cores to assign each application to based on a load metric. A frequency governor is used to select the V-F setting on the core [5]. The *Ondemand* governor periodically checks whether the CPU utilization has crossed a specific threshold (e.g., 95%), it then jumps to maximum frequency. The *Interactive* governor tries to adapt to faster based on the expected load level and adjust the frequency according to the load. This model of operation does not suite HCMPs because the optimal core type and the operating frequency need to be selected together to avoid wasting power or degrading performance. Recent scheduler implementations from Qualcomm try to achieve this goal by allowing the scheduler to pass hints to the frequency governor about the expected load [71]. A more comprehensive solution, known as the Energy-

Aware Scheduler (EAS), is being developed by Linaro and ARM, aiming to integrate the decision making of which core and frequency to assign to each task into the scheduler [72]. The optimizations envisioned by this project includes improving energy efficiency by making better choices of whether to increase the frequency setting for one task or move it to a more powerful core at a lower frequency, and load consolidation, among others. All the current proposals are not well suited to maximize performance on HCMPs under a given power budget despite adapting to interactive tasks or improving energy efficiency. Moreover, these proposal do not rigorously consider which V-F states can act as power holes that can significantly degrade performance.

It is worth noting that most of the decisions made by mainstream frequency governors are based on CPU utilization. An application waiting for a memory access is still considered utilizing the core. Therefore, such an approach leads to inaccurate power management policies. Other frequency governors that rely on power and performance models to optimize the selection of the V-F setting have been proposed, such as the *green governors* [73]. Similar to [68], these governors are not suited to walking the V-F curves in a heterogeneous processor.

5.9 Summary

We demonstrate the necessity to properly sift the default set of operating points when optimizing for performance in power-limited HCMPs with multiple V-F operating points per core type. We show that naively walking the power-performance curve of the default operating points leads to power holes, or points that waste power for low performance benefit. Selecting operating points using Pareto-optimal sifting suffers similar performance degradation. We propose PH-Sifter, a fast operating point sifting technique that starts at the lowest operating point, filters all the points until the next operating point that achieves the highest Delta Performance / Delta Power value, and continues until all the points are considered. We reason about the optimality of PH-Sifter and show significant performance improvements compared to Pareto-sifting for a single application, multiple applications, and systems with high-priority applications.

Chapter 6

Conclusions

6.1 Summary

Technology scaling trends have resulted in processors that have abundant transistors but suffer from a high power density due to the end of Dennard scaling. Processor designers are forced to activate only a limited fraction of the processor at any time to limit total power consumption and avoid thermal problems, a phenomenon known as dark silicon. In this era of dark silicon, power has become a scarce resource for processor operation. Maximizing performance requires novel techniques that optimally exploit the available power budget.

In this work, we focus on heterogeneous chip-multiprocessors (HCMPs) as a promising technique to cope with limited power budgets. HCMPs mix high-performing but power-hungry 'big' cores with power-efficient but low-performance 'little' cores. Under power constraints, applications can run on the big core when power headroom is available but then they must return to a little core to cool the processor down afterwards. Optimized scheduling techniques are essential to successfully use the capabilities of HCMPs to squeeze every drop of performance from the available power budget. Unfortunately, a wide body of scheduling work that optimizes performance under power limits using DVFS is not readily applicable to HCMPs.

Our goal in this work is to propose novel scheduling solutions that maximize performance on power-limited HCMPs. To that end, we zoom into the layers of this composite problem, and address each of them separately. Maximizing performance requires optimizing the power budget partitioning among the multiple applications that may concurrently utilize the processor. An optimized solution is also required to leverage the power budget allocated for

individual applications by scheduling their different phases on big and little cores during the course of application execution. An optimized scheduler must also cater for the complex cases where HCMPs consist of multiple core types each featuring several voltage-frequency operating points. We adopt a generalized definition of a power constraint that allows the power consumption to momentarily exceed the power limit as long it is preserved over a technology-dependent period of time, which we call the power period. Prior techniques have leveraged this flexible view of a power constraint to improve responsiveness of interactive applications. In this work, we exploit this relaxed view of a power constraint to maximize sustained performance. The added flexibility comes at the expense of a more complicated scheduling problem because it adds a time dimension to optimize over. Our optimization target, along with the power definition we adopt, sets our work apart from all prior work that target other optimization objectives or conservatively assume power consumption can never exceed the power limit. Our contributions in this work address various parts of the problem.

Optimizing Performance for Single-Threaded Applications. We aim to maximize the sustained performance of single-threaded applications. We show that the most relevant technique targeting sustained performance, called sprint-and-rest, fails to leverage the heterogeneity available in an HCMP processor. Sprint-and-rest has been proposed for multi-threaded applications. When adopted for single-threaded applications, sprint-and-rest exceeds the power limit as it runs on the big core (sprint) and then cools the processor down (rest) by turning it off. We show that sprint-and-rest degrades performance significantly due to ignoring the heterogeneous cores of the processor. We propose a new technique, called sprint-and-walk, that exploits the heterogeneous core types in the processor. Sprint-and-walk estimates the total energy it is allowed to consume within the power period based on the allowed sustained power limit. It uses the little core, which consumes power at a rate lower than the allowed limit, to accumulates energy credit. It then burns the accumulated credit by running on the big core, which normally dissipates power above the limit. The scheduler repeats the same cycle of sprinting on the big core then walking on the little core throughout the application execution. Using exhaustive search, we also determine the highest performance attainable through scheduling under a power limit. Our results show that sprint-and-walk improves performance over sprint-and-rest by 9% on average across all SPEC CPU2006 applications, that reaches up to 19%, for a moderate power budget of 1.25 W. The improvement increases as the power budget gets tighter. For a budget of 0.5 W, the average improvement of sprint-and-walk relative to sprint-and-rest equals 43% and reaches up to 76%. More importantly, sprint-and-walk

achieves a performance improvement that is within a few percents of optimal performance.

Optimizing the Performance for Multi-Programmed Workloads. We seek to optimize the performance for an HCMP running a multi-programmed workload. This task relies on optimally dividing the power budget among the concurrent applications. We show that both techniques that partition the power budget equally among the co-running applications and techniques that let the co-executing applications greedily compete over the shared power budget lead to sub-optimal performance. These problems ignore the power and performance characteristics of the co-executing applications. We show that even techniques that rely on commonly used scheduling metrics, e.g., performance per Watt or big-to-little performance ratio, are also misleading. These ranking metrics do not consider the restricted big core utilization imposed by the limited power budget. Therefore, they cannot accurately estimate each application's benefits from the allocated power budget. Counter to intuition, we show that in many cases, memory-intensive applications dissipate less power than compute-intensive ones, allowing them to utilize the big core for a longer time duration, leading to higher overall performance with the same power budget. In addition to the lack of a metric that prioritizes the applications when partitioning the power budget, there is also a lack of a strategy to determine the fraction of the budget each application should receive after they have been ranked.

We formulate the problem as a linear programming optimization problem and mathematically solve it to derive an optimal power budget partitioning strategy. We show that allocating power to each application such that it can only partially run on the big core is sub-optimal. An optimal partitioning of a budget *"P"* over *"n"* application schedules all applications either on the big or little cores, except for one application that needs to migrate between the two. We also derive a novel ranking metric based on big-to-little delta performance/delta power (DPe/DPo) for each application. This metric enables a fast and scalable way to sort the applications according to their worthiness of running on the big core. We propose DPDP, a fast and scalable scheduler that maximizes performance on a power-limited HCMP by optimizing the power budget partitioning among concurrent applications. Our evaluations with DPDP on a heterogeneous processor consisting of four big.LITTLE pairs show that DPDP improves chip performance by 16% on average and up to 40% over a strategy that greedily and globally utilizes the power budget. We also show that DPDP outperforms all the other techniques as we change the available power budget. However, if the power budget is either too restrictive or too loose, the opportunity for optimization reduces. Similarly, the performance improvements of DPDP remain significant even when we

change the mix of cores in the HCMP. Interestingly, our results indicate that using an out-of-order little core yields a lower performance improvement over the other techniques. The bigger gap between the power and performance characteristics of the little core and the big core makes scheduling decisions of a higher impact on the overall performance.

We analyze DPDP's impact on the per-application performance, and propose an extension that allows DPDP to improve overall throughput while remaining within a tolerable application slowdown. Our results show that DPDP can strike a sweet spot that allows it to considerably improve system throughput while slightly improving the application latency.

Optimizing Performance on HCMPs with Multiple V-F Operating Points. Most proposals for power management are proposed for DVFS, and cannot schedule applications across the heterogeneous cores in HCMPs composed of an arbitrary number of core types (e.g., big, medium, and little), each equipped with multiple voltage-frequency operating points. The scheduler has an exploded set of options, and has to assign each application to an operating point at any core type and migrate it to any other operating point on any core when necessary. To find the points that maximize performance for a give power budget, the scheduler typically walks the performance-power curve starting at the lowest operating point on the little core. It keeps walking the curve until it reaches the first operating point that exceeds the power limit. To leverage the whole power budget, the scheduler continuously migrates the application between this point and the last point it encountered that does not exceed the power limit. The two selected points could be on two different cores. We show that naively walking the default set of operating points leads the application to inefficient operating points which drain power without significant performance benefit. We call these points Power Holes (PH) as they drain the power budget for sub-optimal performance. Contrary to intuition, we show that even using a power-performance curve of Pareto-optimal operating points still degrades performance significantly for the same reason. We propose PH-Sifter, a fast and scalable technique that sifts the default set of operating points and eliminates power holes. We show significant performance improvements for PH-Sifter compared to Pareto-sifting for three use cases: (i) maximizing performance for a single application, (ii) maximizing system throughput for multi-programmed workloads, and (iii) maximizing performance for a system in which a fraction of the power budget is reserved for a high-priority application. Our results show performance improvements of 13, 27, and 28 percent on average that reach up to 52, 91 percent, and 2.3x respectively, for the three use cases.

6.2 Future Work

As the abundant transistors are being used to integrate more functional and specialized units to improve performance, we can foresee the necessity to extend our work to investigate emerging trends in processor design. Here we describe a couple of opportunities: (i) extending the work to include other forms of heterogeneity; (ii) considering heterogeneous cache and memory hierarchies.

One way of combating dark silicon relies on heterogeneous architectures. In this work we focused on processors with heterogeneous cores. Other forms of heterogeneity have been proposed to improve performance in the era of dark silicon. We could imagine a processor tailored from single-ISA heterogeneous multi-cores, specialized or FPGA-based accelerators, and graphics processing units (GPUs). Maximizing performance of such a system under power constraints involves coordinating the execution of all these units. Partitioning the power budget among the concurrent applications or kernels, selecting when to activate the particular units, and the operating frequency to use for each particular computation. Among the challenging tasks a scheduler has to partake is the proper estimation of the power consumption and the performance that can be achieved by running on each computation resource, and the impact of co-executing multiple applications and kernels on such a heterogeneous platform. Studies have reported a shift from the original workload behavior as proportions of the workload are distributed among the various components. The dynamic change in the workload behavior should be taken into account when determining the proper ways to optimize for performance under a power constraint. The mathematical formulation we used in this thesis may need to be reexamined to account for all the extra variables. A fast and scalable policy is expected to be more critical due to the significant expansion of the solution space.

We assume a power limit on a processor that employs heterogeneity at the core level. As memory systems are evolving to deploy heterogeneous technologies at various levels of the stack, we look to support a more generalized memory hierarchy. In particular, processor manufacturers are moving towards stacking DRAM modules on the same die as the processor. Regardless of the stacking technology and the role of the DRAM components (i.e., main memory or cache), these stacked memory modules share the power budget with the processor. A general power management strategy is necessary to decide whether the application benefits from the stacked DRAM or not, whether it should run on the big or the little core with the stacked DRAM, and how much of the stacked DRAM is sufficient for the mix of applications given under a limited power budget. The problem is further complicated

when considering multiple voltage-frequency operating points per core, and perhaps for the stacked DRAM modules. Devising architectural techniques to tune the power consumption of the DRAM modules could be a challenge on its own. Such a task involves proposing strategies to control the DRAM refresh operation, and intelligently partitioning the dataset among DRAM modules to enable fast activation and deactivation of the modules while incurring minimal overhead for warm-up or copying dirty data blocks.

Bibliography

[1] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," ARM White paper, September 2011.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[4] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[5] M. Själander, M. Martonosi, and S. Kaxiras, "Power-efficient computer architectures: Recent advances," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 3, pp. 1–96, 2014.

[6] A. Raghavan, Y. Luo, A. Chandawalla, M. C. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational sprinting," in *18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 249–260.

[7] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, no. 2, pp. 20–27, 2012.

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS), 2012, pp. 37–48.

[9] J. Birnbaum and R. S. Williams, "Physics and the information revolution," *Physics Today*, vol. 53, no. 1, pp. 38–43, 2000.

[10] R. Merritt, "Arm cto: power surge could create'dark silicon'," *EE Times, Oct*, 2009.

[11] A. Shafaei Bejestan, Y. Wang, S. Ramadurgam, Y. Xue, P. Bogdan, and M. Pedram, "Analyzing the dark silicon phenomenon in a many-core chip multi-processor under deeply-scaled process technologies," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015, pp. 127–132.

[12] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 205–218.

[13] Samsung Electronics, "Samsung primes Exynos 5 Octa for ARM big.LITTLE technology with heterogeneous multi-processing capability," Press release, September 2013.

[14] NVIDIA, "Variable SMP – a multi-core CPU architecture for low power and high performance," White paper, 2011.

[15] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer, "QuickIA: Exploring heterogeneous architectures on real prototypes," in *18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–8.

[16] T.-Y. Lin, M.-H. Lee, L. Chou, C. Peng, J.-M. Hsu, J.-M. Chen, J.-C. Chen, A. Chiou, A. Chiu, D. Lee *et al.*, "Helio x20: The first tri-gear mobile soc with corepilot™ 3.0 technology," in *IEEE 28th Hot Chips Symposium (HCS)*, 2016, pp. 1–24.

[17] B. Jeff, "big.LITTLE technology moves towards fully heterogeneous global task scheduling," ARM White paper, November 2013.

[18] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic performance setting for dynamic voltage scaling," in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, 2001, pp. 260–271.

[19] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2003, pp. 38–48.

[20] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer, "Energy-conscious compilation based on voltage scaling," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES)*, 2002, pp. 2–11.

[21] C. Isci and M. Martonosi, "Identifying program power phase behavior using power vectors," in *IEEE International Workshop on Workload Characterization (WWC)*, 2003, pp. 108–118.

[22] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006, pp. 359–370.

[23] J. Donald and M. Martonosi, "Techniques for multicore thermal management: Classification and new exploration," in *33rd International Symposium on Computer Architecture (ISCA)*, 2006, pp. 78–88.

[24] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 123–134.

[25] S. Eyerman and L. Eeckhout, "A counter architecture for online DVFS profitability estimation," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1576–1583, 2010.

[26] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 449–460.

[27] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 511–522.

[28] S. Akram, J. B. Sartor, and L. Eeckhout, "DVFS performance prediction for managed multithreaded applications," in *International Symposium*

on Performance Analysis of Systems and Software (ISPASS), 2016, pp. 12–23.

[29] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *36th International Symposium on Microarchitecture (MICRO)*, 2003, pp. 81–92.

[30] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 64–75.

[31] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, 2008.

[32] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 213–224.

[33] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference (DAC)*, 2009, pp. 927–930.

[34] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[35] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *45th International Symposium on Microarchitecture (MICRO)*, 2012, pp. 317–328.

[36] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 137–149.

[37] V. Petrucci, M. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, L. Tang *et al.*, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 246–258.

[38] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 177–187.

[39] AMD, "10h desktop processor power and thermal data sheet," White paper.

[40] S. Gunther, A. Deval, T. Burton, and R. Kumar, "Energy-efficient computing: power management system on the Nehalem family of processors." *Intel Technology Journal*, vol. 14, no. 3, 2010.

[41] A. Raghavan, L. Emurian, L. Shao, M. C. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational sprinting on a hardware/software testbed," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 155–166.

[42] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, 2006, pp. 347–358.

[43] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, pp. 363–374.

[44] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive DVFS and thread packing under power caps," in *Proceedings of the 44th annual International Symposium on Microarchitecture (MICRO)*, 2011, pp. 175–185.

[45] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009, pp. 314–324.

[46] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 29–40.

[47] A. Lukefahr, S. Padmanabha, R. Das, R. Dreslinski Jr, T. F. Wenisch, and S. Mahlke, "Heterogeneous microarchitectures trump voltage scaling for low-power cores," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 237–250.

[48] H. H. Najaf-abadi and E. Rotenberg, "Architectural contesting," in *15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 189–200.

[49] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[50] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, p. 28, 2014.

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[52] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies, "The fuzzy correlation between code and performance predictability," in *37th International Symposium on Microarchitecture (MICRO)*, 2004, pp. 93–104.

[53] Samsung Electronics, "A mobile processor that goes beyond mobile innovation," Press release, September 2017.

[54] NVIDIA, "Nvidia Tegra K1 – a new era in mobile computing," White paper, 2014.

[55] NVIDIA, "Nvidia Tegra X1 – Nvidia's new mobile superchip," White paper, 2015.

[56] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing Frontiers*, 2006, pp. 29–40.

[57] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 125–138.

[58] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, p. 25.

[59] S. Ghiasi, T. Keller, and F. Rawson, "Scheduling for heterogeneous processors in server systems," in *Proceedings of the 2nd Conference on Computing Frontiers*, 2005, pp. 199–210.

[60] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[61] B. Kuttana, "Technology insight: Intel Silvermont microarchitecture," Intel Developer Forum, 2013.

[62] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004, pp. 81–92.

[63] D. Brooks and M. Martonosi, "Dynamic thermal management for high-performance microprocessors," in *7th International Symposium on High-Performance Computer Architecture (HPCA)*, 2001, pp. 171–182.

[64] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin, "Utilizing dark silicon to save energy with computational sprinting," *IEEE Micro*, vol. 33, no. 5, pp. 20–28, 2013.

[65] S. Fan, S. M. Zahedi, and B. C. Lee, "The computational sprinting game," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 561–575.

[66] T. S. Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores." in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 161–176.

[67] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili, "Cooperative boosting: Needy versus greedy power management," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 285–296.

[68] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "PPEP: Online performance, power, and energy prediction framework and DVFS space exploration," in *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, 2014, pp. 445–457.

[69] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010, pp. 26–36.

[70] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of DVFS for realistic memory systems," in *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO)*, 2012, pp. 155–165.

[71] J. Ho and A. Frumusanu, "Understanding Qualcomm's Snapdragon 810: Performance preview," 2015. [Online]. Available: https://www.anandtech.com/show/8933/snapdragon-810-performance-preview/4

[72] I. Rickards and A. Kucheria, "Energy-aware scheduling (EAS) progress update," 2015. [Online]. Available: https://www.linaro.org/blog/energy-aware-scheduling-eas-progress-update/

[73] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS," in *Green Computing Conference and Workshops (IGCC)*, 2011, pp. 1–8.