POWER-CONSTRAINED AWARE AND LATENCY-AWARE MICROARCHITECTURAL OPTIMIZATIONS IN MANY-CORE PROCESSORS

by Sudhanshu Shekhar Jha

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Power-constrained Aware and Latency-aware Microarchitectural Optimizations in Many-core Processors

by
Sudhanshu Shekhar Jha

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

Doctor of Philosophy / Doctor por la UPC

in

Computer Architecture

Departament d' Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Advisors: Ayose Falcón, Antonio González & Jordi Tubella

September 2016

Dedication

To my family — parents, siblings and kids

Abstract

As the transistor budgets outpace the power envelope (the *power-wall* issue), new architectural and microarchitectural techniques are needed to improve, or at least maintain, the power efficiency of next-generation processors. Run-time adaptation, including core, cache and DVFS adaptations, has recently emerged as a promising area to keep the pace for acceptable power efficiency.

However, none of the adaptation techniques proposed so far is able to provide good results when we consider the stringent power budgets that will be common in the next decades, so new techniques that attack the problem from several fronts using different specialized mechanisms are necessary. The combination of different power management mechanisms, however, bring extra levels of complexity, since other factors such as workload behavior and run-time conditions must also be considered to properly allocate power among cores and threads.

To address the power issue, this thesis first proposes Chrysso, an integrated and scalable model-driven power management that quickly selects the best combination of adaptation methods out of different core and uncore micro-architecture adaptations, per-core DVFS, or any combination thereof. Chrysso can quickly search the adaptation space by making performance/power projections to identify Pareto-optimal configurations, effectively pruning the search space. Chrysso achieves 1.9× better chip performance over core-level gating for multi-programmed workloads, and 1.5× higher performance for multi-threaded workloads.

Most existing power management schemes use a centralized approach to regulate power dissipation. Unfortunately, the complexity and overhead of centralized power management increases significantly with core count rendering it in-viable at fine-grain time slices. The work leverages a two-tier hierarchical power manager. This solution is highly scalable with low overhead on a tiled many-core architecture with shared LLC and per-tile DVFS at fine-grain time slices. The global power is first distributed across tiles using GPM and then within a tile (in parallel across all tiles). Additionally, this work also proposes DVFS and cache-aware thread migration (DCTM) to ensure optimum per-tile co-scheduling of compatible threads at runtime over the two-tier hierarchical power manager. DCTM outperforms existing solutions by up to 12% on adaptive many-core tile processor.

With the advancements in the core micro-architectural techniques and technology scaling, the performance gap between the computational component and memory component is increasing significantly (the *memory-wall* issue). To bridge this gap, the architecture community is pushing forward towards multi-core architecture with on-die near-memory DRAM cache memory (faster than conventional DRAM). Gigascale DRAM Caches poses a problem of how to efficiently manage the tags. The Tags-in-DRAM designs aims at efficiently co-locate tags with data, but it still suffer from high latency especially in multi-way associativity.

The thesis finally proposes Tag Cache mechanism, an on-chip distributed tag caching mechanism with limited space and latency overhead to bypass the tag read operation in multi-way DRAM Caches, thereby reducing hit latency. Each Tag Cache, stored in L2, stores tag information of the most recently used DRAM Cache ways. The Tag Cache is able to exploit temporal locality of the DRAM Cache, thereby contributing to on average 46% of the DRAM Cache hits.

Acknowledgement

Firstly, I would like to thank Prof. Lieven Eeckhout and Wim Heirman for their direct supervision and guidance throughout this work. Though, I was not part of their research group, they were very supportive and extremely approachable. They helped me develop a methodology for critical thinking which helped me with my research. It has driving force for this thesis and will continue to guide my career forward.

Ayose Falcón, my PhD guide has been very supportive and extremely approachable throughout my research. I cannot thank him enough for the last five years of weekly discussions and guidance which has significantly impacted this thesis.

Prof. Antonio González and Prof. Jordi Tubella have helped with tutoring this thesis work and have been instrumental in providing the financial support and logistical support for my research.

My parents have been a source of inspiration throughout my life and encouraged me in taking up research and engineering. My sister, brother-in-law and brother have provided their constant support and shown infinite patience. My nieces and nephew constantly came up with lighter moments in my life which nicely balanced the oft-experienced stress that a researcher goes through. Words are not enough to describe their contribution to my life.

A special thanks to Sabria family Marta, Joaquim and their parent (Joaquim and Maria Teresa) have been very patient, generous and kind to me. They treated me as a close family friend. They have made my stay in Barcelona very comfortable and memorable with countless help and invitation to their house for get-together.

My friends and lab mates got my creative juices flowing with numerous discussions on random topics of research in microarchitecture.

Lastly, thanks to everyone at ARCO, Intel Barcelona Research Center and DAC-UPC for their help and support.

Thanks to Universitat Politècnica de Catalunya (UPC) for awarding me the FPI-UPC fellowship and funding my research and the DAC administration for arranging trips to the conferences and solving countless administrative problems.

Barcelona, 2016

Table of Contents

Dedication iii

Abstract v

Acknowledgement vii

List of Figures xiii

List of Tables xv

List of Acronyms xvii

Chapter 1. Introduction 1

1.1 Performance: Issues and Challenges . . . 1

1.1.1 Multi/Many-core Processor . . . 2

1.1.2 Issues in the Era of Many-core Processor . . . 4

1.2 Power/Energy Issues . . . 5

1.2.1 Power Dissipation in Processors . . . 5

1.2.2 Schemes for Reducing Power and Energy . . . 6

1.3 Problem Statement . . . 8

1.3.1 Scalable Power Management on Many-core Processors . . . 8

1.3.2 Die-stacked Gigascale DRAM Cache Latency . . . 9

1.4 Thesis Scope . . . 9

1.5 Thesis Layout . . . 10

Chapter 2. Background: Overview and Related Work 11

2.1 Power Management Schemes . . . 11

2.1.1 Isolated Mechanisms . . . 11

2.1.2 Co-ordinated Approaches . . . 16

2.2 Near-Memory Architecture . . . 17

2.2.1 Designing DRAM Caches . . . 18

2.2.2 Block-Based Caches . . . 19

2.2.3 Page-Based Caches . . . 19

I Optimizing Performance under Power Constraints 21

Chapter 3. Integrated Power Management in Constrained Many-Core Processors 23

3.1 Introduction . . . 23
3.2 Chrysso Overview . . . 25
3.2.1 Many-core power management . . . 25
3.2.2 Chrysso adaptation . . . 25
3.3 Chrysso Optimization Algorithm . . . 26
3.3.1 Per-core prediction and Pareto-optimal search . . . 26
3.3.2 Utility-driven optimization . . . 26
3.3.3 Critical-thread aware Chrysso . . . 27
3.3.4 Optimizing for other criteria . . . 28
3.4 Chrysso Projection Models . . . 28
3.4.1 Performance projection . . . 28
3.4.2 Power projection . . . 31
3.4.3 Hardware support . . . 31
3.5 Experimental Setup . . . 32
3.5.1 Simulator . . . 32
3.5.2 Workloads . . . 32
3.5.3 Adaptive many-core architecture . . . 33
3.5.4 Alternate power management policies . . . 34
3.6 Results and Discussion . . . 35
3.6.1 Offline Analysis . . . 35
3.6.2 Chrysso Evaluation . . . 37
3.7 Related Work . . . 45
3.7.1 Micro-architecture Adaptation . . . 45
3.7.2 Dynamic Power Management . . . 45
3.7.3 Critical Thread Acceleration . . . 46
3.8 Summary . . . 46

Chapter 4. Shared Resource Aware Scheduling on Power-Constrained Tiled Many-Core Processors 49

4.1 Introduction . . . 49
4.2 Motivation . . . 50
4.2.1 Limitations of a Centralized Approach . . . 50
4.2.2 Cache-aware Thread Migration (Cruise) . . . 51
4.3 Two-Tier Hierarchical Power Management . . . 52
4.4 DVFS and Cache-aware Thread Migration (DCTM) . . . 53
4.4.1 DVFS and LLC Sensitivity Analysis . . . 53
4.4.2 DCTM Scheduling Rules . . . 54
4.4.3 Putting It All Together . . . 55
4.4.4 Quantifying DVFS Sensitivity: DCTM vs. Cruise . . . 55
4.5 Experimental Setup . . . 57

4.5.1 Simulation Framework . . . 57
4.5.2 Adaptive Micro-Architecture . . . 57
4.5.3 Workloads . . . 59
4.6 Evaluation . . . 60
4.6.1 Hierarchical vs. Centralized Power Management . . . 60
4.6.2 Two-Tier Approach: Performance vs. Power Budget . . . 62
4.6.3 Static Assignment vs. Dynamic Migration . . . 63
4.6.4 Sensitivity to Migration Interval . . . 66
4.6.5 Fine-grained Power Redistribution . . . 67
4.7 Related Work . . . 67
4.7.1 Micro-architecture Adaptation . . . 67
4.7.2 Centralized Dynamic Power Management . . . 68
4.7.3 Tiled Architecture and Hierarchical Power Manager . . . 68
4.8 Summary . . . 68

II Reducing Die-Stack DRAM Hit Latency 71

Chapter 5. Tag Caching for Gigascale Die-Stacked DRAM Caches: A High-Performance Many-Core Perspective 73
5.1 Introduction . . . 73
5.2 Background and Motivation . . . 74
5.2.1 Impact of Associativity . . . 75
5.2.2 Cache Designs with Tags-in-DRAM . . . 75
5.2.3 Latency Mitigation Methods . . . 76
5.2.4 Ideal Tags-in-DRAM Cache . . . 76
5.3 Tag Cache . . . 77
5.3.1 Tag Cache Operations . . . 78
5.3.2 DRAM Cache Access Latency with Tag Cache . . . 78
5.4 Experimental Setup . . . 80
5.4.1 Simulation Framework . . . 80
5.4.2 Workloads . . . 82
5.5 Evaluation and Results . . . 82
5.5.1 Tags-in-DRAM Cache Performance . . . 82
5.5.2 Effectiveness of Tag Cache . . . 84
5.5.3 Tag Cache Sensitivity . . . 88
5.5.4 Closed versus Open Policy . . . 90
5.5.5 Energy Consumption . . . 93
5.6 Related Work . . . 93
5.7 Summary . . . 95

Chapter 6. Conclusions and Future Work 97
6.1 Overview . . . 97

6.2 Summary of Research 97
6.3 Potential Future Work 99
6.3.1 Adaptive SMT Cores 99
6.3.2 Power Management 99
6.3.3 Resizeable DRAM Cache 99
6.4 Work published 100

References **101**

List of Figures

Figure 1.1 Moore's Law and microprocessor transistor count.. 2

Figure 1.2 40 years of microprocessor data (or Microprocessor data [1975–2015]) [151].. . . . 3

Figure 3.1 Adaptive many-core architecture with Chrysso, featuring per-core power/performance monitoring units (PMU) and a global power manager (GPM).. 25

Figure 3.2 Event flow in Chrysso.. 26

Figure 3.3 The Chrysso algorithm consists of Pareto frontier selection followed by utility-based optimization.. 27

Figure 3.4 Per-knob modeling error of Chrysso's performance (*left*) and power (*right*) projection models.. 35

Figure 3.5 Effectiveness of Chrysso compared to pOracle through offline analysis for multi-program workloads.. 36

Figure 3.6 Relative STP vs. power budget: comparing Chrysso against alternative power management techniques for multi-program workloads.. 37

Figure 3.7 Performance vs. power budget of the evaluated power management techniques for multi-threaded benchmarks. 38

Figure 3.8 Critical-thread aware Chrysso for the thread-imbalance workloads NPB (MG), PARSEC (blackscholes) and SPEC OMPM2001 (gafort_m).. 38

Figure 3.9 Isolated versus integrated optimization using Chrysso for the WL0 multi-program workload.. 39

Figure 3.10 Number of cores that are adapted for each knob as a function of the available power budget for WL0.. 40

Figure 3.11 Average normalized turnaround time (ANTT) of Chrysso and core gating for the WL0 multi-program workload. *(Lower is better.)*. 40

Figure 3.12 Energy per instruction versus performance at various power settings.. 41

Figure 3.13 Chrysso configuration changes through time at 50% power setting and a 1 ms (reconfiguration) time slice.. 42

Figure 3.14 Chrysso at different reconfiguration time slice at 50% power setting.. 44

Figure 3.15 Total and per-core power (top) and performance (bottom) for swaptions at 50% TDP.. 44

Figure 3.16 Average number of adaptations per time slice (10 ms) for the 64 cores running WL0.. 45

Figure 4.1 Generic tiled many-core architecture with centralized (top) versus hierarchical (bottom) power management.. 51

Figure 4.2 Normalized runtime overhead for hierarchical power management with varying tile size at 1 ms time slice.. 53

Figure 4.3 Application classification based on LLC and DVFS sensitivity.. 54

Figure 4.4 *DCTM* and *Cruise* through time for 4 cores with LLCT applications.. 56

Figure 4.5 STP (normalized to *Centralized*) for *Hierarchical* and *DCTM* at 60% power budget vs. core count.. 60

Figure 4.6 STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 64-core setup.. 61

Figure 4.7 STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 128-core setup.. 62

Figure 4.8 STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 256-core setup.. 63

Figure 4.9 *Static* assignment and *DCTM* through time for selected SPEC CPU2006 applications on 64-core setup at 80% Power Budget.. 64

Figure 4.10 *Static* versus *DCTM* relative to *Hierarchical* for *WL0*.. 65

Figure 4.11 Sensitivity to DCTM's thread migration interval for *WL0* on 64 cores and L2 invalidates: *Hierarchical* vs *DCTM* for single-thread *WL0* workload.. 66

Figure 4.12 Fine-grain (*Hierarchical*) versus coarse-grain (*CPM*) power redistribution for the two-tier hierarchical power manager for *WL0* on 64-core setup.. 67

Figure 5.1 Tags-in-DRAM Cache Design: Loh-Hill Cache (left) and Alloy Cache (right). 75

Figure 5.2 Tiled Many-core Architecture with an on-package DRAM Cache and an on-die distributed Tag Cache.. 77

Figure 5.3 Closed-page latency with Tag Cache (TC) hit/miss. (DRAM Cache tag accesses are shown in gray.). 79

Figure 5.4 Open-page latency with Tag Cache (TC) hit/miss and row buffer (RB) hit/miss. (DRAM Cache tag accesses are shown in gray.). 79

Figure 5.5 Hit rate (%) for the Alloy Cache and Loh-Hill Cache. 83

Figure 5.6 STP (relative to no DRAM Cache) for the Alloy Cache and Loh-Hill Cache. 83

Figure 5.7 Performance comparison with tag lookup and Oracle Tags for Loh-Hill Cache. 84

Figure 5.8 Shared L2 miss rate (*LHC* normalized). 85

Figure 5.9 Average LLC miss latency breakdown. 86

Figure 5.10 Average DL1 miss latency for the 2 GB Loh-Hill Cache variants. 87

Figure 5.11 Relative STP for the 2 GB Loh-Hill Cache variants. 87

Figure 5.12 Hit rate for the 2 GB Loh-Hill Cache variants. 88

Figure 5.13 Tag Cache sensitivity wrt. associativity for a 2 GB DRAM Cache. 89

Figure 5.14 Relative STP versus DRAM Cache size. 89

Figure 5.15 DRAM Cache (w/ Tag Caching) hit rate sensitivity to size. 90

Figure 5.16 Relative STP for all multi-program workloads on 64-core setup with a 2 GB DRAM Cache considering closed and open-page policies.. 91

Figure 5.17 Average DL1 miss latency for all multi-program workloads on 64-core setup with a 2 GB DRAM Cache considering closed and open-page policies.. 92

Figure 5.18 Energy reduction (*LHC* normalized) for a 64-core setup with 2 GB DRAM Cache (shaded region denote static energy); higher is better.. 94

List of Tables

Table 3.1 Configuration knobs and corresponding architectural parameters and values.. 29
Table 3.2 Multi-programmed workloads.. 33
Table 3.3 Multi-threaded workloads.. 33
Table 3.4 Base configuration.. 34
Table 4.1 Micro-architectural adaptations. 57
Table 4.2 Tile-based many-core architecture.. 58
Table 4.3 Workloads.. 59
Table 5.1 Simulated architecture configuration.. 80
Table 5.2 Tags-in-DRAM Cache Configurations.. 81
Table 5.3 Workloads.. 81
Table 5.4 Tag Cache Configuration.. 82

List of Acronyms

ACET	Average Case Execution Time
ANN	Artificial Neural Network
ANTT	Average Normalized Turnaround Time
APKI	Access Per Kilo Instructions
ARM	Advance RISC Machine
ATD	Auxiliary Tag Directory
CAM	Content-Addressable Memory
CPI	Cycles Per Instruction
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage Frequency Scaling
DVS	Dynamic Voltage Scaling
FIVR	Fully Integrated Voltage Regulator
GPM	Global Power Management
IPC	Instructions Per Cycle
IPS	Instructions Per Second
LLC	Last Level Cache
LQ	Load Queues
LRU	Least Recently Used
MESI	**M**odify **E**xclusive **S**hared **I**nvalid
MIPS	Million Instructions Per Second
MPARM	Multi-Processor ARM
MPKI	Misses Per Kilo Instructions
MPSoC	Multiprocessor System-on-Chip
MR	Miss Rate
NoC	Network-on-Chip
OS	Operating System
PMU	Power/Performance Monitoring Unit
RISC	Reduced Instruction Set Computer
ROB	Re-Order Buffer
RS	Reservation Station
SPI	Seconds Per Instruction
SQ	Store Queues
SRAM	Static Random-Access Memory
STP	System Throughput
SVM	Support Vector Machine
TD	Tag Directory
TDP	Thermal Design Power
TPM	Tile Power Management
UPS	Uninterruptible Power Source/Supply
WCET	Worst Case Execution Time

Chapter 1
Introduction

For several decades, semiconductor devices have seen tremendous progress in performance and functionality due to exponential growth in the number of transistors per chip. In 1971, the Intel 4004 processor held 2300 transistors. In early 2014, Intel released Xeon Ivy Bridge-Ex with more than 4.3 billion transistors [6]. This exponential growth in number of transistors is popularly known as Moore's law [129], which states that the number of transistors in a chip doubles approximately every two years. Each succeeding technology generation has introduced new obstacles in taking benefit from the increased chip transistor count.

First, the power dissipation of the microprocessors started reaching sky high and semiconductor industry hit the power wall, where the performance improvements of microprocessor were limited by power constraints [72]. It motivated the research in low power computing techniques such as dynamic voltage and frequency scaling (*DVFS*), near threshold computing (*NTC*) and sub-threshold operations. According to Dennard scaling [43], as transistors get smaller their power density stays constant, so that the power used stays in proportion with area. The breakdown of Dennard scaling and the failure of Moore's law to yield dividends in improved performance [24, 65] prompted a switch among chip manufacturers to focus on multi-core processors [96]. In the multi-core revolution with increasing number of cores, operating all cores simultaneously requires exponentially high energy per chip. However, whereas the energy requirements grow, chip power delivery and cooling limitations remain largely unchanged across technologies imposing the power wall [79]. As a result we will soon be incapable of operating all transistors simultaneously, pushing multi-core scaling to an end [52, 71]. This trend is leading us into an era of *dark silicon* [52] where we will be able to build denser devices but we will not be able to power them up entirely.

Second, the rate of improvement in microprocessor speed exceeds the rate of improvement in off chip memory (DRAM) speed [177]. This—referred as memory wall problem—drives the innovation towards creating low latency caches and other higher-level techniques such as prefetching [140, 146]. In addition technique like multi-threading [168] either reduces the memory latency, or keep the processor occupied for longer latency memory operations.

In this thesis, we address the power-wall and memory-wall issues in the future many-core processors and propose solutions to mitigate/manage these challenges. Before we dwell in the details of the thesis, we first describe the issues related to performance and power in multi/many-core processors.

1.1 Performance: Issues and Challenges

Moore's Law [129] (the doubling of transistors on chip every 2 years) has been a fundamental driver of computing, see Figure 1.1. In the past four decades, there has been a consistent and exponential increase

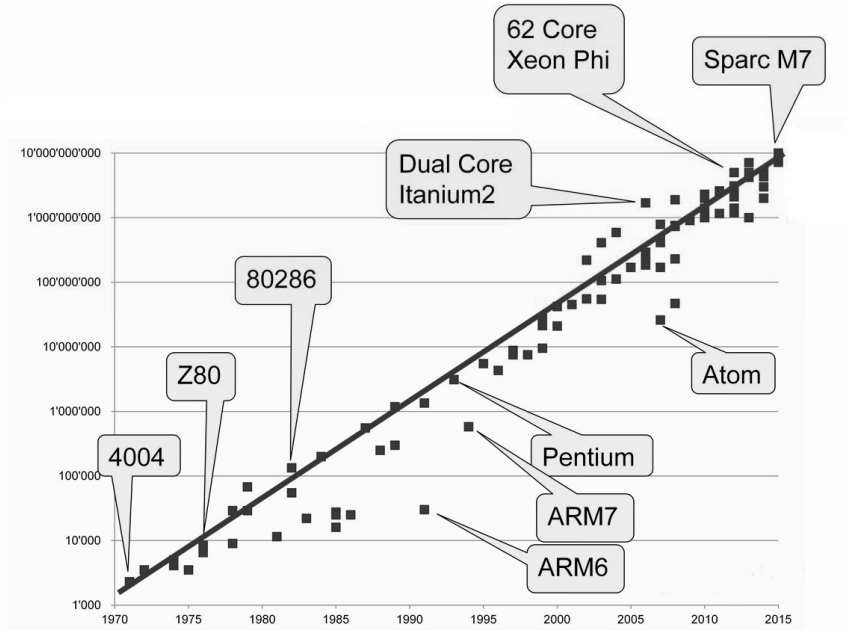


Figure 1.1: Moore's Law and microprocessor transistor count.

in performance of processors through advancements in transistors, circuits, micro architecture and compilers. Dennard's scaling theory showed how to reduce the dimensions and the electrical characteristics of a transistor proportionally to enable successive shrinks that simultaneously improved density, speed, and energy efficiency. The end of Dennard scaling, process technology scaling can sustain doubling the transistor count every generation, but with significantly less improvement in transistor switching speed and energy efficiency. This transistor scaling trend presages a divergence between energy efficiency gains and transistor density increases. The recent shift to multi-core designs, which was partly a response to the end of Dennard scaling, aimed to continue proportional performance scaling by utilizing the increasing transistor count to integrate more cores, which leverage application and/or task parallelism, see Figure 1.2. Even though power and energy have become the primary concern in system design, it is difficult to predict how severe the power problem will be for multi-core scaling, especially given the large multi/many-core design space.

Previous studies [79] showed that regardless of chip organization and topology, multi-core scaling is power limited to a degree underestimated by the computing community. Based on estimates in 2011, about 21% of a fixed-size chip would be powered off at 22 nm technology. Esmaeilzadeh et al. [52] estimated that at 8 nm, this number grows to more than 50%. The authors further emphasized that by 2024, only 7.9× average speedup is possible across commonly used parallel workloads, leaving a nearly 24-fold gap from a target of doubled performance per generation. Based on current technology scaling trend, the full scale commercial implementation of 10 nm technology node is expected to be available in 2017–2019 (based on Intel's new technology scaling trend, the 5 nm technology node will happen around 2020–2022 [92]). Further investigations also show that beyond a certain point increasing the core count does not translate to meaningful performance gains. These power and parallelism challenges threaten to end the multi-core era, defined as the era during which core counts grow appreciably.

1.1.1 Multi/Many-core Processor

The concept of multiple cores may seem trivial at first instance. However, as we will see in the section about scalability issues there are numerous trade-offs to be considered.

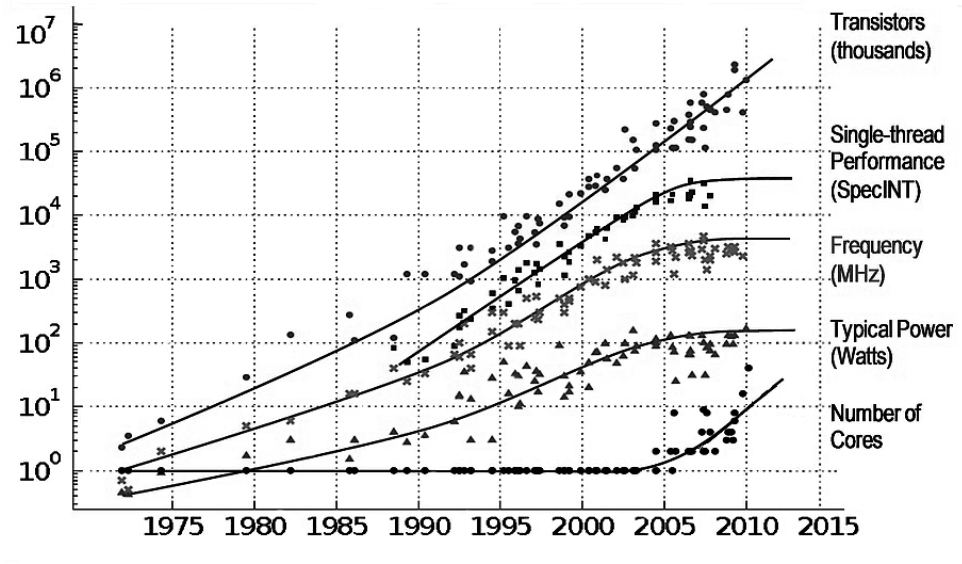


Figure 1.2: 40 years of microprocessor data (or Microprocessor data [1975–2015]) [151].

First of all, we need to consider whether the processor should be homogeneous or expose some heterogeneity. Most current general purpose multi-core processors are homogeneous both in instruction set architecture and/or performance. This means that the cores can execute the same binaries and that it does not really matter, from functional point of view, on which core a program runs. Recent multi-core architectures, however, allow for system software to control the clock frequency for each core individually in order to either save power or to temporarily boost single-thread performance. Most of these homogeneous architectures also implement a shared global address space with full cache coherency, so that from a software perspective, one cannot distinguish one core from the other even if the process (or thread) migrates during run-time.

On the contrary, a heterogeneous architecture features at least two different kinds of cores that may differ in both the instruction set architecture (ISA) and functionality and performance. The example of a heterogeneous multi-core architecture is the Cell BE architecture, jointly developed by IBM, Sony and Toshiba [66] and used in areas such as gaming devices and computers targeting high performance computing. A homogeneous architecture with shared global memory is undoubtedly easier to program for parallelism—that is when a program can make use of the whole core—than a heterogeneous architecture where the cores do not have the same instruction set. On the other hand, in the case of an application which naturally lends itself to be partitioned into long-lived threads of control with little or regular communication, it makes sense to manually put the partitions onto cores that are specialized for that specific task.

Secondly, internally the organization of the cores can show major differences. Most high-performance designs also have cores with speculative dynamic instruction scheduling done in hardware [160]. These techniques increase the average number of instructions executed per clock cycle. However, they are of less importance with modern multi-core architectures because of limited instruction-level parallelism (ILP) in legacy applications. Besides, these techniques tend to be both complicated and power-hungry as well as taking up valuable silicon real estate. In some cases these techniques might not compensate the hardware investment. In fact, some recent architectures such as Intel's Knights Corner [157] have reverted to simple in-order cores augmented with specialized accelerators, in an attempt to reduce the silicon footprint and power consumption of each core. In the next subsection, we highlight the issues in the era of many-core processor.

1.1.2 Issues in the Era of Many-core Processor

So far in this chapter we looked at the current well established technologies used in building multi-core processors as well as at the emerging technologies that are targeting some of the shortcomings of current technologies. On fabrication front, major chip manufacturers estimate that it is possible to reach approximately 6 nm using the current CMOS technologies. As the size of transistors will be measured in just a few tens of atoms at most, quantum effects will have to be taken into account and consequently we will see an increased unreliability of the hardware, with components failing more often and more importantly intermittently. The unreliability of future hardware will likely lead to the implementation of redundant execution mechanisms. Multiple cores could be configuration to perform the same computation, in order to increase the probability that at least one will succeed; in some cases a voting scheme on the result (verifying if all the computations yielded the same result) may be used to guarantee correctness of the calculation. Such mechanisms will likely be invisible to the software, but will impact the complexity of the design.

Another issue is the scalability bottleneck in future many-core processor.

- Scalability bottleneck issue related to design of cache coherency protocols [148]. Synchronizing access to the same memory area from large number of cores will increase the complexity of coherency design and will lead to increasing delay and latency in accessing frequently modified memory areas. In our view, it will be an uphill battle to maintain a coherent view across hundreds, let alone a thousand cores; even if we will be able to do this, it will be hard to justify the cost associated with it.
- Memory bandwidth will be another scalability bottleneck [135]. The leveling out of the core frequency will lead to reduced latency, but the increase in the number of cores will multiply the amount of data required by the cores and thus the aggregate memory bandwidth that future chips will require. If we will indeed see the continuation of Moore's law, translated into an ever-increasing number of cores—perhaps following the same trend of doubling core count every two years—a similar trend would need to be followed by memory bandwidth, something the industry failed to deliver in the past.

Based on these observed bottlenecks, the following trends will dominate the design of future many-core processors:

- Shift towards simple, low frequency, low complexity cores, coupled with an increase of the core count to the level of several hundreds within five to ten years; heterogeneity—not in ISA, but rather in core capabilities—will play a role simply because it is an easy optimization gain.
- Focus on novel memory technologies that can deliver higher bandwidth access; technologies such as 3D-stacking [135] and optical interconnects [19] will see an accelerated uptake by mainstream processor designs; especially optical interconnects have the potential of easing some of the pressure on how chips are structured and interconnected.
- The size of on-chip memory will also see a dramatic increase and we will see innovations emerging that will reduce the footprint, power consumption and complexity of designing such solutions, similar to the development of the embedded DRAM technology; once again 3D-stacking may be one of the technologies to watch.
- HW accelerators will be abundant—these have low footprint and low power consumption, thus we will see realizations in HW of an increasing array of algorithms.
- Aggressive power optimization mechanisms—near threshold operation [45], full or selective power gating [109], dynamic frequency and voltage scaling [104]—will be pervasive not only in traditionally low power domains, but also in most areas where processors are used.

Some of these observations/predictions may fade away, but, in the absence of a revolutionary new method of designing processors, increasing core count, heterogeneity and reliance on aggressive power optimization methods will likely dominate the chip industry for the coming five to ten years.

1.2 Power/Energy Issues

Power has become a vital design goal in recent years within the computer architecture and design community. Previously, low-power circuit, chip, and system design was considered the purview of specialized communities, but this is no longer the case, as even high-performance chip manufacturers can be blindsided by power dissipation problems.

1.2.1 Power Dissipation in Processors

Power dissipation in CMOS circuits arises from two different mechanisms:

- Dynamic power—largely the result of switching capacitive loads between two different voltage states.
- Static/Leakage power—caused by the transistor not completely turning off.

Dynamic power is dependent on frequency of circuit activity, while *static power* is independent of the frequency of activity and exists whenever the chip is powered on. When CMOS circuits were first used, one of their main advantages was the negligible leakage of current flowing with the gate at DC or steady state. Practically all of the power consumed by CMOS gates was due to dynamic power consumed during the transition of the gate. In charging a load capacitor C up ΔV volts and discharging it to its original voltage:

- A gate pulls an amount of current equal to $C \cdot \Delta V$ from the V_{dd} supply to charge up the capacitor and then sinks this charge to ground discharging the node.
- At the end of a charge/discharge cycle, the gate/capacitor combination has moved $C \cdot \Delta V$ of charge from V_{dd} to ground, which uses an amount of energy equal to $C \cdot \Delta V \cdot V_{dd}$ that is independent of the cycle time.

The average dynamic power of the node, i.e. the average rate of its energy consumption, is given by the following equation [38]:

$$P_{dynamic} = C \cdot \Delta V \cdot V_{dd} \cdot \alpha \cdot f \tag{1.1}$$

where, f is the clock frequency and α is the expected activity ratio of the node or circuit array.

The dynamic power for the whole chip is the sum of this equation over all nodes in the circuit. Using Equation 1.1 we know the factors which can reduce the dynamic power of a system. We can either reduce the capacitance being switched, the voltage swing, the power supply voltage, the activity ratio, or the operating frequency. Most of these options are available to a designer at the architecture level. For a specific chip, the voltage swing ΔV is usually proportional to V_{dd}, so Equation 1.1 is often simplified to the following:

$$P_{dynamic} = C \cdot V_{dd}^2 \cdot \alpha \cdot f \tag{1.2}$$

Static power or leakage power is due to our inability to completely turn off the transistor, which leaks current in the sub-threshold operating region [165]. The gate couples to the active channel mainly through the gate oxide capacitance, but there are other capacitances in a transistor that couple the gate to a "fixed charge" (charge which cannot move) present in the bulk and not associated with current flow [136, 137]. If these extra capacitances are large (note that they increase with each process generation as physical dimensions shrink), then changing the gate bias merely alters the densities of the fixed charge and will not turn the channel off. Static/leakage power is proportional to V_{dd}. For a particular process technology, the per-device leakage power is given as follows [32]:

$$P_{static} = I_{leakage} \cdot V_{dd}^2 \tag{1.3}$$

Static energy is the product of static power times the duration of operation. It is clear from Equation 1.3 what can be done to reduce the leakage power dissipation of a system:

- Reduce leakage current $I_{leakage}$.
- Reduce the power supply voltage V_{dd}.

Both options are available to a designer at the architecture level. To lower leakage power and maintain device operation, voltage levels are set according to the silicon bandgap and intrinsic built-in potentials, in spite of the conventional scaling algorithm. The rate at which physical dimensions such as gate length and gate oxide thickness have been reduced is faster than for other parameters, especially voltage, resulting in higher power densities on the chip surface. Thus, power densities are increasing dramatically for next-generation chips.

The problem of power and heat dissipation now extends to the DRAM system, which traditionally has exhibited low power densities and low costs. Due to repeated high-speed access to DRAMs, high-performance DRAMs are dynamically throttled if their operating temperatures surpass design thresholds [113].

1.2.2 Schemes for Reducing Power and Energy

There are numerous mechanisms in the literature that attack the power dissipation and/or energy consumption problem. Here, we will briefly describe techniques such as dynamic voltage frequency scaling and power down unused hardware blocks for reducing leakage power.

Dynamic Voltage Frequency Scaling

Recall that total energy is the sum of switching energy and leakage energy, which, to a first approximation, is equal to the following:

$$E_{tot} = [(C_{tot} \cdot V_{dd}^2 \cdot \alpha \cdot f) + (N_{tot} \cdot I_{leakage} \cdot V_{dd})] \cdot T \tag{1.4}$$

where, C_{tot} is the total capacitance, V_{dd} is the supply voltage, α is the activity ratio, f is clock frequency, N_{tot} is the total number of devices leaking current, $I_{leakage}$ is leakage current and T is the computation time.

Variations in processor utilization affect the amount of switching activity (the activity ratio α). However, a light workload produces an idle processor that wastes clock cycles and energy because the clock signal continues propagating and the operating voltage remains the same. However, none of the approaches affects $C_{tot} \cdot V_{dd}^2$ for the actual computation or substantially reduces the energy lost to leakage current. Instead, reducing the supply voltage V_{dd} in conjunction with the frequency f achieves savings in switching energy and reduces leakage energy. For high-speed digital CMOS, a reduction in supply voltage increases the circuit delay as shown by the following equation [15]:

$$T_d = \frac{C_{totN} \cdot V_{dd}}{\mu \cdot C_{ox} \cdot (W/L) \cdot (V_{dd} - V_{th})^2} \tag{1.5}$$

where, T_d is the delay or the reciprocal of the frequency f, V_{dd} is the supply voltage, C_{totN} is the total node capacitance, μ is the carrier mobility, C_{ox} is the oxide capacitance, V_{th} is the threshold voltage, and W/L is the width-to-length ratio of the transistors in the circuit.

The threshold voltage V_{th} is closely tied to the problem of leakage power, so it cannot be arbitrarily lowered. Microprocessors typically operate at the maximum speed at which their operating voltage level will allow, so there is not much headroom to arbitrarily lower V_{dd} by itself. However, V_{dd} can be lowered if the clock frequency f is also lowered in the same proportion. This mechanism is called *dynamic voltage frequency scaling (DVFS)* [104] and it appears in nearly every modern microprocessor. Based on task under computation, DVFS sets the microprocessor's frequency to the most appropriate level. Most research quantifies the effect that *DVFS* has on reducing dynamic power dissipation because dynamic power follows V_{dd} in a quadratic relationship. However, lowering V_{dd} also reduces leakage power, which is becoming just as significant as dynamic power.

Note also that even though *DVFS* is commonly applied to microprocessors, it is perfectly well suited to the memory system as well. As a processor's speed is decreased through application of *DVFS*, it requires less speed out of its associated SRAM caches, whose power supply can be scaled to keep pace. This will reduce both the dynamic and the static power dissipation of the memory circuits.

Powering-Down Unused Blocks

A popular mechanism for reducing power is simply to turn off functional blocks that are not needed. This is done at both the circuit level and the chip or I/O device level.

- At the circuit level, the technique is called *clock gating (CG)*. The clock signal to a functional block (e.g., an adder, multiplier, or predictor) passes through a gate, and whenever a control circuit determines that the functional block will be unused for several cycles, the gate halts the clock signal and sends a non-oscillating voltage level to the functional block instead. The latches in the functional block retain their information; do not change their outputs; and, because the data is held constant to the combinational logic in the circuit, do not switch. Note that, the circuits in this instance are still powered up, so they still dissipate static/leakage power; clock gating is a technique that only reduces dynamic power [94].
- *Power gating (PG)* technique aims at reducing static/leakage power along with dynamic power. It relies on introducing sleep transistors which are placed between the logic circuit and the ground, thus creating an intermediate *virtual-gnd*. For example, in caches, unused blocks can be powered down using $Gated - V_{dd}$ or Gated-ground techniques [143]. $Gated - V_{dd}$ puts the power supply of the SRAM cell in a series with a transistor. With the stacking effect introduced by this transistor, the leakage current is reduced drastically. This technique benefits from having both low-leakage current and a simpler fabrication process requirement since only a single threshold voltage is conceptually required. At the device level, for instance in SRAM array or DRAM chips, the mechanism puts the device into a low-activity, low-voltage, and/or low-frequency mode such as sleep or doze. For example, microprocessors can dissipate anywhere from a fraction of a watt to over 100 W of power; when not in use, they can be put into a low-power sleep or doze mode that consumes few watts to milli-watts.

Drowsy technique [102] is an alternative technique to reduce static/leakage power in SRAM arrays. This is similar to $Gated - V_{dd}$ and gated-ground techniques in that it uses a transistor to conditionally enable the power supply to a given part of the SRAM. The difference is that this technique puts infrequently accessed parts of the SRAM into a *state-preserving*, low-power mode unlike *clock gating*. A second power supply with a lower voltage than the regular supply provides power to memory cells in the *drowsy* mode. Leakage power is effectively reduced because of its dependence on the value of the power supply.

1.3 Problem Statement

Regardless of the internal hardware implemented on a chip, it is extremely improbable that all of its resources will be simultaneously used due to program dependences and/or thread's synchronization. Moreover, designing the thermal envelop for this "worst case" is somewhat expensive and not very efficient. Computer architects, instead of designing the processor for the worst case scenario, look for the average case, and face these "special" cases by using both power saving and thermal management techniques. Dynamic thermal management aims at reducing the processor power consumption (and performance) during time intervals so it can cool down. One way to achieve this goal is to set a power budget to the processor.

This processor's power budget is not only useful to control power and temperature but also in other situations. For instance, there could be external limitations on the power consumption independent of the microprocessor (or even the system) that needs to satisfied, without shutting off the whole system. There are also situations where device power constraints are more restrictive than the power needs of a processor at full speed. In most of the cases we cannot afford to design a new processor to meet whatever power constraint because it is too expensive. The problem increases when, as usual, power constraints are transitory and after some period of time we want all the processor's performance back. Imagine a computation cluster connected to one or more UPS units to protect from power failures. If there is a power cut, all processors will continue working at full speed consuming all of the UPS battery quickly, and then switching the computers off when the battery is close to running out and, consequently, losing all the work on fly. During the power failure (many times they are of limited duration), if the processors are not doing critical work, it might be more interesting to extend the UPS battery duration at the expense of degrading some performance, than to lose all the work done because the battery runs out.

Another example where setting a power budget could be useful is the case of a computing center that shares a power supply among all kind of electric devices (i.e., computers, lights, air conditioning, etc.). In a worst case scenario (e.g., in summer at mid-day with all the computers working at full speed), if we integrate some kind of power budget management into the processors, during critical day hours or conditions when the air conditioning is consuming a big part of the total power of the computing center, we could decrease the power of all processors, lowering the ambient temperature and having more power for the air conditioning. In this way, we could design the power capacity of the computing center for the average case, reducing its cost.

As described, the future of many-core processor design not only faces the *power-wall* challenges but also need to feed instructions and data to many-tens of cores with reasonable latency (*memory-wall*). The thesis addresses the issues described below:

1.3.1 Scalable Power Management on Many-core Processors

In context of power management in multi-core and many-core processors, there exist a number of mechanisms, namely dynamic voltage and frequency scaling (DVFS) [42, 80, 167], core microarchitecture adaptation [12, 49, 59, 139] and cache adaptation [4, 123, 178]. These mechanisms are quite effective at managing power in isolation at high to moderate power budgets. However, under constrained conditions [109, 110], the existing power management schemes have to revert to power gating the cores. Ideally on a many-core processor, the power manager should provide an integrated way to provide multiple adaptation mechanism like adapting the cores' microarchitecture, cache configuration, *and* DVFS setting in an integrated way. Moreover, the power management needs to be scalable to large core counts and provide adaptations at fine time scale granularities. Finally, the power management needs to be flexible enough to adapt to various optimization targets or metrics, such as maximize performance within a given power budget, or minimize overall energy consumption.

1.3.2 Die-stacked Gigascale DRAM Cache Latency

Memory bandwidth is not keeping up with the execution bandwidth as memory bus bandwidth has not increased in proportion to Moore's Law. In other words, integrating tens of cores onto a single die would be futile if all the cores cannot be kept fed with code and data. Mitigating the memory wall issue is the reason why gigascale on-die DRAM Caches has received considerable industrial and research interest in recent years. Recent approaches address the fundamental issue of high-latency DRAM Cache tag accesses by storing the tags along with the data in the DRAM array. Prior work proposing the Tags-in-DRAM concept include the Loh-Hill Cache [116] and the Alloy Cache [144]. These approaches mitigate the high latency access issue by avoiding DRAM Cache accesses on misses, either by an additional tracking structure on the logic die such as Loh-Hill Cache's MissMap table or through prediction as in the Alloy Cache. Alloy Cache organizes tags and data together to form a single Tag And Data (TAD) entry in a direct-mapped DRAM Cache. The Alloy Cache allows transferring a TAD per request to avoid the tag serialization penalty. Alloy Cache thus optimizes for hit latency instead of hit rate. However, it suffers from a significant amount of conflict misses as core count increases. On the other hand, the Loh-Hill Cache preserves a high level of associativity for its stacked DRAM Cache—29 ways in a single DRAM row buffer to leverage row buffer locality. In the context of many-core processors, the higher associativity of Loh-Hill Cache is likely to be a more advantageous design. Yet reading the tags from DRAM in a separate column access still incurs a significant latency cost. Previously proposed tracking solutions use a centralized entity to either bypass the DRAM Cache on misses, as is the case for the MissMap [116], or store tag information for faster access [60, 75, 181]. Unfortunately, these centralized structures may become a bottleneck with increased core count.

1.4 Thesis Scope

To address the challenges described in the previous section, this thesis focuses on ways to mitigate the issues of *power* and *memory* wall. In the first part of the thesis, the proposed architecture, besides considering performance as its primary objective, is able to account for the pre-configured power budget and adapts the many-core architecture using a model-driven global power manager without incurring significant overheads. It also includes measures to significantly reduce the design complexity of the global power manager on a many-core architectures. In the second part of the proposal, we propose a solution to address the memory-wall issue. The idea and benefits of above ideas will be described in detailed throughout this thesis. The work includes:

- Integrated Power Management approach to many-core processors:
  - Integrated Power Management in Constrained Many-Core Processors — An integrated, scalable, fine-grain, and low-overhead power management method. It use analytical performance and power models to dynamically adapt a many-core processor along multiple axes, including core microarchitecture adaptation, private cache adaptation and per-core DVFS (voltage regulator (VR) per core)
  - Shared Resource Aware Scheduling on Power-Constrained Tiled Many-Core Processors — A two-tier hierarchical power management methodology to exploit per-tile voltage regulators and clustered last-level caches on a tiled many-core processor. It also includes a novel thread migration layer that analyzes threads running on the tiled many-core processor for shared resource sensitivity in tandem with core, cache and frequency adaptation, and co-schedules threads per tile with compatible behavior.
- Mitigating the Memory-Wall by reducing memory latency:
  - Tag Caching for Gigascale Die-Stacked DRAM Caches: A High-Performance Many-Core Perspective — Explore how associativity affects both latency and hit rate of gigascale die-stacked DRAM cache in the context of high-performance many-core processor architectures. The work

proposes on-chip distributed Tag Caching structure that caches DRAM Cache tags, to reduce hit access latency.

1.5 Thesis Layout

The rest of the thesis is organized as follows. Each contribution has its own dedicated chapter. Chapter 2 discusses in detail the background on power management techniques used in the processors. It also describes the previously proposed techniques and discusses some important related concepts that are important to understand the rest of the thesis. Chapter 3 describes in detail the integrated approach to power management on a many-core processors. In Chapter 4, we expand the scheme proposed in Chapter 3 and leverage the tiled-architecture and scheduling in tandem with hierarchical power manager to optimize the overall system performance. In Chapter 5, we show the importance of multi-way near-memory architecture in a many-core processor and propose a tag-caching mechanism to reduce the hit-latency of large die-stacked DRAM cache. Finally, we conclude and provide hints towards future work in Chapter 6.

Chapter 2
Background: Overview and Related Work

In this chapter, we provide a background on micro-architectural power management schemes. First, we describe an overview of the isolated micro-architectural power management techniques. We further describe the related work on these techniques in general to which we will adhere to for the remainder of the thesis. Next, we discuss the combination of these techniques with supported related works. Later in this chapter, we review the concept of "near" memory architecture on future many-core processors. Finally, we will discuss the essentials related works with respect to DRAM-based "near" memory architecture.

2.1 Power Management Schemes

The power wall is currently the main limiter to achieving high performance in modern CPUs, and has been one of the most critical problems facing computer architects over the past several years [105]. Unfortunately, this problem will only get worse in the future as process technologies continue to scale to smaller feature sizes. As such, power efficiency will remain an extremely important design goal, and will require hardware designers to continue efforts to squeeze wasteful power consumption out of architectures.

2.1.1 Isolated Mechanisms

2.1.1.1 Dynamic Voltage Frequency Scaling (DVFS)

As described in Section 1.2.2, the combination of supply voltage and frequency has a cubic impact on total power dissipation because dynamic power consumption has a quadratic dependence on voltage and a linear dependence on frequency. An intelligent power savings solution would reduce operating frequency and, at the same time, reduce the supply voltage. Some example commercial implementations of dynamic voltage frequency scaling (DVFS) technology are Intel's SpeedStep [133] and AMD's PowerNow [41].

DVFS has been applied at both hardware and operating system/platform levels. The main idea is to scale the supply voltage as low as possible for a given frequency while still maintaining correct operation. The voltage can be dropped only up to a certain critical level, beyond which timing faults occur.

Some hardware mechanisms for DVFS implement timing fault detection in hardware itself using special "safe" flip flops that detect timing violations. While DVFS methods are effective in addressing the dynamic power consumption, they are significantly less effective in reducing the leakage power. As minimum feature sizes shrink, supply voltage scaling requires a reduction in the threshold voltage, which results in an exponential increase in leakage current with each new technology generation. It has been shown that the simultaneous use of adaptive body biasing (ABB) and DVFS can be used to reduce power in

high-performance processors. ABB previously was used to control leakage during standby mode, and has the advantage of reducing the leakage current exponentially, whereas dynamic voltage scaling reduces leakage current linearly.

At the operating system level, several OSs now deploy some form of DVFS. For example, Linux uses a very standard infrastructure called *cpu-freq* [28] to implement DVFS. The *cpu-freq* governors provides a modularized set of interfaces to manage the CPU frequency changes through various low-level, CPU-specific mechanisms and high-level system policies. The *cpu-freq* decouples the CPU frequency controlling mechanisms from the policies and helps in their independent development. Many variations of *cpu-freq* governors have been proposed for different kinds of systems, each with their own power and performance requirements.

Work has been done specifically at the handheld/portable/embedded systems level proposing different techniques for implementing DVFS in battery constrained devices. One such example is AutoDVS [69], a system for handheld computers that offers dynamic voltage scaling (DVS). AutoDVS both lowers the amount of energy used and ensures service quality by estimating user interactivity time, think-time and computation load, system-wide and for each program.

A second technique involves application-directed DVFS. It's possible to bypass the difficult problem of trying to get good results using OS level statistics, given that not all applications behave in a predictable way. Instead, by making applications with bursty behavior power-aware, these applications can specify to the scheduler that controls clock speed and processor voltage both their average execution time and a deadline. An energy priority scheduling algorithm arranges the order for these power-aware tasks based on deadlines and the frequency of task overlap.

DVFS for multi-core processors is another interesting and challenging area. One major design decision is whether to apply DVFS at the chip level or at the per-core level. If the per-core DVFS approach is used, more than one power/clock domain per chip is needed and additional circuitry also is required for synchronization among the chips. Although per-core DVFS is more costly to implement than per-chip DVFS for single-chip multiprocessors, per-core has been reported by an academic or commercial entity to have 2.5$\times$ better throughput. The reason is simple: a per-chip approach has to scale down the entire chip even if only a single core is starting to overheat. In contrast, a per-core approach makes only the core with a hot spot scale downward; other cores keep operating quickly unless they develop heating problems.

DVFS is just one of several methods to control dynamic power consumption in CMOS circuits. While its usage brings a set of verification and implementation challenges, we will continue to see its application in future designs at both the hardware and operating system/platform levels.

2.1.1.2 Cache Resizing

A key place to look for power savings is in the on-chip cache hierarchy. Caches occupy a large portion of the CPU's available die area—upwards of 50% in today's CPUs—so they contribute significantly to a processor's overall power dissipation. In addition, in general the caches are sized for the worst case. This means an average computation cannot effectively utilize all of the cache capacity. Such cache over-provisioning can result in significant waste that, if eliminated, can yield large power savings without sacrificing much performance. The trend for modern CPUs is towards deeper cache hierarchies. However, deeper cache hierarchies distributes the power consumption across many caching levels. For dynamic power consumption, the L1 is the greatest culprit, but the L2 and L3 can also consume non-negligible dynamic power, especially for memory-intensive workloads. For static power consumption, the L3 is by far the greatest concern due to its large area. But non-trivial static power can also be dissipated in the L2 as well. By only controlling the size of a single level of cache, existing techniques potentially miss significant opportunities for power savings.

Several researchers have investigated cache resizing techniques [4, 16, 17, 119, 120, 132, 143, 180] to target this form of waste. Cache resizing is an architecture-level power management technique that determines the minimum cache a program needs to run at near-peak performance, and then reconfigures the cache by enabling/disabling cache ways or sets to implement this efficient capacity. Resizing can reduce the amount of cache activated per access, and also enables circuit-level techniques (i.e. Gated-V_{dd} [143]) to shut down unused portions of the cache. This can translate into significant dynamic and static power savings. On the other hand, cache resizing can lead to the increase in cache miss rate, resulting in higher power dissipation for the next level of cache and degraded overall performance. Thus, techniques must achieve a balance between these conflicting factors in order to achieve a net power efficiency gain. Although there has been significant work on cache resizing, existing techniques are limited in their optimization scope. In particular, most studies consider resizing a single level of cache only [4, 119, 120, 132, 143, 180], typically the L1 cache.

The current lack of comprehensive cache resizing is partly due to the availability of other power management options, especially for caches below the L1. Because these caches are only referenced on an L1 miss, CPU performance is somewhat insensitive to their actual delay. Hence, it is feasible to trade off delay for power in the post-L1 caches. This has been exploited extensively by circuit-level techniques to mitigate static power consumption. In particular, multiple V_{th} deices [13, 101], adaptive body bias (ABB) [99, 132], and dynamic voltage scaling (DVS) [58, 102] all convert modest increases in cache access latency into significant static power reductions. While extremely effective, circuit-level techniques for mitigating static power do not obviate the need for architectural approaches like cache resizing. Circuit mitigation only reduces leakage current. In contrast, cache resizing (plus power gating) can suppress leakage practically to zero for the gated portions of cache. Moreover, circuit- and architecture-level approaches are orthogonal. So, applying them in concert may ultimately yield the greatest static power savings.

In addition to flexibility for reducing static power, the low latency sensitivity of post-L1 caches also offer alternatives for reducing dynamic power. For example, serializing tag and data access ensures only a single data way is energized regardless of the number of total active ways, thus reducing dynamic power at the expense of some increased delay. But again, this does not preclude cache resizing. A serial cache still incurs wasteful tag energy as well as significant interconnect energy that resizing can address. And in some cases, serial caches may be too slow limiting their application.

Selective cache ways [4] used off-line profiling to drive disabling of cache ways for dynamic power savings. DRI caches [143, 178] used cache-miss counts to detect over-provisioning, and resize across either cache sets or ways. In addition, DRI caches also gate the power supply to unused portions of cache, conserving both dynamic and static power. Madan et al. [119] proposed resizing of L2 caches by dynamically extending their capacity into stacked DRAM. Balasubramonian et al. [16, 17] proposed resizing of two levels of cache, either the L1/L2 or the L2/L3, by partitioning a common pool of SRAM arrays to different caching levels. Because partitioning always utilize all of the available SRAM, only one cache's size is controlled independently.

Besides resizing, researchers have studied other adaptive cache techniques as well. Dropsho et al. [46] proposed accounting caches which divides cache's ways into primary and secondary groups. Each cache access searched the two groups sequentially, accessing the secondary only on a primary miss. This saved power if secondary accesses are infrequent. Zhang et al. [180] proposed way concatenation which permits flexible organization of cache banks to form direct-mapped, 2-way, or 4-way set-associative caches.

Silva-Filho et al. [155] and Gordon-Ross et al. [64] studied design-time techniques for optimizing 2-level cache hierarchies. These works tried to find the best block size and associativity—as well as cache capacity—for two caching levels. They considered a complex design space and employ more costly search techniques that are suitable for design analysis only. Similarly, Zhang et al. [180] searched for the best cache architecture using a reconfigurable hardware platform, but only optimization at a single cache level.

Cache partitioning explicitly allocates shared cache across multiprogrammed workloads, providing cache to those programs that can best utilize it. The majority of techniques focused on performance [39, 103, 145, 162, 171]. More recently, techniques have also tried to reduce power consumption [97, 164] by withholding allocation and shutting down portions of the shared cache, similar to cache resizing.

Finally, significant research has explored circuit-level techniques for reducing a cache's static power consumption. Multi-V_{th} techniques [13, 101] employ low-V_{th} devices along critical paths and high-V_{th} devices along non-critical paths to save power while still maintaining performance. Gated-V_{dd} [143] uses high-V_{th} devices to gate the power supply to unused portions of cache. Adaptive body bias [99, 132] controls the backgate voltage to place devices in a standby low-leakage mode when not in use, but then restores the devices to an active high-performance mode when the cache is accessed. Lastly, dynamic voltage scaling [58, 102] can similarly transition between standby and active modes by scaling the supply voltage.

2.1.1.3 Core Resizing

In the past, single-thread performance increased according to Pollack's law [141], which states that the performance improves in proportion to the square-root of the processor area. For the past few years, however, this has no longer been the case, and the performance improvement rate has slowed dramatically, despite ever increasing transistor budgets. One of the reasons for this lack of performance improvement is the memory wall, which is the large speed discrepancy between the processors and the main memory. This severely limits the performance of a computer, because of the long latency in a load if a last-level cache (LLC) miss occurs. Conventional solutions to this problem have involved incorporation of a large cache and a hardware prefetcher [11, 93]. Unfortunately, a large cache is very expensive and in many cases, several megabytes of caches are still insufficient.

Aggressive out-of-order execution is an effective alternative approach to this problem. This method significantly increases the number of in-flight instructions that are supported by the processor through extensive instruction window resources (i.e., reorder buffer (ROB), issue queue (IQ), and load/store queue (LSQ). This allows parallel memory accesses by executing the cache-miss-causing loads as early as possible, and thereby reducing the effective memory latency. This type of parallelism is called memory-level parallelism (MLP). One advantage of this approach is that data fetch is carried out by instruction execution, and not by prediction like a hardware prefetch, and thus the data fetch is accurate. Another advantage is that it can be implemented by applying a simple extension to a conventional superscalar processor. However, the downside is that the large resources adversely affect the clock cycle time. Although this can be solved by pipelining the resources, it prevents instruction-level parallelism (ILP) from being exploited effectively, mainly because of the enlarged IQ. Specifically, pipelining of the IQ makes it impossible to issue dependent instructions back-to-back, because the wakeup-select issue loop takes more than a single cycle to complete. As explained above, there is a trade-off involved in enlarging and pipelining the window resources for exploitation of ILP and MLP. In other words, large pipelined resources are effective for exploiting MLP, and are thus beneficial for memory-intensive programs or execution phases. However, they are harmful when exploiting ILP, which offers high performance in compute-intensive programs or phases.

Apart from performance, power dissipation constitutes as the other primary design objective in aggressive out-of-order processors in the general-purpose, high performance segments [30, 68, 70]. In the last decade, high-end processors like the dual-core IBM POWER 4™ [166] were performance-driven designs where overall power densities [26] were still below acceptable limits, even though the net chip power was well over 100 Watts [8]. However, as reported by Bose et al. [26], localized hot spots in regions like the out-of-order issue queues experienced un-gated power densities as high at 70 watts/cm2. Depending on the affected area and relevant thermal time constants, such a localized hot spot can have a significant impact on the packaging/cooling cost of the chip. Sustained periods of temperature elevation within such

a hot spot can also degrade chip reliability.

Several authors [18, 95, 121, 169] have pointed out that the instruction delivery power is higher than necessary because of the performance-focused design strategy. In such designs, the front-end fetch mechanism provides instructions using the peak designed bandwidth, as early as possible, by making use of sophisticated branch prediction algorithms. This strategy often wastes energy because instructions are frequently fetched earlier than necessary. These instructions spend many needless cycles in the issue queue waiting for dependencies to be resolved (or to be aborted following a misprediction event).

Much of the idle energy waste in the front-end is attributable to incorrect control flow speculations. This can be reduced by using more accurate branch prediction schemes or by using confidence estimation to control fetch-gating [121]. However, reducing misspeculated fetches alone would not necessarily reduce the large component of idle energy that results from earlier-than-needed fetch of instructions in the correct path.

Other methods of fetch gating [18, 95] attempted to reduce idle energy by making the fetch mechanism more demand-driven; that is, instruction fetch was gated when the down-stream utilization is high or the flow rate mismatch (between decode and commit) was high. While fetch gating may provide a level of issue queue utilization appropriate for the application, unused entries will still consume energy. Dynamic adaptation of the issue queue [33, 34, 46, 59, 142], is another technique for saving energy in an underutilized issue queue. In this approach, the issue queue is sized to match its level of utilization or the necessary instruction window demanded by the application. Thus, unnecessary entries are shut down, saving considerable energy. The combination of issue-aware fetch gating and dynamic issue queue adaptation, in which the queue is appropriately utilized and unused entries consume negligible energy, has the potential for both good overall chip and issue queue energy savings.

Manne et al. [121] saved the wasted energy used for fetching, decoding, issuing, and executing instructions along mispredicted paths. They estimated the confidence of every branch prediction when that branch was fetched [82]. Karkhanis et al. [95] proposed to dynamically change the number of in-flight instructions. Unsal et al. [169] developed a compiler-driven static IPC estimation scheme that is based on dependence testing in the compiler back-end. This estimation was used to drive fine-grained fetch-throttling energy saving heuristics. However, dynamic factors such as cache misses and branch mis-predictions can dilute the efficiency of these static IPC-estimation-based heuristics.

Buyuktosunoglu et al [33, 34] proposed to resize the issue queue based on its utilization. Marculescu [122] proposed a mechanism to dynamically adapt the fetch and execution bandwidth based on profiling at the basic-block level. Ponomarev et al. [142] and Dropsho et al. [46] proposed to dynamic allocation of multiple resources, including the issue queue, for low-power. Buyuktosunoglu et al [35] also explored the combination of issue queue adaptation and fetch gating scheme (PAUTI) that attempted to match the size of the instruction window resident in the issue queue to application ILP characteristics, while keeping the utilization of the queue high enough to avoid back-end starvation

Albonesi et al. [5] presented a comprehensive survey of studies on resource resizing to improve power efficiency. Dynamic adaptation of the issue queue size to match application demands is proposed in [2, 3] in order to increase performance and reduce power dissipation. However, it was assumed that the best issue queue size for a given application was known a priori; no attempt was made to adapt within an individual application, and the circuit-level design issues associated with an adaptive issue queue were not addressed in detail. Folegnani et al. [59] proposed a resizing scheme for the IQ that deactivates those parts that contribute little to the performance. Petoumenos et al. [138] proposed an IQ resizing scheme that takes into account MLP. Their scheme basically uses an ILP-aware resizing scheme like that proposed in [59], but its resizing decision is overridden by their proposed MLP-aware Brekelbaum et al. [27], proposed a hierarchical IQ with a large pipelined queue and a small non-pipelined queue. Kora et al. [106] proposed an adaptive dynamic instruction window resizing scheme that enlarges and pipelines the window resources only when MLP is exploitable, and shrinks and de-pipelines the resources when ILP

is exploitable. The proposed scheme changes the size of the window resources by predicting whether MLP was exploitable based on the occurrence of last-level cache misses. Abella and González [1] proposed a mechanism that takes resizing decisions based on the time that instructions spend in both the issue queue and the reorder buffer.

2.1.2 Co-ordinated Approaches

To control power and temperature of processors, there are several proposals that try to merge both DVFS and micro-architectural techniques in a two-level mechanism to benefit from both coarse and fine-grained mechanisms.

Eckert et al. [51] proposed new mechanisms to implement P-states that achieve a power-performance curve closer to DVFS than frequency scaling. The mechanisms leverage previously proposed low-power techniques like front-end and cache resizing that result in a performance loss. Previously, these techniques might have been dismissed due to the effectiveness of DVFS; however, they become viable alternatives to frequency scaling. The proposal focus on techniques that selectively disable or constrain micro-architectural performance optimizations, trading performance for power reductions. By using coordinated DVFS with resizing techniques, power savings can be much higher than using isolated techniques like those described in Section 2.1.1.

Deng et al. proposed a framework that searches the space of per-core and memory frequency settings—voltage values set according to the selected frequencies—in operating system (OS), called CoScale [42]. CoScale framework used an epoch-based (epoch corresponds to an OS time quantum) policy estimates, via performance counters and online models, the energy and performance cost/benefit of altering each component's (or set of components') DVFS state by one step, and iterates to greedily select a new frequency combination for cores and memory. The selected combination trades-off core and memory scaling to minimize full-system energy while respecting a user-defined performance degradation bound.

Ghasemi and Kim, proposed resource and core scaling (RCS) [62] technique that jointly scales the resources of a processor and the number of operating cores to maximize the performance of power-constrained multi-core processors. The authors proposed to uniformly scale the resources that are both associated with each core (e.g., L1 caches and execution units (EUs)) and shared by all the cores (e.g., last-level cache (LLC)) as a means to compensate for lack of a V/F scaling range. Under the maximum power constraint, the RCS technique proposed disabling of resources to increase the number of operating cores, and vice versa. The technique proposed a runtime system that predict the best RCS configuration for a given application and adapt the processor configuration accordingly at runtime. The runtime system only needs to examine a small fraction of runtime to predict the best RCS configuration with accuracy well over 90%, whereas the runtime overhead of prediction and adaptation is small. The framework also proposed to selectively scale the resources in RCS (dubbed sRCS) depending on application's characteristics and demonstrate that sRCS can offer 6% higher geometric-mean performance than RCS that uniformly scales the resources.

Meng et al. [123] proposed an adaptive, multi-optimization power saving strategy for multi-core power management with an aim to meet global chip-wide power budget through run-time adaptation of highly configurable processor cores. The authors proposed analytic modeling to reduce exploration time and decrease the reliance on trial-and-error methods. In addition, the framework uses a risk evaluation to balance the benefit of various power saving optimizations versus the potential performance loss along with a mechanism that integrates multiple power optimizations and globally manages chip multiprocessor processor power consumption to honor a chosen power budget. We believe this is timely given the prominence of multi-core processors and growing interest in run-time optimization. Our approach addresses concerns related to the large search space in a CMP system with many core-level optimizations, complex relationships between these optimizations, and transient resource demands due to very

short-lived global phases.

Sharifi et al. [154], proposed PEPON, a two-level power budget distribution mechanism, especially targeted for NoC-based multi-core processors. At the first level, PEPON distributes the overall power budget of the multi-core system among on-chip resources like the cores, caches, and NoC. In the second level, the power budget assigned to cores is further partitioned among individual cores and, similarly, the power budget assigned to L2 caches is further partitioned across individual L2 caches. Both these distributions are oriented towards maximizing workload performance without exceeding the specified power budget—chip-wide power cap. The authors proposed to employ a different power distribution strategy at each level. For the first-level distribution, a regression-based performance model is adopted that decides the most performance-efficient distribution of power. For the second-level distribution, the PEPON framework uses a different strategies for caches and cores. The power budget assigned to cores is distributed using a control theoretic approach and is implemented using multiple DVFS adaptations. On the other hand, for power budget distribution across different L2 banks (selective-way resizing) using utility-based model. In nutshell, PEPON uses 10 DVFS adaptations for cores and NoC along with selective-way resizing for L2 cache to provide feasible working configuration for high-moderate power budget constraints.

Sasanka et al. [152] proposed the use of DVS and micro-architectural techniques to specifically reduce the power consumption in real time video applications. The authors selected micro-architectural techniques try to reduce the power of functional units and the instruction window. Winter et al. [174] also proposed the use of a two-level approach that merges DVFS and thread migration to reduce temperature in SMT processors.

2.2 Near-Memory Architecture

The “memory wall” problem is pretty straightforward, and it’s by no means new to the multi-core era. The problem arises when the execution bandwidth (i.e., aggregate instructions per second, either per-thread or across multiple threads and programs) available in a single socket is constrained by the amount of memory bandwidth available to that socket. As execution bandwidth increases, either because clock-speeds get faster or because the die contains more cores, memory bandwidth has to increase in order to keep up.

But memory bandwidth is not keeping up. Memory bus bandwidth (latency and/or throughput) has not increased quickly enough in proportion to Moore’s Law, a fact that leaves processors starving for bytes. In this respect, the “memory wall” is a classic producer/consumer problem, and it’s the reason that on-die cache sizes have ballooned in recent years. As the memory wall gets higher and higher, it takes more and more cache to get you over it. At this point, it would be fair to say that most modern server processors are really high-speed memories with some processor core stuck on the die, instead of vice versa.

A suitable replacement for the hard-working, but aging synchronous dynamic random-access memory (SDRAM) standard has been a long time coming. While the current DDR3 memory standards—as well as offshoots like GDDR5—have been serving the CPU and GPU well, they’re starting to show signs of being based on early-’90s technology. Essentially, each revision of SDRAM makes use of the same double data rate (DDR) principle as the original technology, which syncs memory to a system bus (allowing it to queue up one process while waiting for another), and also transfers data on both the rise and fall of the clock signal in order to work twice as fast. DDR2 further refined this idea by running its internal clock at half the speed of the data bus. This trick not only allowed it to produce a total of four data transfers per internal clock cycle (effectively running twice as fast as DDR), but the slower clock speed also reduced the voltage requirement to 1.8 V. DDR3 halved the internal clock again, resulting in a quadrupled clock signal for even faster performance and increased transfer rates of up to 17 GB/s per module, and 1.5 V of operating voltage. The DDR4 memory makes use of a new bus, higher clock speeds, and denser chips in order to reach its maximum transfer rate of 25.6 GB/s per module, and a lower 1.2 V of operating voltage.

DDR4's lower power consumption and higher density will benefit mobile devices and server farms, but from a performance point of view, there's not much to write home about. While there's long been talk of hitting the so-called "memory wall" at around 16 CPU cores, that's not currently much of an issue for the desktop, where a lower latency is currently preferred over a high-bandwidth solution. In the GPU world, though, bandwidth is king, which is why you often see another type of SDRAM in graphics card

Increasing the speed and/or bandwidth of the memory component is one of the solutions to mitigate the memory wall issue. In addition, integrating a memory array onto the CPU/GPU die would provide an overall reduction in the overall latency. For instance, Intel integrated DRAM onto its CPUs in the form of on-package (not on-die) eDRAM—Crystalwell Iris Pro graphics. Microsoft's Xbox 360 and Nintendo Wii U used a similar setup to supplement system memory. However, eDRAM is still an expensive proposition, and once again, there are significant space limitations on the amount of memory that can be integrated. Still, moving the memory closer to the CPU/GPU does dramatically increase the bandwidth and performance.

3D-stacked memory—new, stacked memory design like Hynix's High Bandwidth Memory (HBM) or Micron's Hyper Memory Cube (HMC)—brings the DRAM as close as possible to the logic die. In a traditional setup, the individual DRAM chips are placed side by side, and connected to the CPU/GPU via long copper traces on a PCB. On the other hand, 3D-stacked memory stacks the memory chips on top of each other, dramatically reducing the overall footprint required, allowing for the use of extremely wide data buses and much slower clock speeds in order to hit the required levels of performance. With stacked memory, instead of buses being hundreds of bits wide, they can be thousands of bits wide, and because everything is so close, the power consumption per transported bit is much lower.

For gigascale near-memory architecture, both bandwidth and latency play a vital role in achieving high performance, which implies that future designs must exploit both opportunities. While today's servers need tens to hundreds of gigabytes of DRAM each, the projections for die-stacked DRAM capacity vary between hundreds of megabytes to several gigabytes [53, 159]. Many research/industrial proposals for die stacking advocate using the stacked DRAM as a cache [90, 98, 114, 115, 116, 175]. Unfortunately, the DRAM Cache designs inherently suffers from limitations—significant tag storage requirement due to their large capacity, whose lookup necessarily adds extra latency to the critical path.

2.2.1 Designing DRAM Caches

To leverage the benefits that the die stacking technology provides as a DRAM cache there are some key design points to be considered:

- Tag lookup and storage: Since the tag lookups are on the critical path of all requests coming to the cache, it is of utmost importance that the tag lookup latency must be minimized. This gains more importance in the context of DRAM caches, due to their tag array size. The total storage dedicated to tags or other metadata should be minimal as does incur high cost—both latency and area.
- Off-chip traffic: While cache misses are responsible for most of the off-chip bandwidth overhead, various cache features can adversely impact off-chip bandwidth even further. Examples include the use of large cache blocks that saturate off-chip bandwidth. Reduction in off-chip traffic is the main driver for 3D-stacked DRAM adoption, and as such should be among the top priority goals.
- Hit and miss latency: The DRAM caches must optimize for both hit and miss latency using micro-architectural techniques to either reduce hit latency or predict/bypass misses.

Moreover, to minimize the stacked DRAM and off-chip DRAM access latencies, cache designs must be take into account the parameters like hit ratio, locality of references, row-buffer management policy, access scheduling and optimal allocation of space for data. To better understand such challenges, there are two main DRAM cache designs that optimize for different constraints: block-based caches and page-based caches.

2.2.2 Block-Based Caches

On-chip caches have traditionally been designed to primarily exploit temporal locality, and to make the best use of their limited capacity. Trade-offs between the effective cache capacity, temporal and spatial locality resulted in 16- to 128-byte cache blocks, 64-byte being the most common block size employed today. For large DRAM caches, 64-byte blocks would require huge tag storage which is infeasible to build in SRAM, thereby forcing the tags to be embedded in DRAM [60, 75, 115, 116, 144, 156, 181]. Embedding tags in DRAM, however, results in multiple DRAM accesses per cache request—and, consequently, in substantially higher hit and miss latencies. Intelligent co-location of data with the corresponding tags in the same DRAM row [116, 144, 156] accompanied with optimized access scheduling, obviates the need for multiple DRAM accesses per request. However, the optimization only partially reduces the high hit latency, because of the need for several operations to be performed within the DRAM row-buffer. Furthermore, the co-location of tags and data mandates particular data placement policies that diminish DRAM locality. It also requires a way to determine the presence of a block in the cache prior to accessing the tags, as well as additional multi-megabyte storage for that purpose, whose access latency is on the critical path. Most importantly, block-based designs fall short of exploiting abundant spatial locality. Instead, they focus on limited temporal locality, experiencing high miss ratios, thus frequently exposing full off-chip latency to incoming requests. However, due to the small fetch unit and the efficient management of cache capacity, block-based designs minimize off-chip traffic, making them a favorable option for high-throughput servers.

2.2.3 Page-Based Caches

Increasing the block size allows for a proportionate reduction in tag storage. The use of larger allocation/fetch units (e.g., 1-8 KB) makes the placement of tags in SRAM feasible at acceptable storage overhead [87, 88, 90]. The large fetch unit allows for maximum DRAM access efficiency, fully exploiting locality in both off-chip and stacked DRAM. For instance, a single DRAM row opening is needed per off-chip DRAM fetch, eviction, or stacked DRAM fill, for a whole page, assuming that the page size does not exceed the DRAM row size. While large DRAM caches exhibit limited temporal locality, they show significant spatial locality, which can be easily leveraged by large fetch units providing an order of magnitude more hits compared to a block-based cache of the same size. Cache hits are critical to exploiting the latency advantages of die-stacked DRAM and page-based caches provide them at lower latency. Unfortunately, many of the cached pages contain data that are not used prior to the page eviction, resulting in excessive data over-fetch and capacity waste. As a result, page-based caches tend to increase the off-chip traffic of the baseline system without a DRAM cache by up to an order of magnitude in the worst case, which negates a key benefit of die-stacked DRAM caches.

Part I

Optimizing Performance under Power Constraints

Chapter 3
Integrated Power Management in Constrained Many-Core Processors

As motivated in previous chapters, modern microprocessors are increasingly power-constrained as a result of slowed supply voltage scaling (end of Dennard scaling) in conjunction with the continuation of transistor density scaling (Moore's Law). Although existing many-core power management techniques such as chip-wide and per-core DVFS, and core and cache adaptation were shown to be quite effective in isolation at moderate to high power budgets, they are unable to match stringent power budgets and hence have to revert to core gating. Overall, existing techniques do not scale well to larger core counts, smaller time scales and more stringent power budgets in future chip technologies.

In this chapter, we present Chrysso, an integrated, scalable, fine-grain, and low-overhead power management method using analytical performance and power models to dynamically adapt a many-core processor along multiple axes, including core microarchitecture adaptation, cache adaptation and per-core DVFS. By integrating multiple power management techniques into a common methodology, Chrysso outperforms isolated mechanisms by significantly improving system performance within a given power budget.

3.1 Introduction

Modern microprocessors are very much power-constrained as a result of two prominent trends in chip technology. Moore's Law [129] refers to the doubling of transistors on chip every 18 months, and has been a fundamental driver of computing. Unfortunately, because of the end of Dennard scaling [43] (slowed supply voltage scaling), we may become so power-constrained that we will no longer be able to power on all transistors at the same time—a problem referred to as *dark silicon* [52]. Moreover, run-time factors such as thermal emergencies [29] and power capping [57] further constrain the available chip power. We thus need to be smart about which transistors are helpful to performance and which are not, and only power on those transistors that actually help us maximize performance within a given power budget at any given time.

There exist a number of mechanisms to manage power, including Dynamic Voltage and Frequency Scaling (DVFS) [42, 80, 167], core microarchitecture adaptation [12, 49, 59, 139], and cache adaptation [4, 123, 178]. Although these mechanisms are quite effective at managing power in isolation at high to moderate power budgets, we need an integrated approach moving forward. Each of the above techniques are limited in scope and are unable to meet stringent power budgets when applied in isolation, and hence have to revert to core gating under constrained conditions [109, 110]. Ideally, we want to adapt the many-core processor by adapting the cores' microarchitecture, cache configuration, *and* DVFS setting in an integrated way. Moreover, we also want power management to be scalable to large core counts and fine time scale granularities. Prior work is either limited to adapting resources in isolation, operates at a

fairly coarse time scale granularity, incurs relatively large run-time overhead, and/or relies on heuristics and/or sampling to search the optimization space.

We propose Chrysso[1], an integrated many-core power management methodology to quickly adapt the processor architecture and settings to workload characteristics and run-time conditions. Chrysso leverages analytical performance models along with table-based power models to explore the complex optimization space comprising core and cache adaptation along with per-core DVFS, at small time scale granularities (e.g., 10 ms) and at large core counts (multiple tens of cores). Chrysso uses run-time statistics of a past time slice, and predicts both core performance and power for a wide range of many-core configurations for the next time slice to identify a configuration that yields (near-) optimal performance within a given power budget. By doing so, Chrysso exploits both inter-workload variability as well as intra-workload phase behavior to optimize energy-efficiency over time under varying workload conditions. To reduce the search space to be explored at run-time, Chrysso uses a two-tier process: it first performs a Pareto-optimal design space exploration for each core/thread, and then performs a global, chip-wide search over these design points. In addition, Chrysso uses utility-based optimization to give a larger fraction of the total power budget to cores/threads that benefit the most, in particular critical threads in multi-threaded workloads and power-hungry programs in multi-program workloads.

We experimentally evaluate the efficacy of Chrysso by simulating a 64-core system with adaptation opportunities inside the cores (changing pipeline width and buffer resources in a balanced way), the caches (selecting ways in the last-level caches), as well as through per-core DVFS. We report that Chrysso outperforms DVFS, core gating, core adaptation, and cache adaptation in isolation by a significant margin over a broad range of power envelops. At stringent power budgets (40% of TDP and less), none of the isolated power management mechanisms are able to find configurations that match the power envelop and thus have to revert to core gating. On the contrary, Chrysso can improve system throughput by 1.9× on average over core gating for SPEC CPU2006 multi-program workloads. For a collection of multi-threaded workloads from NPB, PARSEC and SPEC OMPM2001, Chrysso outperforms core gating by 1.5×. Furthermore, Chrysso improves the average per-thread progress rate by 75% over core gating, and is shown to be able to adapt to time-varying workload and run-time conditions. Chrysso incurs little execution time overhead (less than 1%), while requiring limited hardware overhead as well (roughly 3 KB per core to assist adaptation decisions), which makes it a scalable solution both in time (at small time scales) and space (at large core counts).

We make the following contributions:

- We make the case that power saving techniques such as core and cache adaptation, and per-core DVFS, when employed in isolation, are insufficient to meet stringent power budgets on many-core processors.
- We propose simple, yet effective analytical performance and power models for steering many-core adaptation. The inputs required for these models can be collected at low overhead.
- We propose a global optimization algorithm to quickly and effectively search the adaptation space, and partition the available power budget among cores/threads. Besides leveraging analytical models, the key features of the search algorithm include:
  - identifying per-core Pareto-optimal configurations to prune the search space for global exploration, and
  - utility-driven optimization by budgeting more power to cores/threads that benefit the most.
- We propose and evaluate the Chrysso framework and demonstrate that Chrysso outperforms isolated power saving techniques by a significant margin. Chrysso improves system performance by 1.9× and 1.5× over core gating at stringent power budgets for multi-program and multi-threaded workloads, respectively.

[1] Chrysso is a spider genus whose color is variable. By analogy, Chrysso adapts the many-core processor configuration to variable workload and run-time conditions.

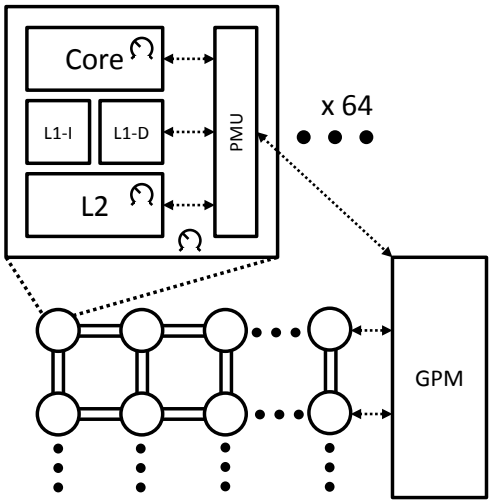


Figure 3.1: Adaptive many-core architecture with Chrysso, featuring per-core power/performance monitoring units (PMU) and a global power manager (GPM).

3.2 Chrysso Overview

3.2.1 Many-core power management

Chrysso is evaluated in the context of a many-core processor, as shown in Figure 3.1. Each core has a number of configuration knobs that together define distinct operating points, each with a different power-performance trade-off. A per-core power/performance monitoring unit (PMU) keeps track of core activity and controls the core configuration in response to requests made by the global power manager (GPM). This global manager combines information from all cores, and performs the global power/performance optimization. By being knowledgeable about differences in per-core behavior, the available power budget can be dynamically distributed across cores.

In commercial designs, both the per-core PMU and global GPM are already present in some form, see for example [149]. The PMU typically collects power consumption and junction temperatures, and performs control functions such as P-state (DVFS) and C-state (various levels of core gating) transitions. The GPM is implemented as an integrated microcontroller and runs firmware algorithms that interface with the PMUs and both on-chip and off-chip voltage regulators.

3.2.2 Chrysso adaptation

Figure 3.2 depicts the Chrysso event flow. Time is divided into fixed-sized time slices, typically several milliseconds. During each time slice, cores keep track of activity statistics using the hardware counters in the per-core PMUs. At the end of a time slice, the PMU uses analytical performance models along with table-based power models to predict/project the performance and power of all possible core configurations (48 configurations in total in our setup). Each core then sends a list of Pareto-optimal core configurations to the GPM, which globally optimizes the many-core configuration within the given power budget. Finally, the GPM instructs each core to adapt itself based on the configuration that was decided upon. Using a 10 ms time slice, we find the Chrysso flow to be very low overhead (less than 1%) of which the projection and optimization parts can be overlapped with workload execution, to reduce overhead even further.

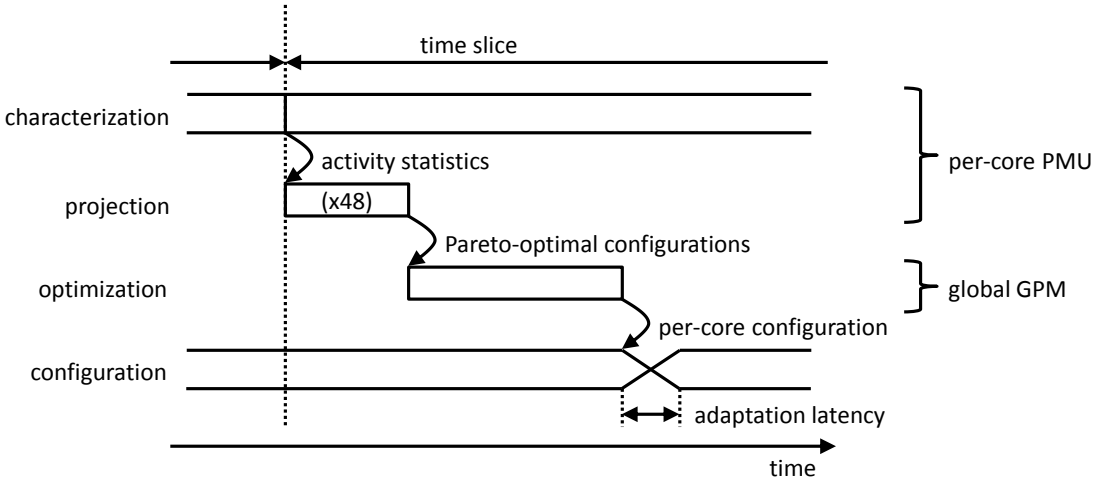


Figure 3.2: Event flow in Chrysso.

3.3 Chrysso Optimization Algorithm

The aim of Chrysso's optimization algorithm is to find a combination of per-core configuration settings that maximizes performance within the imposed chip-wide power budget. The search space is multi-dimensional with local minima that is not trivially navigated. However, by combining per-core Pareto frontier generation with a *utility*-driven power budget allocation, Chrysso can quickly navigate this search space and converge to a (near-)optimal configuration at small time granularities. Chrysso's scalability and adaptivity depends on two key features: *Pareto-optimal search* and *utility-based optimization*.

3.3.1 Per-core prediction and Pareto-optimal search

Chrysso first predicts performance and power for all possible configuration tuples (w, c, f) for each core, with w denoting the core's width, c the number of cache ways enabled, and f the core's frequency. This is done by the PMUs for the respective cores using the projection models which we discuss in detail in Section 3.4.

Once each core's PMU has computed the projected performance and power values for each possible configuration, we then discard all non Pareto-optimal configurations. The Pareto frontier consists of those points for which no other point can at the same time provide higher performance *and* lower power consumption. This significantly reduces the number of configurations that have to be taken into account in the global optimization round: out of a total of 48 per-core configurations, typically only 5 to 10 are Pareto-optimal, depending on the workload characteristics. These points are then sent to the GPM for global optimization.

3.3.2 Utility-driven optimization

The global optimization algorithm uses the current many-core configuration as a starting point. As long as the projected power consumption of the current configuration exceeds the power budget, cores are successively selected to step down their configuration. All settings occur only along each core's Pareto frontier—by using Pareto-optimal points only, we have already linearized the per-core selection process without loss in optimality.

At each iteration, the core that is selected to perform a *step-down* is the one that will provide the highest

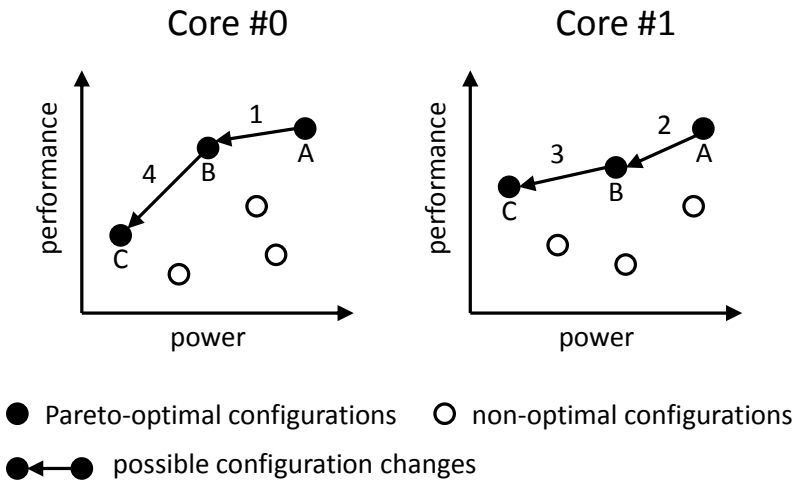


Figure 3.3: The Chrysso algorithm consists of Pareto frontier selection followed by utility-based optimization.

utility—this is the core that is able to achieve the highest reduction in power consumption while giving up the least amount of performance along the core's Pareto frontier. To speed up the algorithm, and to reduce the chance of ending up in local minima, cores can also be selected to *step-down* by multiple steps at once. The algorithm works by constructing a list of down-steps, between 1 and M steps along the Pareto frontier for each core. It then sorts these steps by *utility* (power savings divided by performance loss). The first *step-down* in this list is applied to the current configuration, after which the algorithm performs the next iteration with another *step-down* as long as the predicted power consumption exceeds the available power budget.

Once the power limit has been reached, or if the initial configuration already falls below the power limit, the *step-up* phase of the algorithm starts in which cores can be stepped up. This allows Chrysso to take up any spare power budget that was made available by the last *step-down* phase, if any.

Figure 3.3 illustrates this process: for two cores #0 and #1, it shows the various configurations with the Pareto-optimal design points shown as black dots. Down-steps are selected based on utility, i.e., by the largest reduction in power for the least reduction in performance—which corresponds to the available configuration change with the shallowest slope: Chrysso selects down-steps 1, 2, 3 and 4 for cores #0, #1, #1, and #0, respectively.

3.3.3 Critical-thread aware Chrysso

The utility-driven global optimization strategy as just described allows for giving a larger share of the total power budget to cores/threads that benefit more than others. In a multi-program workload environment this helps overall system performance by giving a relatively larger power budget to compute-intensive applications over memory-intensive applications (as we will quantify in the results section). For multi-threaded workloads, it makes sense to give a larger share of the power budget to critical threads to improve overall application performance. A thread is considered critical at a given point in time if it determines the progress of the whole application. In particular, a serial thread is the critical thread as all other threads wait for the serial thread to finish. Likewise, threads that have reached a barrier wait for the lagging critical thread to also reach this barrier. Similarly, a thread holding a lock is the critical thread while other threads wait for the lock to be released.

Chrysso leverages the notion of thread criticality to allocate power among the cores/threads: it computes

thread criticality as proposed by [47]. The criticality value quantifies how much time a thread is performing useful work (active/running) and how many threads are concurrently waiting within a given time slice. Intuitively, a thread that is actively running while other threads are waiting due to synchronization is more critical and therefore receives a larger criticality value. We make Chrysso critical-thread aware by identifying the thread with the largest criticality value and we prevent the critical thread from entering a *step-down* phase, i.e., we do not want the critical thread to be slowed down. Instead, during the *step-up* phase, we allocate a higher power budget to the critical thread (of course, we remain within the given power budget). By preventing the critical thread from being slowed down, and by even speeding it up (if possible), we achieve a more balanced execution, and hence we reduce overall application execution time. Note that we recompute the criticality values for all threads in each time slice, and we thus dynamically determine the most critical thread and repartition the power budget accordingly. Computing thread criticality can be done in hardware with minimal overhead (65 bits per core [47]) or in the OS kernel [48].

3.3.4 Optimizing for other criteria

While we implement and evaluate Chrysso to optimize performance under a given power budget, it can easily be adapted to other optimization criteria, as long as (i) a per-core Pareto frontier can be constructed to reduce the number of candidate configurations, and (ii) a utility metric can be constructed that allows the effectiveness of different configurations to be compared.

One alternative optimization criterion could be to minimize energy for a given performance threshold, as might be of interest in battery-powered devices with performance guarantees (such as video playback). Here, Pareto points would be constructed using energy and performance, rather than power and performance. Utility of different configurations is compared by computing the slope of energy savings versus performance loss, while the optimization algorithm would *step-up* cores as long as the performance target is not met. In addition, the optimization algorithm can be tuned to skip the *step-down* phase for high-priority applications at the cost of low-priority applications.

3.4 Chrysso Projection Models

As mentioned earlier, Chrysso uses model-based prediction to explore the optimization space at run time. This allows Chrysso to scale much more easily to architectures with multiple configuration knobs, while incurring little run-time overhead. In contrast, sampling-based methods used in prior work [12, 59, 139] have to *measure* power/performance in selected operating points, potentially leading to sampling errors introduced by application phases, interpolation errors due to the non-linearity of the search space, and higher overheads to try out a number of suboptimal configurations. The inputs to Chrysso's prediction models are the activity statistics collected using hardware counters during the current time slice. Assuming workload behavior is relatively constant over time (and at least representative for the next time slice), this information is then used to predict performance and power for all possible core configurations (w, c, f) during the next time slice.

3.4.1 Performance projection

Performance of a candidate configuration is modeled based on the notion of a CPI stack, which breaks down the average number of cycles executed per instruction into individual CPI components representing cycles 'lost' due to branch and memory stalls, in addition to a *base* component.

Knob	Parameter	Values			
Core (w)	Width	1	2	3	4
	ROB size	16	32	64	128
	Reservation station entries	4	8	16	32
	Load queue entries	6	12	24	48
	Store queue entries	4	8	16	32
Cache (c)	L2 cache ways	4	8	12	16
	Capacity (KB)	128	256	384	512
DVFS (f)	Frequency (GHz)	0.8	1.0	1.2	—
	Vdd (V)	0.7	0.75	0.8	—

Table 3.1: Configuration knobs and corresponding architectural parameters and values.

$$CPI = CPI_{core} + CPI_{mem} \tag{3.1}$$
$$CPI_{core} = CPI_{base} + CPI_{branch} \tag{3.2}$$
$$CPI_{mem} = CPI_{L1} + CPI_{L2} + CPI_{dram} \tag{3.3}$$

To take into account frequency changes, we introduce the seconds per instruction (SPI) metric. SPI is a function of CPI and clock frequency f:

$$SPI = CPI/f. \tag{3.4}$$

We use the CPI stack of the $Current\ (C)$ configuration to predict performance of other configurations by rescaling individual CPI stack components. In the discussion to follow, we will compute the performance of a $Target\ (T)$ configuration tuple (w^T, c^T, f^T), given performance information obtained from the previous slice which ran at the $Current\ (C)$ configuration tuple defined by (w^C, c^C, f^C).

As mentioned before, we consider three degrees for adaptation: within the core, within the last-level cache, and through per-core DVFS (see Table 4.1). We now describe in detail the performance models for each of these adaptations.

3.4.1.1 Core knob

The core knob affects the core's execution width and the size (quadratically with core width) of various supporting structures (ROB, reservation station, load/store queues). None of the memory-related components nor the branch predictor are assumed to be affected, so their CPI components are kept constant.

We observe in our experiments that changing the core width along with the respective buffers has a saturating effect on ILP. Using the relation between processor width and ILP, the effect of changing the core knob is predicted by:

$$CPI^T_{base} = CPI^C_{base} \cdot \frac{w^C}{w^T} \tag{3.5}$$

3.4.1.2 Cache knob

Cache projections are based on data obtained from auxiliary tag directories (ATDs) [145], which we use to estimate the cache miss rate would be for each of the possible $Target\ (T)$ configurations. To gauge the performance impact incurred by a change in miss rate, we assume constant DRAM access time and

memory-level parallelism. The CPI_{dram} component can therefore be assumed to scale with the miss rate estimated by the ATDs:

$$CPI^{T}_{dram} = CPI^{C}_{dram} \cdot \frac{ATD(c^{T})}{ATD(c^{C})} \tag{3.6}$$

When the current miss rate $ATD(c^C)$ is zero—for certain application phases in which the working set fits in c^C—we avoid a division by zero by assuming a fixed cost per DRAM access. CPI^T_{dram} is then estimated by multiplying the (uncontended) DRAM access latency with the number of expected LLC misses per 1,000 instructions (MPKI).

3.4.1.3 Frequency and voltage knob

When changing a core's clock frequency and voltage, this affects the behavior of the core itself as well as that of the L1 and L2 caches—which in our architecture are tied to the core clock. The speed of operations taken by the core or caches, when measured in clock cycles, will therefore not change. DRAM access latency, however, will stay constant only when measured in absolute time. Thus, we can predict total core performance as:

$$SPI^{T} = (SPI_{core} + SPI_{L1} + SPI_{L2}) \cdot \frac{f^{C}}{f^{T}} + SPI_{dram} \tag{3.7}$$

3.4.1.4 Putting it all together

To estimate performance (in instructions per second, IPS) for a configuration of interest, Equations 3.5–3.7 are first applied to rescale the performance components of the current configuration to the new configuration; subsequently, application of Equations 3.1–3.4 yields a prediction for overall core performance.

The baseline *IPS* calculated before is rescaled using the ratio of $measured$ IPS versus $estimated$ IPS for the $Current\ (C)$ configuration. This step dynamically corrects the modeling inaccuracies and adapts to application variability. It helps in improving the prediction accuracy for future time slices.

$$IPS^{T}_{estimated-scaled} = IPS^{T} \cdot (\frac{IPS^{C}_{measured}}{IPS^{C}}) \tag{3.8}$$

3.4.1.5 Accuracy versus complexity

Note that for each of these knobs, more elaborate models can be constructed that may reduce modeling error. For instance, by taking average dependency distance into account, the saturating effect of processor width on ILP extraction could potentially be estimated. Likewise, the current model ignores the effect that larger ROBs can expose more MLP; also, the model assumes that memory access latency is not affected by DRAM nor NoC bandwidth. However, more complex models would incur more overhead (which would compromise scalability for many-core system) for both collecting the required statistics and for running the projection models, while not necessarily enabling the reconfiguration algorithm to make better decisions.

In the evaluation section, we will compare reconfiguration decisions based on our models with oracle-based decisions, and show that these simple models provide a good balance between accuracy and complexity; a trade-off opted to improve scalability for many-core systems. Moreover, because we rescale performance and power projections relative to the current configuration's measured performance and power numbers, as just described, good relative accuracy suffices for the purpose of identifying appropriate core configurations during utility-based optimization.

3.4.2 Power projection

Power consumption prediction uses a table-based approach. For each core configuration (w, c, f), the table contains the static power consumption P_{static}, the dynamic energy consumption based on activity costs per instruction (E_{instr}), and the dynamic energy consumption per L2 cache access (E_{L2}). The values in this look-up table are obtained through offline analysis using a broad set of applications. This is a one-time cost and can be done 'at the factory'. Due to process variability, each processor chip might have slightly different table values.

Now, to estimate power consumption for the next time slice, we need an estimate for the dynamic instruction count and cache access count during the next time slice. This is estimated using the following equations. The dynamic instruction count is estimated by multiplying the time slice length (in seconds) with the projected IPS; the cache access count is computed by scaling the dynamic instruction count with the cache access rate of the current time slice.

$$Icount^T = time\ slice \cdot IPS^T_{estimated-scaled} \tag{3.9}$$

$$L2access^T = (L2access^C / Icount^C) \cdot Icount^T \tag{3.10}$$

Computing projected power is done by multiplying the projected instruction and L2 access counts with their respective energy costs, divided by the time slice length, and added to the static power for a given configuration T.

$$P^T_{estimated} = P^T_{static} + \frac{(E^T_{instr} \cdot Icount^T) + (E^T_{L2} \cdot L2access^T)}{time\ slice} \tag{3.11}$$

This estimate is subsequently rescaled using the ratio of $measured$ versus $estimated$ power for the $Current\ (C)$ configuration:

$$P^T_{estimated-scaled} = P^T_{estimated} \cdot (P^C_{measured} / P^C_{estimated}) \tag{3.12}$$

$P^C_{estimated}$ is computed using Equation 3.11 with the table values corresponding to the $Current\ (C)$ configuration, while $P^C_{measured}$ is obtained from hardware energy counters.

3.4.3 Hardware support

Core-level projections rely on hardware support for collecting CPI stacks. On out-of-order cores, hardware collection can be complicated because of various overlap effects between miss events. Recent commercial processors such as the IBM Power5 [124] and current generation Intel processors [77, 179] do have support for computing memory stall components, making most of the required information already available.

Cache projections are based on data obtained from auxiliary tag directories (ATDs) [145], which we use to keep track of what the cache miss rate would be given each of its configuration settings. Since we use selective ways, only a single array of tags has to be maintained per cache set, corresponding to the largest possible configuration. Assuming LRU replacement policy, an access would be a hit when the cache was configured to be n ways *iff* its LRU position is less than N (with 0 being MRU and $N-1$ being LRU in an N-way cache). Set sampling can be employed to reduce hardware overhead with minimal impact on accuracy. In our experiments, we sample 32 randomly selected sets out of 512 sets in the LLC, which incurs an overhead of 2,688 bytes per core[2].

For power projections, we employ simple models that predict the relative difference between the current

[2] 42-bit tags and 16-way maximum associativity.

configuration and other configurations of interest. Input to the models are activity statistics that count the total number of instructions and the number of LLC accesses. In addition, the current power consumption is used as a correction factor, which can be obtained from energy counters as available in current generation Intel processors [149]. The power characterization table itself is populated using data obtained at design time, or could be filled in with per-core specific values after chip fabrication to take into account process variation. The table consists of three 16-bit numbers per configuration; for 48 configurations in total, this amounts to 288 bytes of storage.

In summary, hardware overhead is limited. Either the required information is already available in existing systems; or can be obtained at low cost—less than 3 KB per core (ATDs, power table, and critical thread calculation).

3.5 Experimental Setup

3.5.1 Simulator

3.5.1.1 Performance simulator

We use the Sniper multi-core simulator [36], version 6.0, and added support for dynamically changing core and cache parameters. Core reconfiguration and DVFS transitions take 2 μs [104]. Because voltage regulators have to stabilize, we assume no computations can be performed during this transition. When reducing the number of cache ways, dirty lines are written back through the simulated memory subsystem, consuming NoC and DRAM bandwidth. Since cache reconfiguration is relatively infrequent, these write-backs take up no more than 0.6% of total DRAM bandwidth.

3.5.1.2 Power consumption

McPAT version 1.0 is used to estimate static and dynamic power consumption [112]. Power savings incurred by reconfiguration are modeled by running McPAT with the modified target parameters as per Table 4.1. Running McPAT along with the performance simulation allows us to emulate the behavior of hardware energy counters at simulated time slices of 10 ms.

3.5.1.3 Chip temperature

Temperature values were obtained using HotSpot [76]. A floorplan was generated at core-level granularity, by arranging cores (including their L1 and L2 caches) in a 16×4 layout. Each core has an area of 10.3 mm^2 as estimated by McPAT in the 22 nm technology. For every time slice, per-core power is calculated by McPAT and fed to HotSpot which then computes temperature distribution. We report maximum and average temperature across all cores.

3.5.2 Workloads

3.5.2.1 Multi-program workloads

We run a number of multi-program workloads composed of SPEC CPU2006 benchmarks; there are 29 CPU programs in total, which along with all of their reference inputs leads to 55 benchmarks in total. We select representative simulation points of 750 million instructions each using PinPoints [134]. Four multi-program workloads are constructed by combining these 55 benchmarks as indicated in Table 5.3.

Workload	Description	Benchmarks
WL0	SPEC average	all 55 + 9 uniform random
WL1	Compute	8 compute bound, ×8
WL2	Mixed	8 compute + 8 memory, ×4
WL3	Memory	8 memory bound, ×8

Table 3.2: Multi-programmed workloads.

Suite	Benchmark/Workload	Inputset
NPB	BT, CG, FT, MG, SP, UA	class A
PARSEC	blackscholes, bodytrack, facesim ferret, fluidanimate, swaptions raytrace, canneal, streamcluster	simlarge
SPEC OMPM2001	wupwise_m, swim_m, mgrid_m applu_m, equake_m, apsi_m gafort_m, fma3d_m, ammp_m	reference

Table 3.3: Multi-threaded workloads.

Each benchmark is pinned to a core. We run the simulation for a fixed amount of simulated time (500 ms). When a benchmark completes before this time, it is restarted on the same core. We quantify system throughput using the STP metric [54] (also called weighted speedup [158]) which quantifies the aggregate throughput achieved by all cores in the system.

3.5.2.2 Multi-threaded Workloads

We use multi-threaded benchmarks from PARSEC [21], SPEC OMPM2001 [10] and NAS Parallel Benchmarks (NPB) [14], as described in Table 3.3. As realistic working sets are required to make the analysis meaningful, we use the *simlarge* input set for PARSEC, the *reference* input set for SPEC OMPM2001, and the *class A* input set for NPB (with reduced number of iterations to keep simulation time manageable). Each benchmark is executed with 64 threads in our 64-core processor. Each thread is pinned to a core. We run each benchmark until completion and report total execution time.

3.5.3 Adaptive many-core architecture

The architecture on which we evaluate Chrysso is a large 64-core processor; see Table 3.4 for more details. The configuration knobs for *core adaptation*, *cache adaptation* and *DVFS* (Table 4.1) are downscaled versions of the baseline architecture.

We define the chip's maximum thermal design power (TDP) using the average power consumption of a full-feature chip (each core at maximum width, maximum number of cache ways, and highest frequency/voltage setting) while running the average SPEC workload (WL0), which was 120 W. Results will be shown for power budgets as a percentage of this value.

Component	Parameters
Core count	64
Core type	4-way issue OOO, 128-entry ROB
Load/store queue	48 load entries, 32 store entries
L1-I cache	32 KB, 4-way, 3 cycle access time
L1-D cache	32 KB, 4-way, 3 cycle access time
L2 cache	512 KB, 16-way, 10 cycle access time, private per core
L2 prefetcher	stride-based, 8 independent streams
Coherence protocol	directory-based MESI, distributed tags
Network On-chip	16×4 mesh, 32 GB/s/link
Main memory	8 controllers, 80 ns latency, 128 GB/s total
Technology	22 nm, 660 mm^2 total area
Frequency	1.2 GHz
Vdd	0.8 V
TDP	120 W

Table 3.4: Base configuration.

3.5.3.1 Core configuration

The first configuration knob adapts the core itself. The core width can be adapted, along with the size of various structures. We maintain a quadratic relation between execution width and size of microarchitectural buffers [55]. Unused components are power-gated to reduce both static and dynamic power consumption.

3.5.3.2 Cache configuration

For cache adaptivity, we use a flushing, selective-way LLC implementation as described by [4]. By controlling which ways are on and off, we can power-gate portions of the cache to reduce its capacity and lower power usage. We use selective-ways because of their simple design, as selective-sets require changes to the number of tag bits used [178]. By using the flushing cache policy when shrinking to a smaller number of ways, we can turn off the corresponding cache ways sooner, reducing the static power consumption of the cache.

3.5.3.3 DVFS configuration

Finally, we assume the availability of on-die voltage regulation [31, 108] to enable fast per-core DVFS [80, 104] with a range between 800 MHz at 0.7 V to 1.2 GHz at 0.8 V, which is in line with Intel Xeon Phi settings [86].

3.5.4 Alternate power management policies

For comparison against Chrysso, we implemented two DVFS-based power management policies. Chip-wide DVFS *(DVFS-CW)* runs all cores at the same frequency. In some commercially available processors, this is referred to as P-states [109]. Minimum-power DVFS *(DVFS-minPower)* represents state-of-the-art per-core DVFS, which iteratively reduces frequency and voltage for the core with the lowest power [80]. The intuition is that these cores are likely memory-bound and will not give up much performance when their frequency is reduced.

In addition, we also compare Chrysso against power-gating cores until the power budget is met. Under

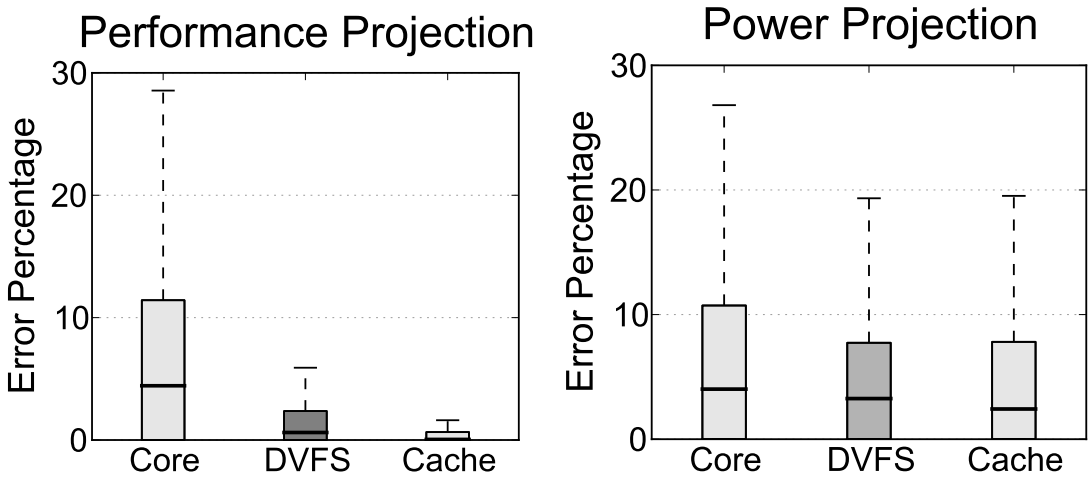


Figure 3.4: Per-knob modeling error of Chrysso's performance (*left*) and power (*right*) projection models.

core gating, an equal-time scheduler is used to time-share all 64 application threads on the active cores. We assume a time slice of 10 ms, unless mentioned otherwise.

3.6 Results and Discussion

Before presenting overall results for Chrysso during dynamic execution, we first evaluate two critical Chrysso components in isolation through offline analysis, namely projection model accuracy and optimization algorithm effectiveness.

3.6.1 Offline Analysis

3.6.1.1 Projection model accuracy

Figure 3.4 shows the (absolute) modeling error distribution for performance (left) and power (right) for all benchmarks and across all architecture configurations at a 10 million instruction granularity. Each box plot shows the 5 to 95 percentile; the horizontal line inside the box shows the mean error; the outliers are shown using the dashed line. On average, both the performance and power effects of changes made to each knob can be predicted within 6% accuracy, and 95% of all predictions are within 13%. The largest errors occur for the *Core knob*, which arguably has the simplest model, and for configurations that are far away from the current one. Still, the projections allow the reconfiguration algorithm to make the correct decision due to Chrysso's ability to run at small time slices and to quickly correct from errors by leveraging projection rescaling in subsequent time slices, as discussed in Section 3.4.

3.6.1.2 Optimization algorithm effectiveness

To measure how closely Chrysso approaches the actual global optimum, we set up a trace-based experiment in which we implement an oracle scheme. Single-program runs were done for all benchmarks and all possible configurations in order to measure the actual performance and power consumption at a granularity of 10 million instructions. Because finding the global optimum through exhaustive search is infeasible—64 cores with 48 possible configurations each, leads to a search space of 48^{64} entries—we

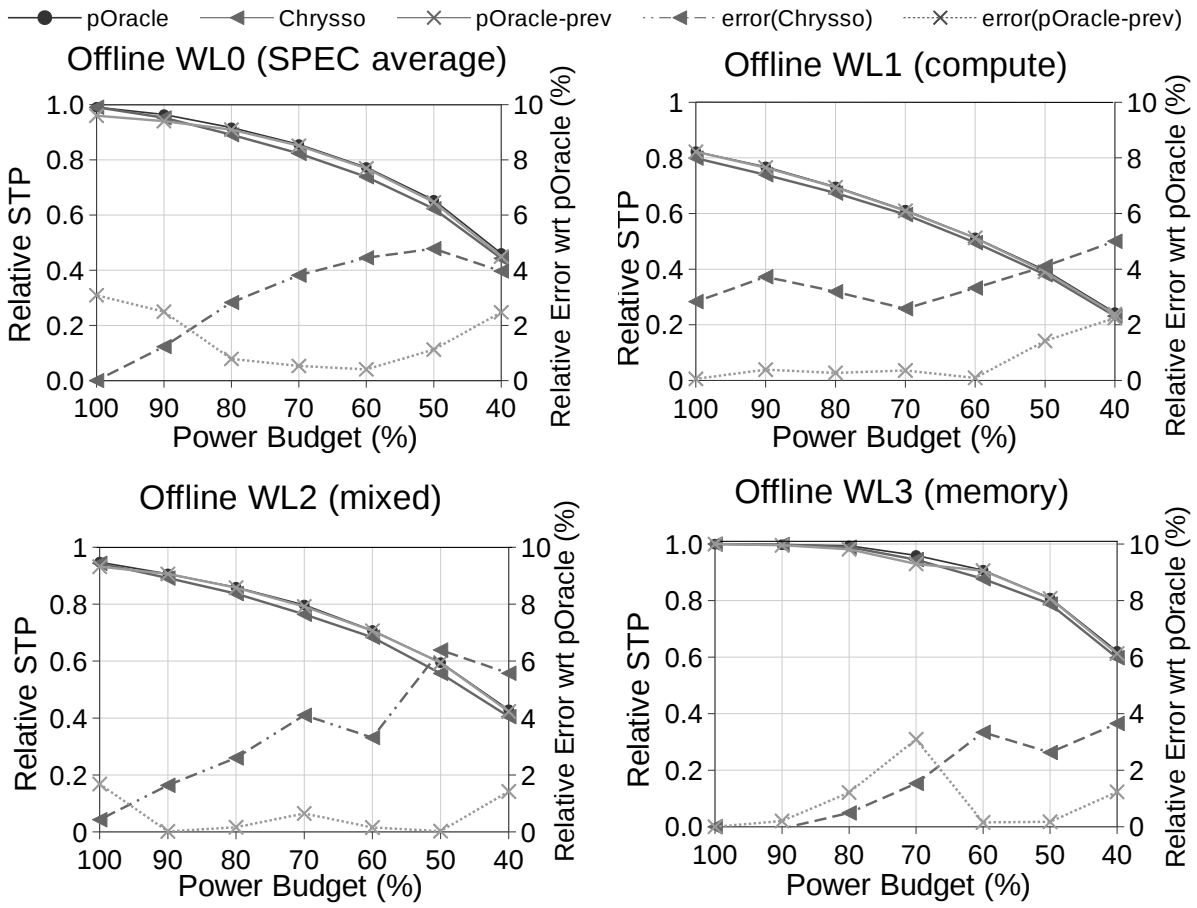


Figure 3.5: Effectiveness of Chrysso compared to pOracle through offline analysis for multi-program workloads.

consider a pseudo-oracle *(pOracle)* scheme, which we found to be within 96–100% of the true oracle for 4–6 cores.

Figure 3.5 plots the results of *Chrysso* in comparison to *pOracle*. *Chrysso* uses performance counter information from the previous time slice, applies the projection models from Section 3.4, and uses the search algorithm described in Section 3.3 to find a solution that satisfies the power budget. *pOracle-prev* forgoes the modeling step and uses actual performance and power data from the trace for the previous time slice. This allows us to isolate the effect of time-varying workload behavior. *pOracle* uses actual performance and power from the trace for the next time slice, removing both projection error and workload variability from the equation.

In the figure, solid lines plot STP relative to the full-feature base configuration (on the left vertical axis) obtained by each algorithm for the different power budgets, while dashed lines report the difference with *pOracle* (on the right vertical axis). Chrysso is able to find a solution that is within 7% of the optimum. The largest errors occur for lower power settings; these are the regions in the optimization space in which there are more degrees of freedom and it is easier to end up in a local minimum. The error for *pOracle-prev* is even lower and never exceeds 3%—showing that improving the projection models can further increase effectiveness—whereas workload variability at this time scale is of less importance.

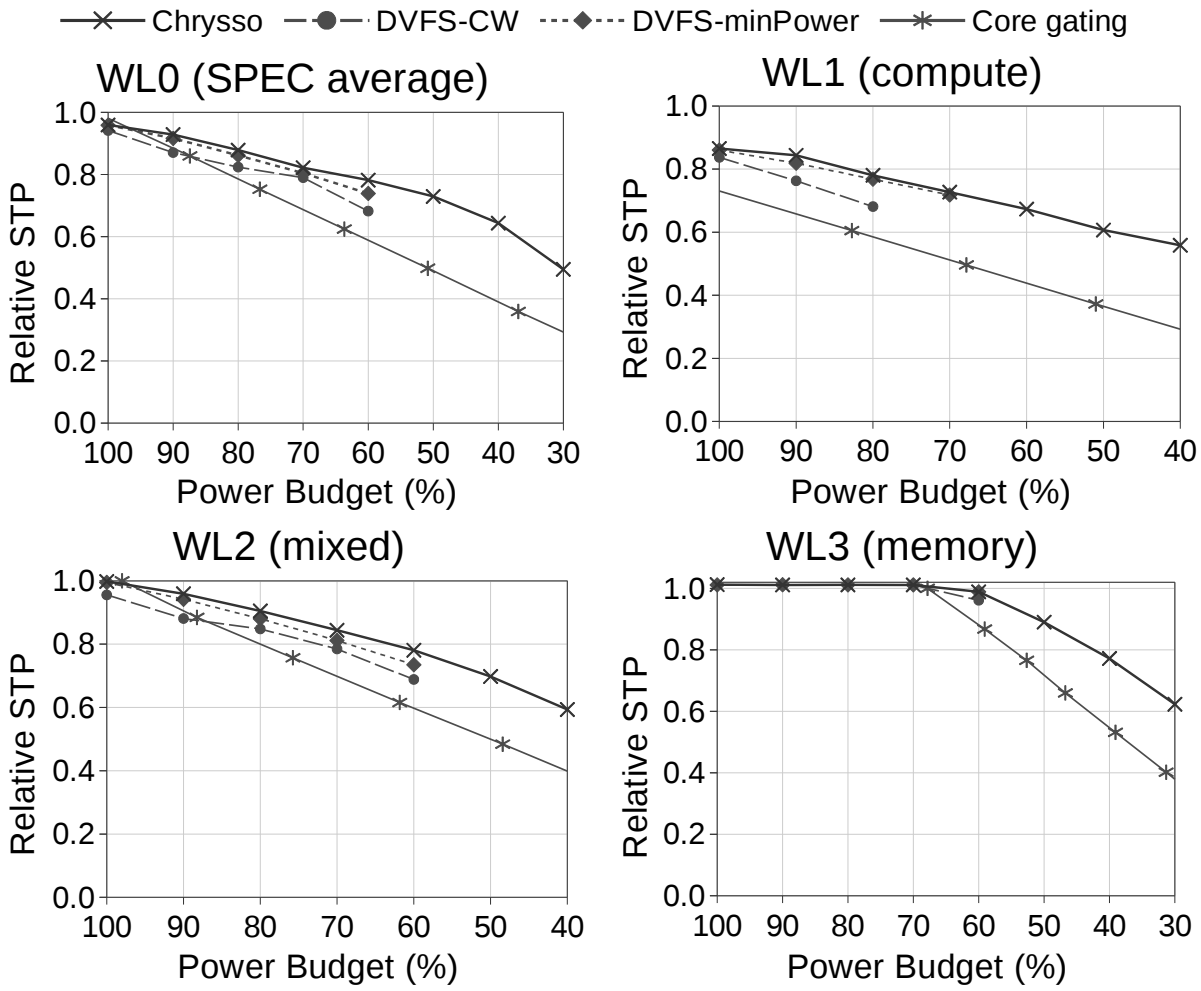


Figure 3.6: Relative STP vs. power budget: comparing Chrysso against alternative power management techniques for multi-program workloads.

3.6.2 Chrysso Evaluation

Having evaluated its two most critical components, we now evaluate Chrysso through online simulation.

3.6.2.1 Multi-program performance

Figure 3.6 plots system throughput (STP) at different power budgets for the four multi-program workloads. The horizontal axis plots the available power budget relative to the TDP (120 W). On the vertical axis we plot STP relative to a full-feature chip. Note that some workloads (most notably the compute-bound WL1) have a full-feature chip power consumption that is higher than 120 W; they therefore show performance degradation for any of the evaluated mechanisms even at 100% TDP. At the other extreme, the memory-bound workload needs only 70% TDP at full configuration, so it does not incur any performance penalty even at a power budget of only 70%. Using DVFS alone, power consumption cannot be reduced below 60% (or even below 70% for WL1; only 80% with *DVFS-CW*). By leveraging additional power saving mechanisms next to DVFS, Chrysso is able to outperform both chip-wide and per-core DVFS even for more modest power reduction settings.

Compared to core gating, both Chrysso and DVFS are able to obtain power savings that are larger than

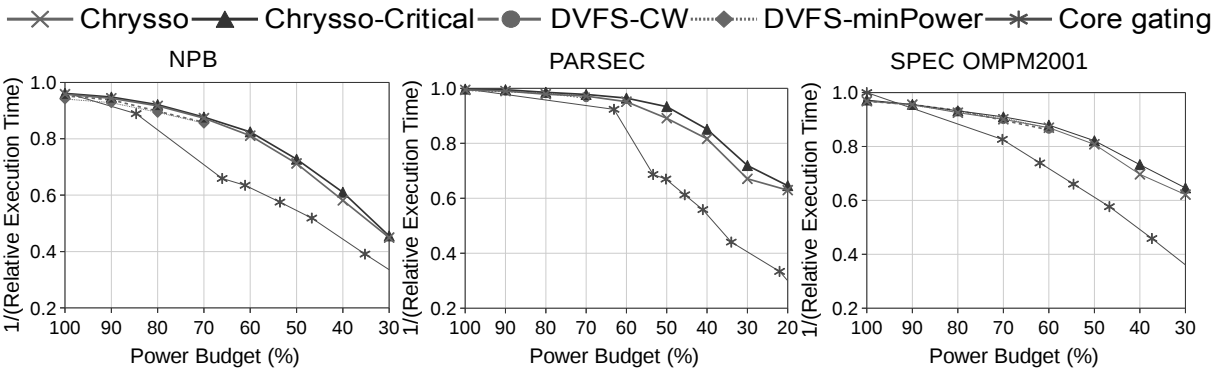


Figure 3.7: Performance vs. power budget of the evaluated power management techniques for multi-threaded benchmarks.

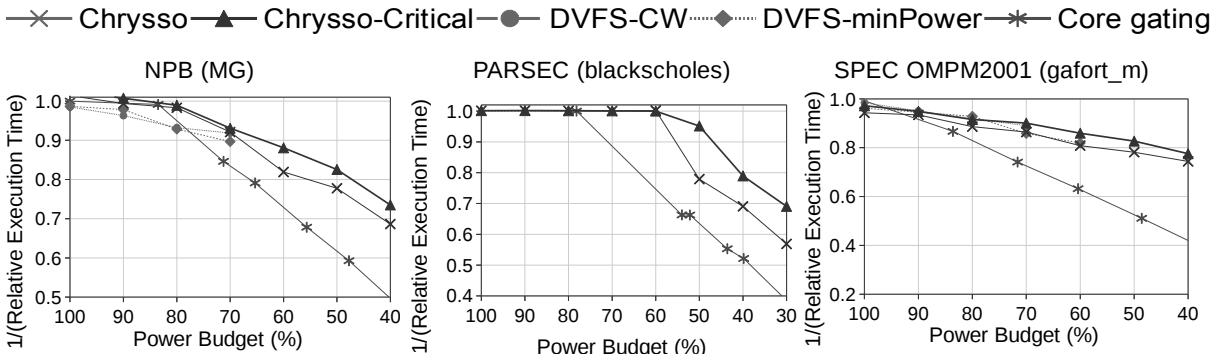


Figure 3.8: Critical-thread aware Chrysso for the thread-imbalance workloads NPB (MG), PARSEC (blackscholes) and SPEC OMPM2001 (gafort_m).

the relative loss in STP. While core gating is able to limit power consumption to any desired budget, it does so at a linear decrease in performance. In contrast, Chrysso is able to detect which parts of the chip do not contribute to performance as much and down-steps these cores first. At stringent power budgets (40%), core gating achieves a relative STP around 0.34 for WL0, while Chrysso achieves a relative STP of 0.64, or 1.9$\times$ higher.

3.6.2.2 Multi-threaded performance

Chrysso is also effective for multi-threaded workloads. Figure 3.7 plots performance (inverse of execution time relative to full-feature chip) at different power budgets. The graphs show average performance per benchmark suite. Compared to core gating, both Chrysso and DVFS are able to obtain power savings that are larger than the relative loss in execution time. In other words, while core gating is able to limit power consumption to any desired budget, it does so at the cost of a non-uniform increase in execution time (due to inter-thread dependencies). In contrast, Chrysso is able to give a relatively smaller power budget to cores that do not contribute to performance as much. Making Chrysso critical-thread aware ('Chrysso-Critical') clearly improves performance for a number of benchmarks: by 2–5% on average for NPB, 1.5–7.2% for PARSEC, and 1–5% for SPEC OMPM2001 at medium to stringent power budgets. Benchmarks that are most sensitive to workload imbalance benefit the most from making Chrysso critical-thread aware. See for example PARSEC's blackscholes in Figure 3.8: overall application performance

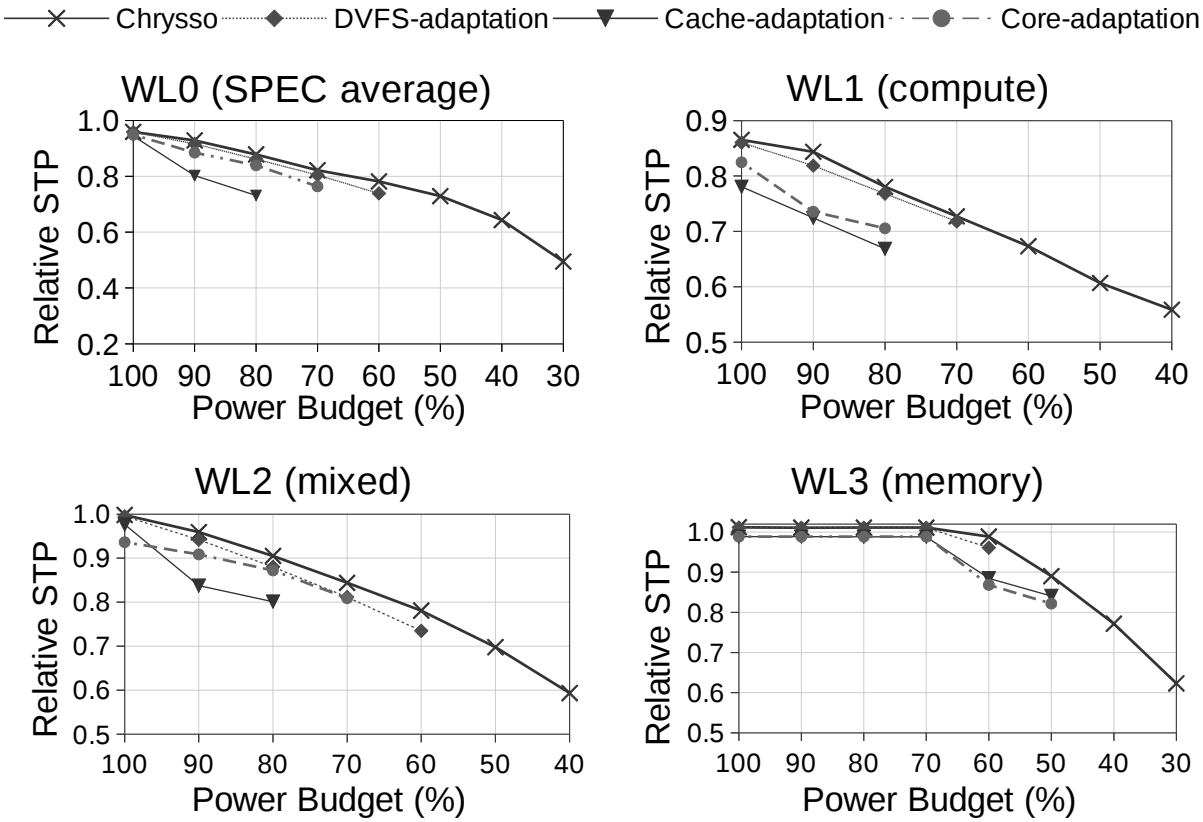


Figure 3.9: Isolated versus integrated optimization using Chrysso for the WL0 multi-program workload.

greatly improves (up to 18%) by giving a relatively larger fraction of the chip power budget to the most critical thread. Critical-thread awareness provides almost 5% and 4% reduction in execution time for applications like NPB's MG and SPEC OMPM2001's gafort_m respectively (Figure 3.8). Overall, we observe a reduction in application execution time of around 1.5× at stringent power budgets through (critical-thread aware) Chrysso over core gating.

3.6.2.3 Integrated vs. isolated optimization

As mentioned before, the key benefit of Chrysso's model-based approach is that it easily allows different adaptation methods to be combined and achieve higher performance within the same power budget. Figure 3.9 confirms this by plotting results for Chrysso in comparison to isolated adaptation (*core*, *cache* and *DVFS*) for the WL0 multi-program workload. Using adaptation techniques in isolation, the power budget can be reduced by 40% at most; on the other hand, combining all three knobs can reduce power budget by 70%. Even at moderate power budgets Chrysso outperforms isolated adaptation: at 80% of TDP, Chrysso improves system throughput by 16.8% over cache adaptation, by 17% over core gating, by 6.3% over chip-wide DVFS, by 4.5% over core adaptation, and by 2% over per-core DVFS. Furthermore, at modest power budgets, e.g. 60%, isolated adaptation (e.g., DVFS) incurs a performance hit of at least 5.5% compared to combined adaptation using Chrysso.

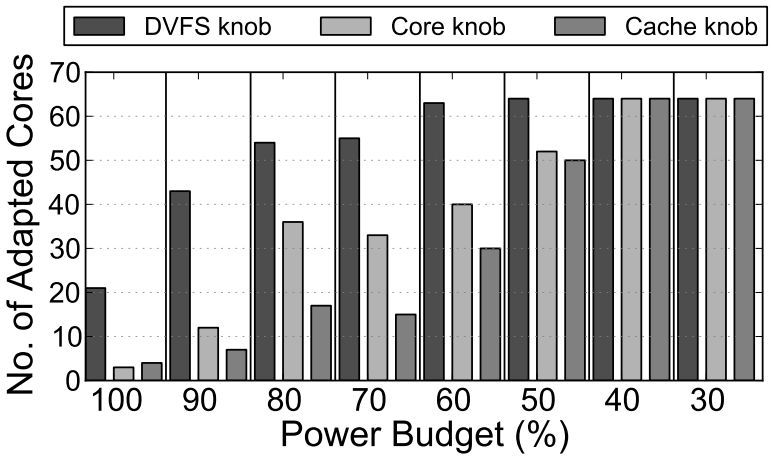


Figure 3.10: Number of cores that are adapted for each knob as a function of the available power budget for WL0.

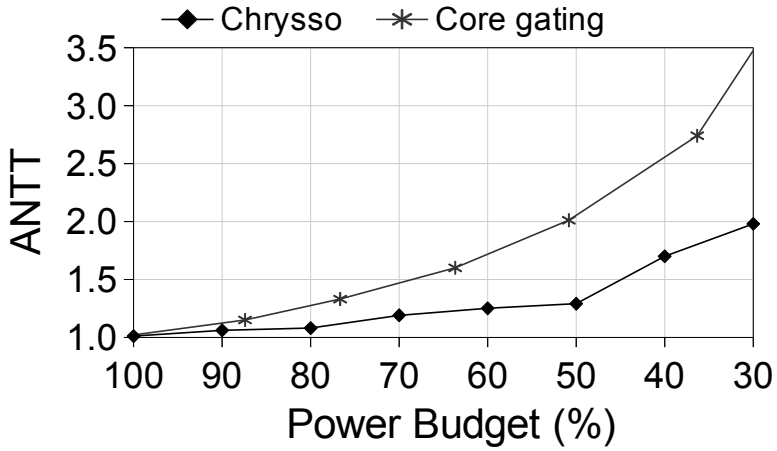


Figure 3.11: Average normalized turnaround time (ANTT) of Chrysso and core gating for the WL0 multi-program workload. *(Lower is better.)*

3.6.2.4 Power allocation

It is interesting to explore where the power savings come from using Chrysso. Figure 3.10 shows the number of cores (out of the 64) that are adapted by a specific knob at least once during its execution time. At high to moderate power budgets, we observe that most cores have their frequency and voltage adapted. At reduced power budgets, a higher number of cores have their microarchitecture adapted, followed by cache adaptation at even lower power budgets. This suggests that Chrysso redistributes power from memory-intensive applications to provide a higher relative power budget to power-hungry, compute-intensive applications, thereby increasing overall chip performance (STP).

The fact that Chrysso reallocates power among cores/threads implies that some threads may make faster progress than others. We quantify this effect using the average normalized turnaround time (ANTT) metric [54], also called the harmonic mean of speedups [117], which measures the average progress rate relative to isolated execution for all threads in the workload. Figure 3.11 shows ANTT for Chrysso compared to core gating: clearly, Chrysso achieves a much better average progress rate than core gating, up to 75% at stringent power budgets. The reason is that Chrysso is able to efficiently redistribute

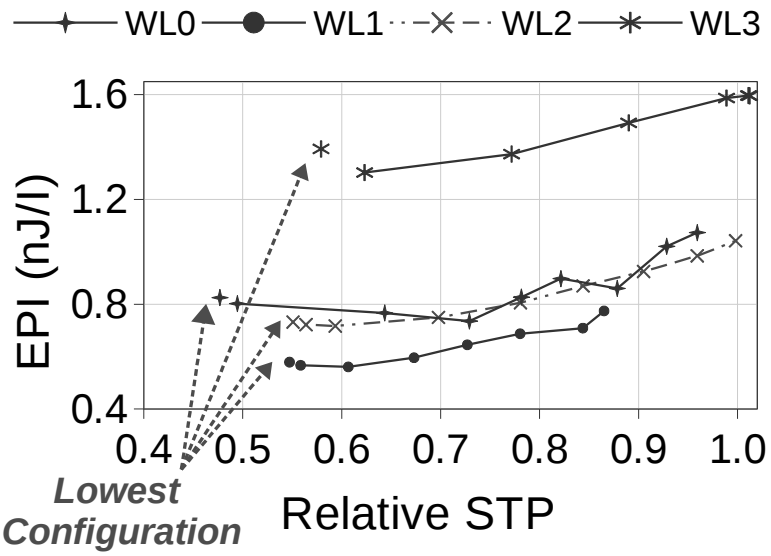


Figure 3.12: Energy per instruction versus performance at various power settings.

power based on core/threads requirements; on the contrary, since core gating is agnostic to the threads' characteristics, individual threads make limited progress due to time-sharing across the remaining, non power-gated cores.

3.6.2.5 Energy efficiency

Although Chrysso does not directly optimize for energy, it is still able to save a significant amount of energy. Figure 3.12 plots the resulting STP versus the energy per instruction (EPI, in nano-Joules/instruction) for each of the power budgets shown in Figure 3.6. The right-most (highest STP, highest EPI) points for each workload correspond to the 100% setting, while moving to the left corresponds to progressively lower power settings down to 30%. For most workloads, an energy reduction of over 1.5× is possible, while the STP cost for achieving this reduction is always under 50%. Note also that the highest energy savings are not necessarily achieved when simply running the whole chip at its minimum configuration. For instance, running WL0 at the lowest configuration costs 3.5% *more* energy than using Chrysso at a 30% power budget, while providing 5.4% lower performance.

3.6.2.6 Dynamic behavior

The dynamic behavior of Chrysso over a 100 ms execution interval is shown in Figure 3.13, at a 50% power setting. For each configuration knob (core, frequency and cache), the minimum, average and maximum settings over all 64 cores are shown at each 1 ms time slice, in addition to the number of cores for which each knob was changed in that time slice. At this power level, most cores have their frequency reduced to the lowest setting. Core reconfiguration is not used much since it usually has a high performance impact and isn't yet needed at this power setting. In contrast, cache reconfiguration is performed often since it has the most power impact, and is also most affected by application phase behavior.

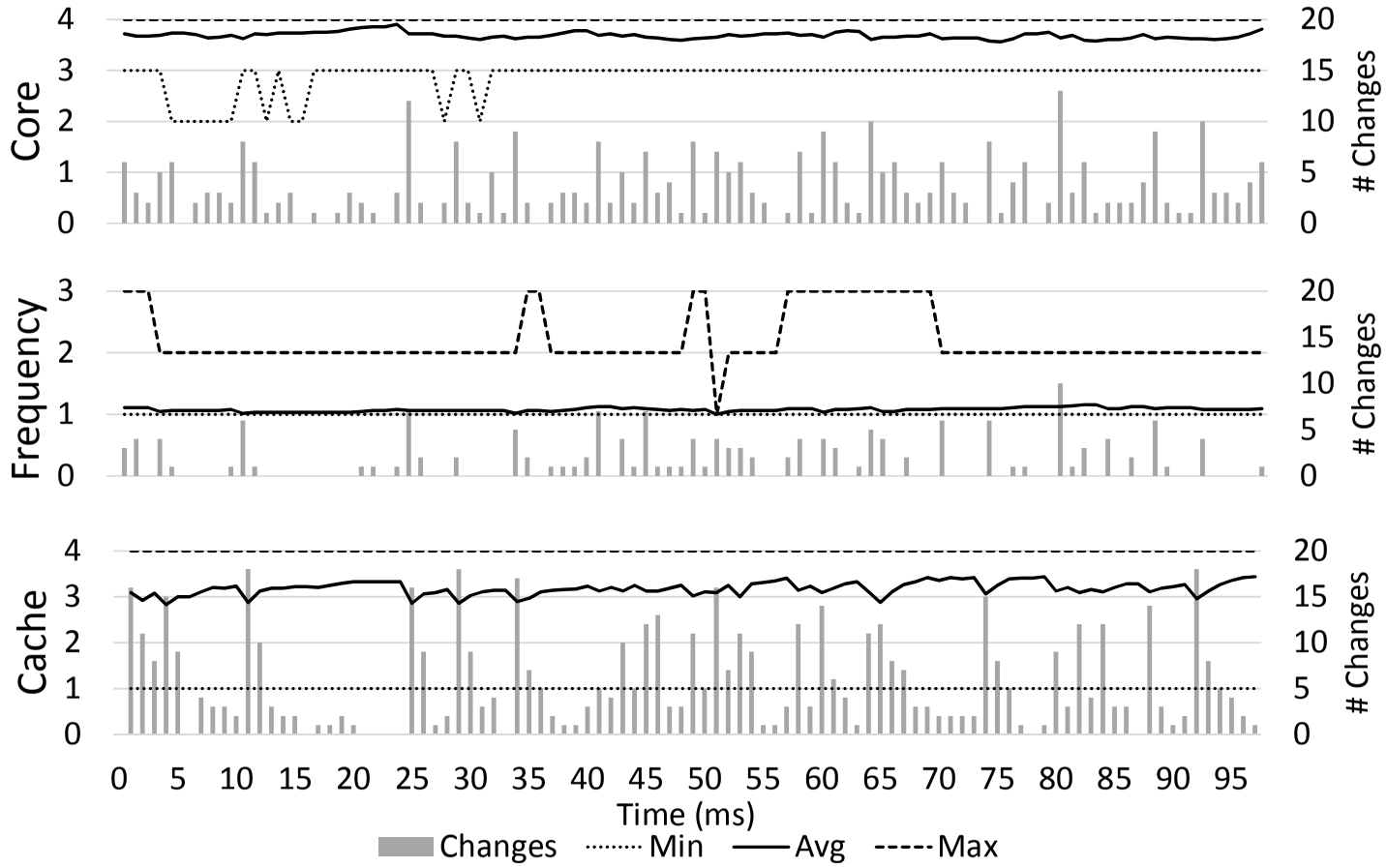


Figure 3.13: Chrysso configuration changes through time at 50% power setting and a 1 ms (reconfiguration) time slice.

3.6.2.7 Reconfiguration time slice

Chrysso is designed to enable fast reconfiguration. Figure 3.14 explores the characteristics of time slices between 100 μs and 10 ms. The left plot displays the distribution of actual power consumption relative to the specified power limit in 1 ms intervals. Due of workload variability, a 10 ms reconfiguration time slice is often too slow to ensure the limit is always met. Using 1 ms and 100 μs time slice progressively reduces the number of violations.

However, when looking at relative STP (Figure 3.14, right), it turns out that faster reconfiguration can be detrimental to performance. The main reason is that, especially at this power setting, frequent cache configuration changes result in a large amount of invalidations and writebacks from the L2 cache, and subsequent misses once the application needs this data again later. This phenomenon occurs when the reconfiguration time slice becomes shorter that the typical time between reuse of data in the caches: whereas the ATDs predicted that a smaller cache will not harm performance for a given time slice, the application does exhibit reuse at longer time scales causing the caches to be resized too aggressively. Core and frequency reconfiguration do not have this problem, as their reconfiguration cost is much smaller (we model both with 2 μs penalties).

We conclude from this experiment that for efficient execution, the reconfiguration time slice should never be shorter than the corresponding time scales of application behavior *on a per-knob basis*. At time scales of 1 ms and shorter, a single reconfiguration time slice no longer suffices as some knobs (core, frequency) can be re-tuned every time slice whereas other knobs (caches) will need to be kept constant for a number of time slices to avoid excessive switching costs.

3.6.2.8 Adapting to time-varying workload conditions

A key asset of Chrysso is its ability to quickly adapt to time-varying workload conditions, and automatically redistribute power. This is illustrated in Figure 3.15, which shows power and performance for all 64 threads (*solid lines* plotted along left vertical axis) of the swaptions benchmark when run at 50% TDP (total power and performance plotted against right vertical axis with *dashed line* and *markers*). All but one of the threads finish execution between 150 ms to 180 ms, i.e., the application transitions from a parallel phase into a serial phase. Chrysso is able to seamlessly allocate more resources to the serial thread (*solid line and '$\times$' marker*)—tuples (w, c, f) at each 10 ms time slice shows the configuration for the serial thread decided by Chrysso's adaptation algorithm.

To achieve even higher power-efficiency, we can power-gate the cores that have finished execution, which we identify by polling the core's issue queue and power-gate the entire core when the issue queue is empty for more than 1 ms. This additional optimization further reduces total power consumption during serial execution from 41.5 W to 2.2 W.

3.6.2.9 Chrysso complexity and scalability

The computational cost of Chrysso reconfiguration is composed of two parts: projection models and global optimization. The projection models amount to less than 1,000 operations to be implemented using fixed-point arithmetic and run on each core's PMU, in parallel with normal execution. Filtering for Pareto-optimal points is done on the PMU as well, resulting in 5–10 points in practice to be sent to the GPM.

The global optimization algorithm has complexity $O(N \log N)$, where N is the number of cores, as we sort all the per-core Pareto-optimal configurations by utility. In our experiments, however, we have observed that, in most cases, less than half of the cores need a configuration change at any point in the execution. To support this, Figure 3.16 shows the average number of core adaptations that Chrysso performs per

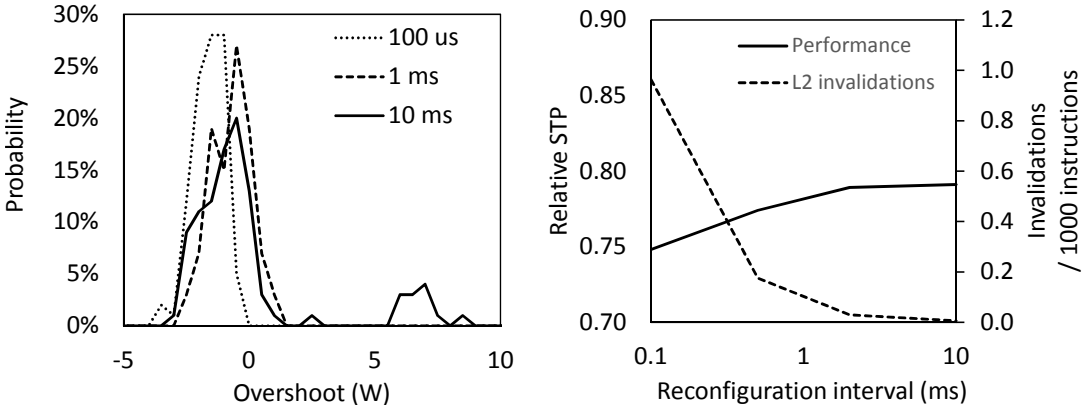


Figure 3.14: Chrysso at different reconfiguration time slice at 50% power setting.

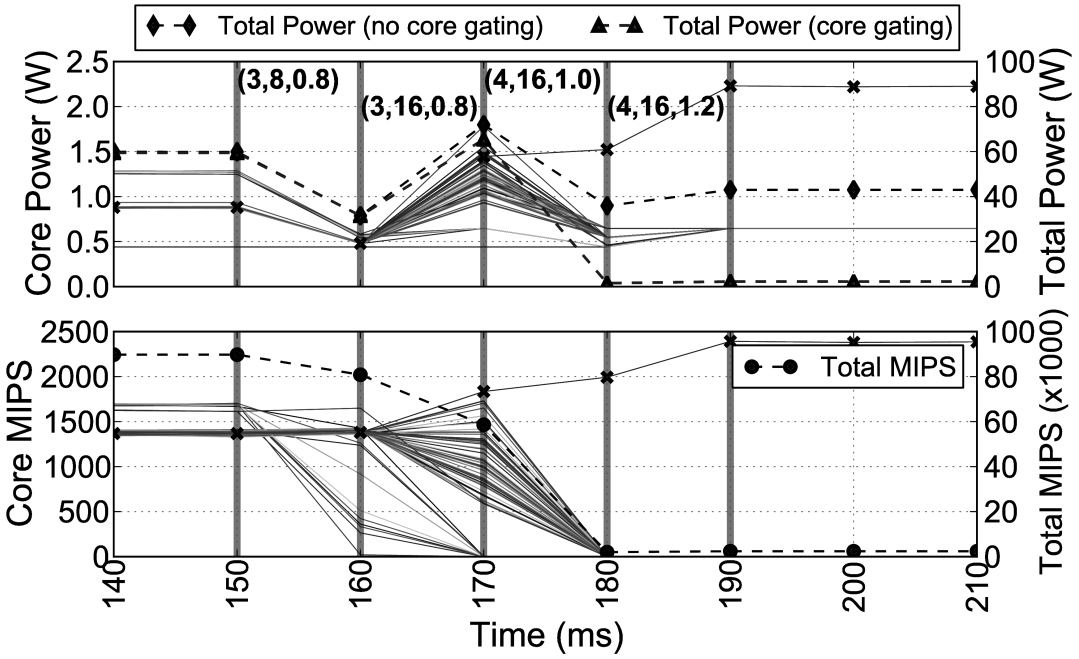


Figure 3.15: Total and per-core power (top) and performance (bottom) for swaptions at 50% TDP.

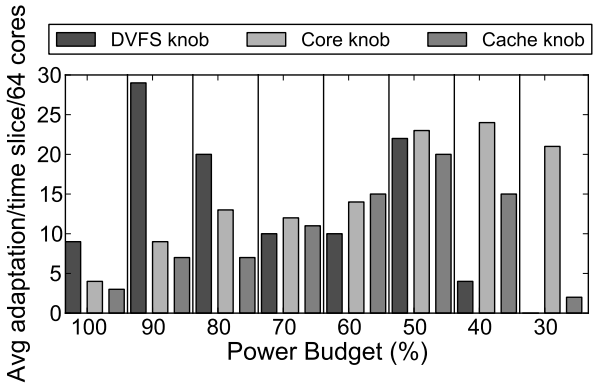


Figure 3.16: Average number of adaptations per time slice (10 ms) for the 64 cores running WL0.

time slice, for different power budgets. Moreover, the adaptation is typically a minor modification from the current configuration.

Overall, the Chrysso workflow including projection (in parallel) and optimization (centralized) takes less than 100 μs, even on modest hardware, and should be easily implementable using existing power management agents as seen in modern-day processors [149]. With a reconfiguration time slice of 10 ms, this amounts to an overhead in execution time of at most 1%.

3.7 Related Work

3.7.1 Micro-architecture Adaptation

A variety of prior work has explored how to improve power-efficiency by adapting microarchitecture structures. Folegnani et al. [59] and Bahar et al. [12] proposed to adapt the instruction window and issue logic to provide greater power/energy efficiency while showing a small reduction in application performance. Gibson et al. [63] proposed the ForwardFlow core as a way to trade off core performance for power. Albonesi [4] and Yang et al. [178] evaluated shutting down portions of the cache, either a number of ways or a combination of ways and sets for improved energy efficiency. These techniques evaluate adapting micro-architectural structures to trade off performance for power and energy. Eckert et al. [51] proposed to combine drowsy caches with front-end pipeline gating and demonstrate better performance-power scaling than dynamic frequency scaling, and even DVFS in some cases. Although their work shows that one can reconfigure the system to perform better than DVFS, they do not perform run-time optimizations of large many-cores in power-constrained environments. Dubach et al. [49] use machine-learning models (trained using profiling) to perform online adaptation of a single core at a time. On the contrary, Chrysso handles online adaptations in a many-core scenario and adapts core, caches and DVFS settings simultaneously. None of these previous works have evaluated integrated power management including fine-grain adaptation of the core microarchitecture, cache, and per-core DVFS settings for many-core processors at stringent power budgets.

3.7.2 Dynamic Power Management

Isci et al. [80] investigated a global power controller to determine different per-core DVFS settings to maximize chip-wide MIPS. Teodorescu and Torrellas [167] proposed variation-aware power-management DVFS algorithms for application scheduling in a CMP to save power or improve throughput at a given

power budget. CoScale [42] deals with co-optimizing DVFS settings for both the CPU and DRAM. Our goal is different as we try to optimize many-core performance by adapting both the core, the cache, and per-core DVFS settings. Both CoScale and Chrysso use utility-based optimization, but our models are different (we adapt core and cache next to DVFS) and we leverage per-core Pareto-optimal design points.

Jayaseelan et al. [84] proposed global DVFS with per-core adaptation based on neural networks to reach the power budget. On similar grounds, Bitirgen et al. [22] formulated global resource allocation using machine learning. Ghasemi et al. [62] proposed RCS, a mechanisms to uniformly change core resources with the number of cores (8/10/12) to exploit application variability at a fixed power budget. The proposed scheme uses a SVM-based machine-learning mechanisms to obtain the number of active cores (with corresponding microarchitectural variation) for each interval. In contrast, Chrysso uses simple projection models providing better scalability towards larger core counts even in power-constrained environment with lower overhead. Meng et al. [123] proposed DVFS adaptation along with cache adaptation for 4-core system. The key difference is that their search algorithm does not scale to large number of cores: it does not prune the global search space by first identifying per-core Pareto-optimal points; and it does not make performance/power predictions relative to the current configuration but relative to no power optimization (maximal power mode), which requires more accurate models. Furthermore, this work does not consider core adaptation, nor does it evaluate the applicability to many cores (64 cores) nor multi-threaded applications.

Petrica et al. [139] proposed Flicker, a scheme to dynamically adjust the capabilities of an out-of-order core at coarse-grained time slice (100 ms) using sampling-based global genetic algorithm to improve performance compared to core power gating at moderate power budgets. This approach has at least three limitations. First, a trade-off has to be made between the length of each sample and the number of sampled configuration points to reduce sampling overhead—during which the processor runs at a suboptimal configuration. Second, each sample is obtained by running a different part of the application; workload variability and samples are essentially not comparable. Finally, sampling may be hard to scale to large configuration spaces. In contrast, Chrysso uses projection models to provide a better scalable adaptation scheme while exploring a broader adaptation space (including cache and DVFS along with core adaptation).

3.7.3 Critical Thread Acceleration

Several prior works have proposed techniques to identify critical threads for acceleration, either by running serial parts at higher clock frequency [9, 127], by running serial code and synchronization bottlenecks on a big core in a heterogeneous multi-core [91, 130, 163]; or by speeding up critical threads in barrier-synchronized applications based on cache behavior [20]. Chrysso integrates the concept of criticality stacks [47] to accelerate critical threads and improve multi-threaded application performance under constrained conditions.

3.8 Summary

Integrated and scalable many-core power management is clearly needed as we move towards even tighter power budgets. Chrysso leverages its scalability and effectiveness from (i) using analytical performance models and table-based power models for core, cache, and per-core DVFS adaptation, (ii) a search process that identifies Pareto-optimal per-core configurations to prune the global optimization space, and (iii) utility-based optimization which reallocates power to the cores/threads that benefit the most, e.g., critical threads in multi-threaded workloads and power-hungry applications in multi-program workloads.

Chrysso outperforms isolated power adaptation techniques by a significant margin at moderate power budgets, and outperforms core gating in system performance by 1.9× and 1.5× for multi-program and

multi-threaded workloads at stringent power budgets, respectively. Chrysso incurs limited run time overhead (less than 1%) and hardware overhead (roughly 3 KB per core).

Chapter 4
Shared Resource Aware Scheduling on Power-Constrained Tiled Many-Core Processors

Power management through dynamic core, cache and frequency adaptation is becoming a necessity in today's power-constrained many-core environments. Unfortunately, as core count grows, the complexity of both the adaptation hardware and the power management algorithms increases exponentially. This calls for hierarchical solutions, such as on-chip voltage regulators per-tile rather than per-core, along with multi-level power management. As power-driven adaptation of shared resources affects multiple threads at once, the efficiency in a tile-organized many-core processor architecture hinges on the ability to co-schedule compatible threads to tiles in tandem with hardware adaptations per tile and per core.

In this chapter, we propose a two-tier hierarchical power management methodology to exploit per-tile voltage regulators and clustered last-level caches. In addition, we include a novel thread migration layer that (i) analyzes threads running on the tiled many-core processor for shared resource sensitivity in tandem with core, cache and frequency adaptation, and (ii) co-schedules threads per tile with compatible behavior.

4.1 Introduction

Industry-wide adoption of chip multiprocessors (CMPs) is driven by the need to maintain the performance trend in a power-efficient way on par with Moore's law [129]. With continued emphasis on technology scaling for increased circuit densities, controlling chip power consumption has become a first-order design constraint. Due to the end of Dennard scaling [43] (slowed supply voltage scaling), we may become so power-constrained that we are no longer able to power on all transistors at the same time — *dark silicon* [52]. Runtime factors such as thermal emergencies [29] and power capping [57] further constrain the available chip power. Owing to all the above factors, power budgeting on many-core systems has received considerable attention recently [62, 110, 118, 128, 167, 173].

Dynamic voltage and frequency scaling (DVFS) for multiple clock domain micro-architectures has been studied extensively in prior work [42, 73, 80, 167, 176]. Current commercial implementations of fully integrated voltage regulators (FIVR) [31, 108] support multiple on-chip frequency/voltage domains with fast adaptation, although per-core voltage regulators incur significant area overhead — 12.5% of core area [31, 107, 161]. Other techniques such as core micro-architecture adaptation [12, 49, 59, 89, 139], cache adaptation [4, 123, 154, 178] and network-on-chip adaptation [154] have been shown to be quite effective at managing power in isolation at high to moderate power budgets. Under more stringent power conditions, core gating [109, 110] along with the above techniques can be used at the potential risk of starving threads.

Most existing power management schemes use a centralized approach to regulate power dissipation

based on power monitoring and performance characteristics. Unfortunately, the complexity and overhead of centralized power management increases significantly with core count [50]. Moreover, the area overhead of on-chip voltage regulators is significant which limits the number of voltage/frequency domains one can have on the chip. Hence, it becomes a necessity to employ a hierarchical approach as we scale fine-grain power management to large many-core processors at increasingly stringent power budgets. We therefore propose a *two-tier hierarchical power manager* for tile-based many-core architectures; each tile consists of a small number of cores and a shared L2 cache within a single voltage-frequency domain. The two-tier power manager first distributes power across tiles, and then across cores within a tile. The architecture also provides support for core, cache and frequency adaptations to avoid core gating at moderate to stringent power budgets.

Tiled many-core processors pose an interesting challenge when it comes to hardware adaptation and scheduling. Changing frequency and reconfiguring the shared L2 cache affects all threads running in the tile. It therefore becomes important to *migrate* threads, such that threads with *compatible* behavior are co-scheduled onto the same tile. Since the execution behavior varies over time, periodic re-evaluation and dynamic thread migration is also required. We therefore classify threads based on their sensitivity to both cache and frequency dynamically at runtime. We propose DVFS and Cache-aware Thread Migration (*DCTM*): a scheduler running on top of the two-tier hierarchical power manager to ensure an optimal co-schedule for all threads running on the power-constrained tiled many-core processor while accounting for the effects of hardware adaptation.

We make the following contributions:

- We propose *integrated two-tier hierarchical power management* for tiled many-core architectures, in which we first manage power across tiles and then within a tile.
- For a collection of multi-program and multi-threaded workloads, we report that our two-tier hierarchical power manager outperforms a centralized power manager by 3% on average, and up to 20% for a 256-core setup.
- We make the observation that thread scheduling is essential in a tiled many-core architecture to account for thread sensitivity towards shared resources. We classify threads based on their sensitivity to both cache and frequency adaptation, and we propose *DVFS and Cache-Aware Thread Migration (DCTM)* to optimize per-tile co-scheduling of compatible threads.
- We provide a comprehensive evaluation of *DCTM* on a tiled many-core processor. We use multi-program workloads consisting of both single-threaded and multi-threaded applications, and we report that *DCTM* improves system performance by 10% on average, and up to 20%. *DCTM* outperforms existing solutions by 4.2% on average (and up to 12%).

4.2 Motivation

4.2.1 Limitations of a Centralized Approach

In the context of power management in many-core processors, prior works [42, 110, 123] have relied on a central entity to manage power using one or more micro-architectural techniques to trade off performance at high to moderate power budgets. At stringent power budgets, neither of power management schemes like DVFS nor core adaptation nor cache resizing *in isolation* can provide a viable solution. As a result, prior work [109, 110] had to resort to core gating at stringent power envelops. Previously proposed state-of-the-art frameworks [89, 123, 139] provide an integrated framework for multi/many-core architectures by combining and coordinating core adaptation, cache resizing and/or per-core DVFS to maximize system performance across a wide range of power budgets. These frameworks provide some form of global power management that operates on the runtime statistics of each core to decide on an optimal per-core working configuration. During each time slice, a per-core *Performance Monitoring*

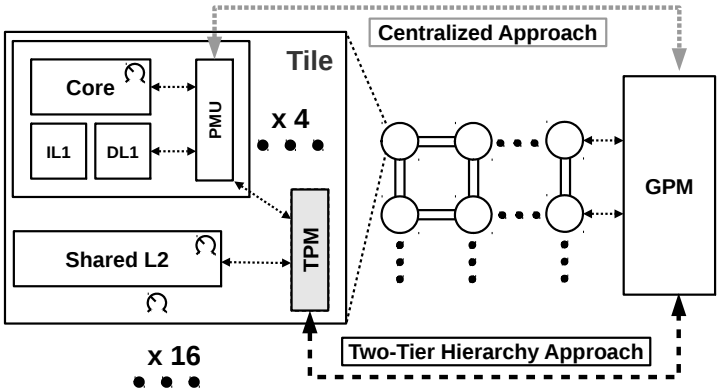


Figure 4.1: Generic tiled many-core architecture with centralized (top) versus hierarchical (bottom) power management.

Unit (PMU) tracks activity statistics using hardware counters, and predicts/projects the performance and power of all possible configurations. Each core's PMU sends a list of optimal configurations to the *Global Power Manager* (GPM), which globally optimizes the many-core configuration within the given power budget. The GPM instructs each core to reconfigure itself based on the global optimization.

In commercial designs, both the per-core PMU and global GPM are already present in some form [149]. The PMU typically collects power consumption and junction temperatures, and performs control functions such as P-state (DVFS) and C-state (various levels of power gating) transitions. The GPM is implemented as an integrated micro-controller and runs firmware algorithms that interface with the PMUs and on-chip voltage regulators. The PMU keeps track of a core's activity and controls the micro-architectural configuration in response to requests made by the GPM; the GPM combines information from all cores and performs the global power/performance optimization, see *Centralized Approach* in Figure 4.1. But as core count continues to grow, the centralized approach becomes inviable: Deng et al. [42] report *quadratic* computational complexity, while Li and Martinez [110] suggest the computational complexity to be *logarithmic* to core count. In future many-core processors [25], a centralized GPM — even with logarithmic complexity — would be a severe bottleneck.

Because a centralized power manager does not scale favorably towards large many-core processors and fine-grain hardware adaptations, we propose *two-tier hierarchical* power management (see Section 4.3).

4.2.2 Cache-aware Thread Migration (Cruise)

When threads are co-scheduled on a multi-core processor with a shared last-level cache (LLC), conflicting thread behavior can lead to suboptimal performance. For instance, when a thread whose working set fits in the shared cache is co-scheduled with a streaming application, the quick succession of cache misses from the streaming application may push the working set of the first application out of the shared cache, thereby significantly degrading its performance. Jaleel et al. [83] propose Cruise: a hardware/software co-designed scheduling methodology that uses knowledge of the underlying LLC replacement policy and application cache utility information to determine how best to co-schedule applications in multi-core systems with a shared LLC.

Cruise monitors the number of LLC accesses per kilo instructions (APKI) and miss rate (MR) for each application. Application classification based on these metrics along with co-scheduling rules then optimize overall system performance. The applications are classified in the following categories:

- Core Cache Fitting (CCF): CCF applications fit in the smaller levels of the cache hierarchy and hence

the LLC size has little impact on performance.

- LLC Trashing (LLCT): LLCT applications are mostly streaming applications with large working sets — larger than the available LLC size. The LLCT applications degrade performance of any application that benefits from the shared LLC.
- LLC Friendly (LLCFR): LLCFR applications are sensitive to the shared LLC size. They benefit from additional LLC capacity, but performance degrades when co-executed with LLCT applications.

The co-scheduling rules in Cruise are as follows[1]:

1. Group LLCT applications onto the same tile/LLC.
2. Spread CCF applications across all tiles/LLCs.
3. Co-schedule LLCFR with CCF applications.

The performance of LLCFR/LLCF applications degrades significantly when they do not receive the bulk of the shared LLC, hence Cruise schedules LLCFR applications with CCF applications whenever possible.

Cruise assumes that all cores run at the same clock frequency. In other words, it does not take DVFS sensitivity into account. This is a limitation as LLCT and (especially) LLCFR applications, being mixed compute- and memory-bound, may be quite sensitive to frequency. We overcome this limitation by proposing DCTM (see Section 4.4).

4.3 Two-Tier Hierarchical Power Management

The *Centralized* approach as described in Section 4.2.1 is inappropriate for large-scale many-core processors, for two reasons. First, it assumes per-core DVFS adaptation which is infeasible for many-core processors as it requires on-chip voltage regulators for all cores, which would incur fairly high chip area overhead [31, 107]. Second, the runtime complexity and overhead of a *Centralized* approach increases considerably with core count.

To address these two limitations, we group cores per tile and add an intermediate layer for power management, the *Tile Power Manager (TPM)*; see *Two-Tier Hierarchy Approach* in Figure 4.1. Chip power is managed via a hierarchical power manager with a GPM steering the per-tile TPMs. This organization reduces the runtime overhead of the power manager dramatically. To quantify the power manager's runtime overhead, we set up the following experiment. We consider an average multi-program workload on a many-core processor with varying core count (we run workload *WL0*, see Section 5.4 for more experimental details). The power manager is invoked every 1 ms. Figure 4.2 quantifies the worst-case overhead of the power manager normalized to an idealized run without power management overhead. We observe that the overhead increases substantially with core count. However, when considering a tiled architecture and a two-tier hierarchical power manager, we are able to significantly reduce the runtime overhead of the power manager. In other words, by keeping the GPM relatively simple and passing more functionality to the TPMs, we avoid GPM to be a bottleneck at high core count. Moreover, as all TPMs can work in parallel, the complexity of the two-tier approach equals O(G) + O($T_c log T_c$), with T_c denoting the number of physical cores per tile, and G the complexity of the GPM (constant in our case). One could adopt an even deeper hierarchy, which would be beneficial in a design with more arbitration levels (intermediate nodes acting as arbitrators for a group of tiles).

[1]In addition to the above mentioned categories, the authors also identify LLC fitting (LLCF) applications by monitoring the miss rate of the application with half the capacity of LLC. In general, these applications exhibit cache characteristics that are similar to LLCFR. In our implementation, we classify LLCF as LLCFR to limit additional hardware overhead especially pertaining to the smallest shared LLC size (see Table 4.1).

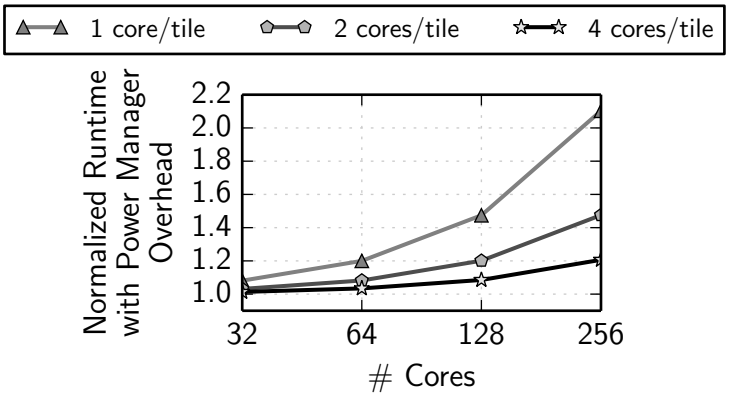


Figure 4.2: Normalized runtime overhead for hierarchical power management with varying tile size at 1 ms time slice.

4.4 DVFS and Cache-aware Thread Migration (DCTM)

A tiled many-core processor architecture with hierarchical power management, as we just established in the previous section, poses a new challenge as threads running on the same the tile share the L2 cache (LLC) and a common clock frequency. In other words, and in contrast to Cruise, threads running on the same tile not only share the LLC but also share a common clock frequency. Therefore, it is important to take both cache size sensitivity and frequency sensitivity into account when mapping threads to tiles, i.e., the thread migration layer needs to be aware of the sensitivity to both DVFS and LLC size.

4.4.1 DVFS and LLC Sensitivity Analysis

To understand an application's sensitivity to clock frequency and LLC size, we set up the following off-line analysis. We run simulations with 55 SPEC CPU2006 application traces for 750 million instructions to observe the performance sensitivity with respect to both LLC and frequency settings. Figure 4.3 plots application performance sensitivity to frequency changes, expressed as the ratio between its performance reduction and the reduction in frequency that was applied. Applications are clustered by their LLC-aware classification type (following Cruise), and plotted in ascending order of sensitivity within each cluster based on Equation 4.1:

$$sensitivity_{freq} = \frac{(MIPS_{freq_A}/MIPS_{freq_B})}{(freq_A/freq_B)}. \tag{4.1}$$

Intuitively, memory-bound applications (LLCT) should have low sensitivity to a change in frequency, while workloads that are completely core-cache fitting (CCF) would see a linear degradation as they are compute-bound. We observe that LLCT applications can still be affected by frequency variations (see the extreme end of LLCT region). The performance of these applications could be significantly affected at stringent power budgets.

We categorize applications into the following DVFS-aware classes, according to their performance sensitivity to DVFS based on Equation 4.1:

- High Sensitivity (HS, $> 66\%$): These applications are highly sensitive to DVFS. The performance of these applications is severely affected when migrated to a tile running at low frequency, whereas performance improves significantly if they can be migrated to a higher-frequency tile. These applications are generally compute-bound.

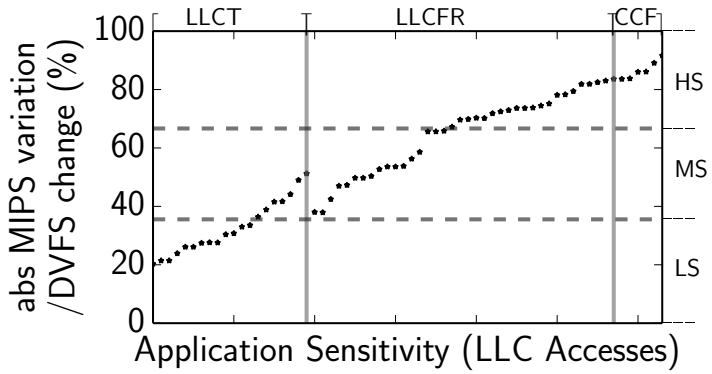


Figure 4.3: Application classification based on LLC and DVFS sensitivity.

- Moderate Sensitivity (MS, 35–66%): These applications are moderately affected by DVFS. Applications with a mix of compute-bound and memory-bound operations are grouped in this category.
- Low Sensitivity (LS, $< 35\%$): These applications degrade slightly when running at a low DVFS setting. It is therefore beneficial to reduce frequency as much as possible to save power. These applications are typically memory-bound.

When co-scheduling applications, the application categorization based on LLC usage (see Cruise, Section 4.2.2) needs to work in tandem with the DVFS sensitivity categorization as just described. Hence, combining the LLC and DVFS classifications, we have 3×3 categories of applications. Not all combinations occur in practice though, as there is some correlation between LLC and DVFS behavior; for instance, CCF applications are almost always compute-bound and hence have high DVFS sensitivity (HS). Figure 4.3 identifies five categories: LLCT with LS and MS, LLCFR with MS and HS, and CCF with HS.

4.4.2 DCTM Scheduling Rules

DVFS and Cache-aware Thread Migration (DCTM) leverages these classifications to steer scheduling of threads to tiles. The power manager will then assign the appropriate adaptation per tile (for frequency and LLC size) and per core (for core configuration). Intuitively speaking, DCTM maps threads with the same classification onto the same tile. Tiles with only LS threads will naturally be configured to run at low frequency (saving power without sacrificing performance much), while tiles with only HS threads preferably use a larger fraction of the total power budget to run at a higher frequency and boost overall system performance. In contrast, mixing LS, MS and HS threads on a single tile leads to a suboptimal situation: either the tile is set to run at low frequency, penalizing performance for the HS threads; or it runs at high frequency which accommodates the HS threads, but wastes power as it does not improve performance of the LS threads. Combining this intuition with the cache-aware scheduling, we create the following scheduling rules for DCTM:

1. Co-schedule LLCT-LS applications on the same tile.
2. Co-schedule LLCT-MS applications on the same tile.
3. Co-schedule CCF-HS applications on tiles with LLCT-MS applications to account for performance impact due to shared LLC contention.
4. Co-schedule the remaining LLCFR-MS and LLCFR-HS applications on the remaining tiles. If possible, co-schedule them with LLCFR-HS applications that are in the CCF-HS category to avoid performance degradation due to shared LLC contention.

The intuition behind co-scheduling all the LLCT-LS applications together onto a tile is that with relatively little allocated power, the co-running applications would incur minimal performance loss. Since the behavioral characteristics of all LLCT-LS applications are similar, the resource requirement would also be similar. The same intuition can be applied to LLCT-MS applications as well; being more sensitive to DVFS, these applications would have better performance than LLCT-LS applications and hence the GPM would allocate a larger fraction of the total power budget to these tiles compared to the LLCT-LS tiles. The applications in the LLCFR-MS and LLCFR-HS categories are co-scheduled or combined with CCF-HS to avoid the performance impact due to the shared LLC. Since the applications in these three categories have moderate to high performance along with much higher sensitivity to DVFS change than LLCT-LS and LLCT-MS applications, the GPM will allocate a relatively larger fraction of the power budget to these tiles, thereby limiting the performance degradation.

4.4.3 Putting It All Together

The DCTM power manager runs at two time scales. The coarse-grain timescale, at 20 ms in our setup, groups threads to tiles using the DCTM scheduling rules as just described in the previous section. One solution to classifying workloads in terms of LLC and DVFS sensitivity may be to employ sampling, i.e., by running a workload's performance at different frequency settings and different LLC sizes for short durations of time. The limitation is that it incurs significant overhead as we would need to monitor performance for various combinations of LLC size and frequency setting. Instead, we leverage the simple, yet effective analytical performance models proposed in [89] to estimate the performance impact of clock frequency and LLC size on overall performance.

The fine-grain timescale, at 1 ms in our setup, distributes power across tiles: the GPM distributes power across all tiles, and within each tile, the TPM regulates the hardware adaptations as per the allocated power. Our processor architecture allows three adaptations: core adaptation, LLC resizing, and per-tile DVFS, as we will describe in more detail in Section 4.5.2. The first fine-grained time slice (1 ms) assumes no power capping, and runs each thread at the maximum configuration (largest core configuration, largest LLC size, highest frequency). We compute the performance of each tile as a ratio of total system performance, i.e., per-tile MIPS divided by chip-wide MIPS. The GPM distributes the total available power budget across all tiles for the next time slice per the MIPS ratios of the tiles in the previous slice, i.e., a high-performance tile is given a larger fraction of the available power budget. The intuition is that compute-intensive tiles need a larger fraction of the total power, boosting overall system performance. Once total power is distributed across the tiles, the TPMs then decide on the optimal configuration for the core, LLC and DVFS setting in each tile. TPM steers adaptation using the performance/power models proposed in [89], with the goal of optimizing performance within the available power budget. Note that, the adaptation and monitoring can be achieved using other frameworks as well with modifications.

4.4.4 Quantifying DVFS Sensitivity: DCTM vs. Cruise

To illustrate the importance of being DVFS aware, we now compare the performance of *DCTM* against *Cruise* for one particular workload consisting of four LLCT SPEC CPU2006 benchmarks running on a tiled architecture with two cores per tile. (For *Cruise*, we replace the *DCTM* scheduling rules by *Cruise*'s at the coarse-grain timescale, while considering the same two-tier power manager at the fine-grain timescale.) Figure 4.4 illustrates how *DCTM* obtains higher overall performance compared to *Cruise*. The applications are arranged in a random fashion at the start of the execution. The top graphs show per-thread performance for both *DCTM* and *Cruise*, while the bottom graphs show the power and frequency settings of both tiles. All graphs have time on the horizontal axis, and run over the course of 100 ms.

To *Cruise*, all four threads belong to the same category, hence no thread migrations are needed. Taking DVFS sensitivity into account as we do in *DCTM*, however, we find that threads th_0 and th_2 have low

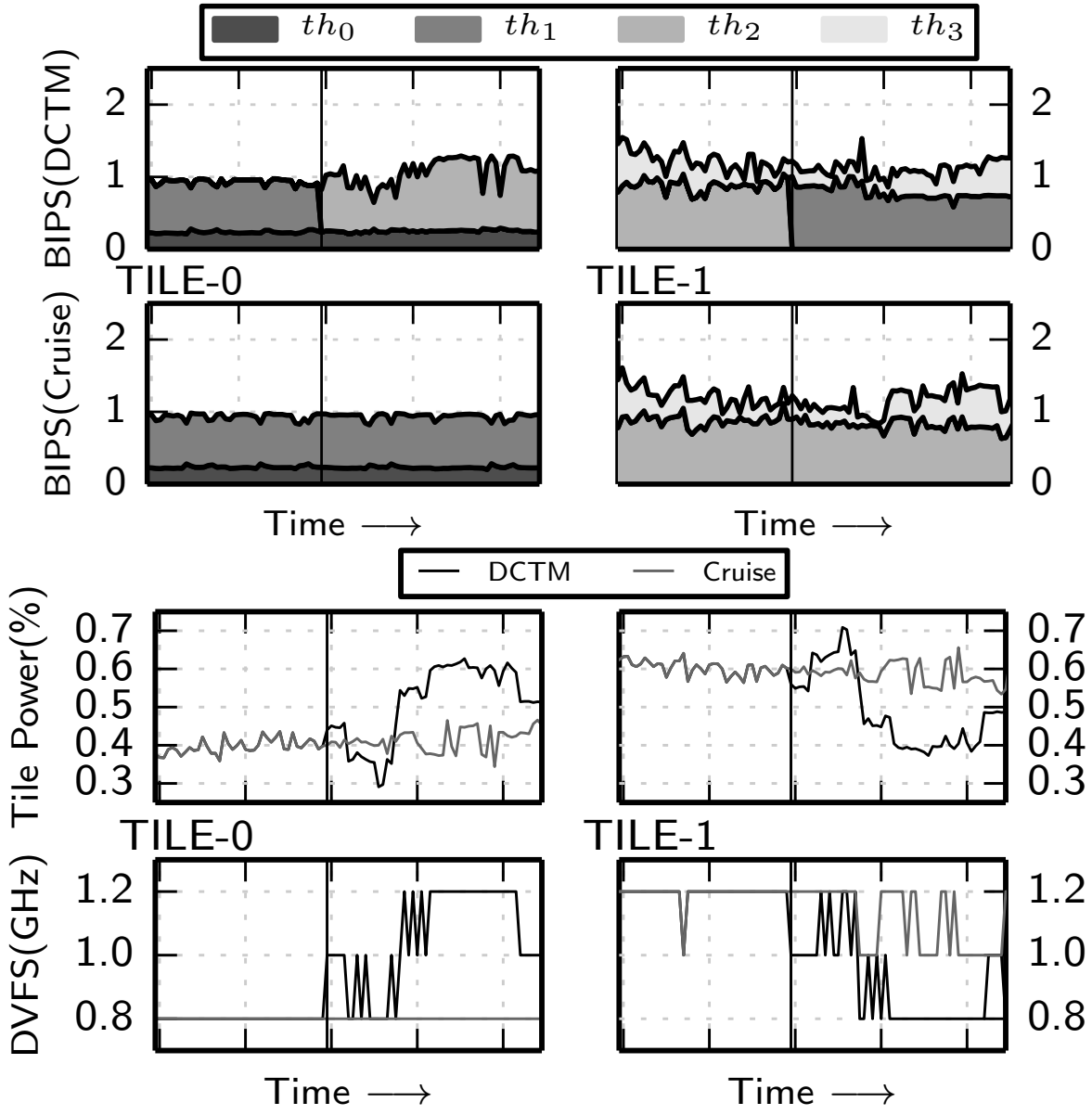


Figure 4.4: *DCTM* and *Cruise* through time for 4 cores with LLCT applications.

Parameter	Values			
Core adaptations				
ROB size	16	32	64	128
Reservation station entries	4	8	16	32
Load queue entries	6	12	24	48
Store queue entries	4	8	16	32
DVFS adaptations per-tile				
Frequency (GHz)	0.8	1.0	1.2	—
Voltage (V)	0.7	0.75	0.8	—
Shared LLC adaptations per-tile				
Cache ways	4	8	12	16
Capacity (KB)	512	1024	1536	2048

Table 4.1: Micro-architectural adaptations

sensitivity (LLCT-LS) while th_1 and th_3 have medium sensitivity (LLCT-MS). *DCTM* will therefore swap threads th_1 and th_2 to co-schedule threads with LLCT-MS behavior together (Rule #2 in Section 4.4.2). After migration, Tile-0 will run both LLCT-MS threads while Tile-1 runs both LLCT-LS threads. Hence, the power budget for Tile-0 can be increased which, due to running threads with high DVFS sensitivity, translates into a significant performance boost. At the same time, the power and frequency of Tile-1 can be reduced at limited performance cost, given that it runs both of the LS threads. The end result is an improvement in total system performance by 2.5% while staying within the same power budget.

4.5 Experimental Setup

4.5.1 Simulation Framework

Performance simulator We use the Sniper multi-core simulator [37], version 6.0, and added support for dynamically changing core and cache parameters. The core adaptation and DVFS transitions combined take $2\,\mu$s during which no computations can be performed — a conservative approach. When reducing the number of cache ways, dirty lines are written back through the simulated memory subsystem, consuming NoC and DRAM bandwidth (observed to account for no more than 5% of total DRAM bandwidth).

Power consumption McPAT version 1.0 is used to estimate static and dynamic power consumption [111] for a 22 nm technology. Power savings incurred by reconfiguration are modeled by running McPAT with the modified target parameters (Table 4.1). Running McPAT along with the performance simulation allows us to emulate the behavior of hardware energy counters at simulated time slices of 1 ms. Note that, changing V/F setting while keeping the other micro-architecture knobs unchanged, we observe that the array layout/size of SRAM and CAM structures does not change in McPAT.

4.5.2 Adaptive Micro-Architecture

To keep all the cores active even at stringent power budgets, we incorporate core micro-architectural adaptation, LLC adaptation and DVFS adaptation simultaneously, thereby providing various operational points in our adaptive tiled many-core processor. As described before, we use the notion of a Globally

Component	Parameters
Core configuration	
Core type	4-way issue OOO, 128-entry ROB
Load/store queue	48 load entries, 32 store entries
L1-I cache	32 KB, 4-way, 3 cycle access time
L1-D cache	32 KB, 4-way, 3 cycle access time
Tile configuration	
Tile size	4 cores
Core count	64, 128, 256
Tile count	16, 32, 64
L2 cache (per-tile)	2048 KB, 16-way, 10 cycle access time
L2 prefetcher	stride-based, 8 independent streams
Coherence protocol	directory-based MESI, distributed tags
Network on Chip	mesh 16×1, 16×2, 16×4 32 GB/s/link
Main memory	8, 16, 32 controllers 80 ns latency, 128 GB/s total
Chip wide configuration	
Frequency-Vdd	1.2 GHz @ 0.8 V
Technology	22 nm
TDP	100 W, 190 W, 350 W

Table 4.2: Tile-based many-core architecture.

Asynchronous Locally Synchronous (GALS) design [81], in which each tile maintains its own voltage-frequency domain. The adaptive core/tile configuration is expressed as a tuple $[core, f_t, llc_t]$, denoting that the core is configured as $core$, running at frequency f_t and llc_t cache ways enabled for the given tile t (see also Table 4.1).

Core Core adaptation pertains to reconfiguring the core micro-architecture. The core width can be adapted, along with the size of various structures (see 'Core adaptation' in Table 4.1). We maintain a quadratic relation between execution width and size of micro-architectural buffers [55]. Unused components are power-gated to reduce both static and dynamic power consumption, providing for an interesting opportunity for power savings for memory-bound or otherwise low-ILP applications. In our tiled architecture, we assume each core's micro-architecture can be adapted individually.

DVFS DVFS adaptation is a widely used technique for enforcing power budgeting. In the proposed architecture, we assume the availability of on-die voltage regulators [31, 108] per-tile to enable DVFS from 0.8 GHz at 0.7 V to 1.2 GHz at 0.8 V (see also Table 4.1), which is in line with the Intel Xeon Phi [86]. In the tiled architecture, the TPM needs to enforce a DVFS setting per-tile (affecting both cores and shared LLC). Choosing an appropriate DVFS setting per-tile is non-trivial as a single DVFS for all threads scheduled on the given tile might not be optimal for performance. Applications with higher sensitivity to DVFS changes are more likely to be affected by imposing a single DVFS setting per tile. Hence, we choose the DVFS setting so as to minimize the severity of the performance impact on the applications with high sensitivity to DVFS. If this setting over-provisions the per-tile power allocated by the GPM, we subsequently down-scale the core micro-architecture until the allocated tile budget is reached.

Shared LLC For cache adaptation, we use a flushing selective-way LLC implementation [4], i.e., a shared LLC per-tile in our setup. By controlling which ways are active, we can power-gate portions of the

(a) Multi-program workloads (SPEC CPU2006)

Workload	Description	Benchmarks
WL0	SPEC average	all 55 + 9 uniform random
WL1	Compute	8 compute bound, $\times$8
WL2	Mixed	8 compute + 8 memory, $\times$4
WL3	Memory	8 memory bound, $\times$8

(b) Multi-program multi-threaded workloads

Workload	Benchmarks	Input set	#Threads
NAS Parallel Benchmark suite			
NPB1	BT, CG, FT, MG	class A	16 each
SPEC OMPM 2001 suite			
OMPM	fma3d, swim, mgrid, applu, equake, apsi, gafort, wupwise	reference	8 each

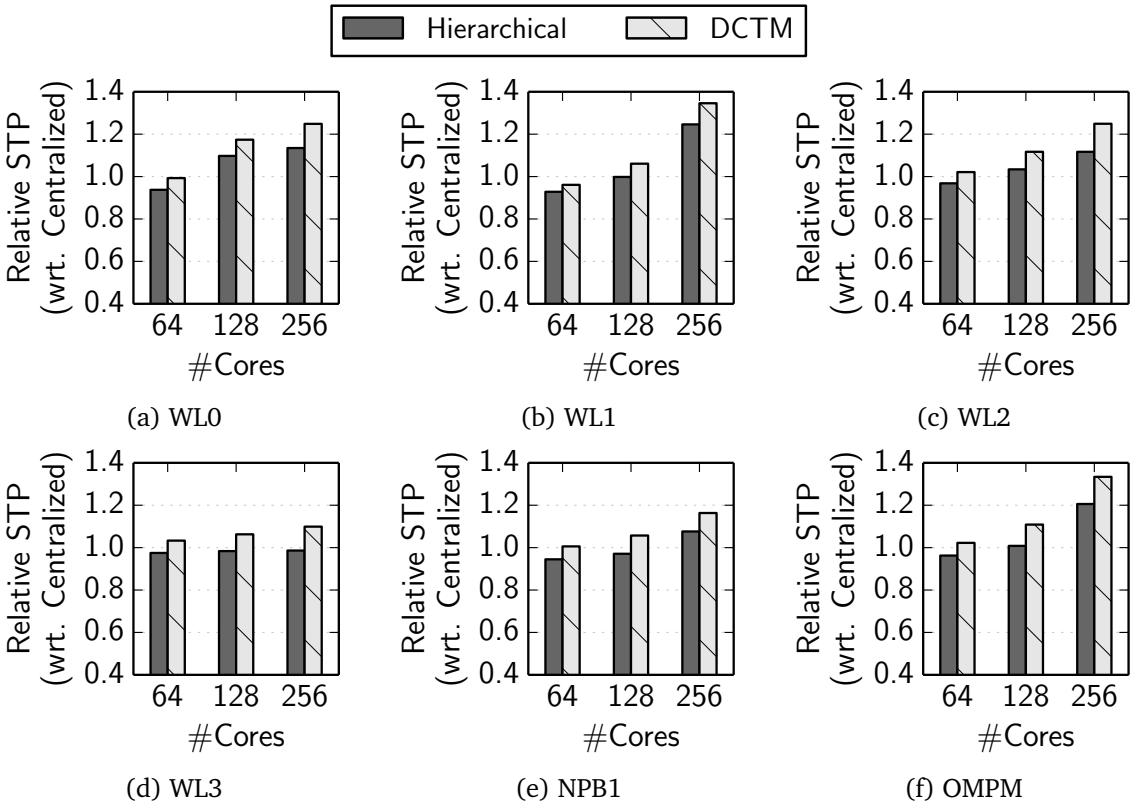
Table 4.3: Workloads.

cache to reduce its capacity and static power. We use selective ways (see also Table 4.1) because of its simple design — selective sets on the other hand require changes to the number of tag bits used [178]. By using the flushing cache policy when shrinking to a smaller number of ways (writing back dirty cache lines), we can turn off the corresponding cache ways sooner, reducing static power consumption of the cache. To estimate the effect of cache capacity changes, we use auxiliary tag directories (ATDs) [145], which estimate the miss rates (32 randomly selected sample sets) for different shared cache configurations. To project the performance impact of threads *sharing* the LLC, we create ATDs per core and annotate cache tags with a core identifier. This is only required for those sets that are part of the ATD's sample set.

4.5.3 Workloads

Multi-program Workloads We run a number of multi-program workloads composed of SPEC CPU2006 benchmarks; 29 programs in total, which along with all reference inputs leads to 55 benchmarks. We select representative simulation points of 750 million instructions each using PinPoints [134]. Four multi-program workloads with 64 benchmarks each are constructed by combining these 55 benchmarks as indicated in Table 5.3(a). We replicate each workload by 2$\times$ and 4$\times$ for the 128-core and 256-core setups, respectively. Each benchmark is pinned to a core unless mentioned otherwise. We run the simulation for 200 ms to keep total simulation time within feasible limits. When a benchmark completes before this time, it is restarted on the same core. We quantify weighted speedup [158] or system throughput (STP) [54] which quantifies the aggregate throughput achieved by all cores in the system.

Multi-program Multi-threaded Workloads We create workloads by combining multiple multi-threaded applications from the SPEC OMPM2001 [10] and NPB benchmark suites [14], see Table 5.3(b). For meaningful analysis, we use the *reference* input set for SPEC OMPM2001, and the *class A* input set for NPB. We construct two workloads, each running 64 threads in total: *NPB1* consists of four different NAS applications running concurrently with 16 threads each, while *OMPM* combines eight SPEC OMPM applications running 8 threads each. When running on the 128-core setup we replicate these workloads by 2$\times$, and by 4$\times$ for the 256-core setup. Execution of all multi-threaded applications in a workload begins after the last application has reached the region of interest (ROI). Again, we run each workload for 200 ms to keep total simulation time manageable.



(a) WL0 (b) WL1 (c) WL2

(d) WL3 (e) NPB1 (f) OMPM

Figure 4.5: STP (normalized to *Centralized*) for *Hierarchical* and *DCTM* at 60% power budget vs. core count.

4.6 Evaluation

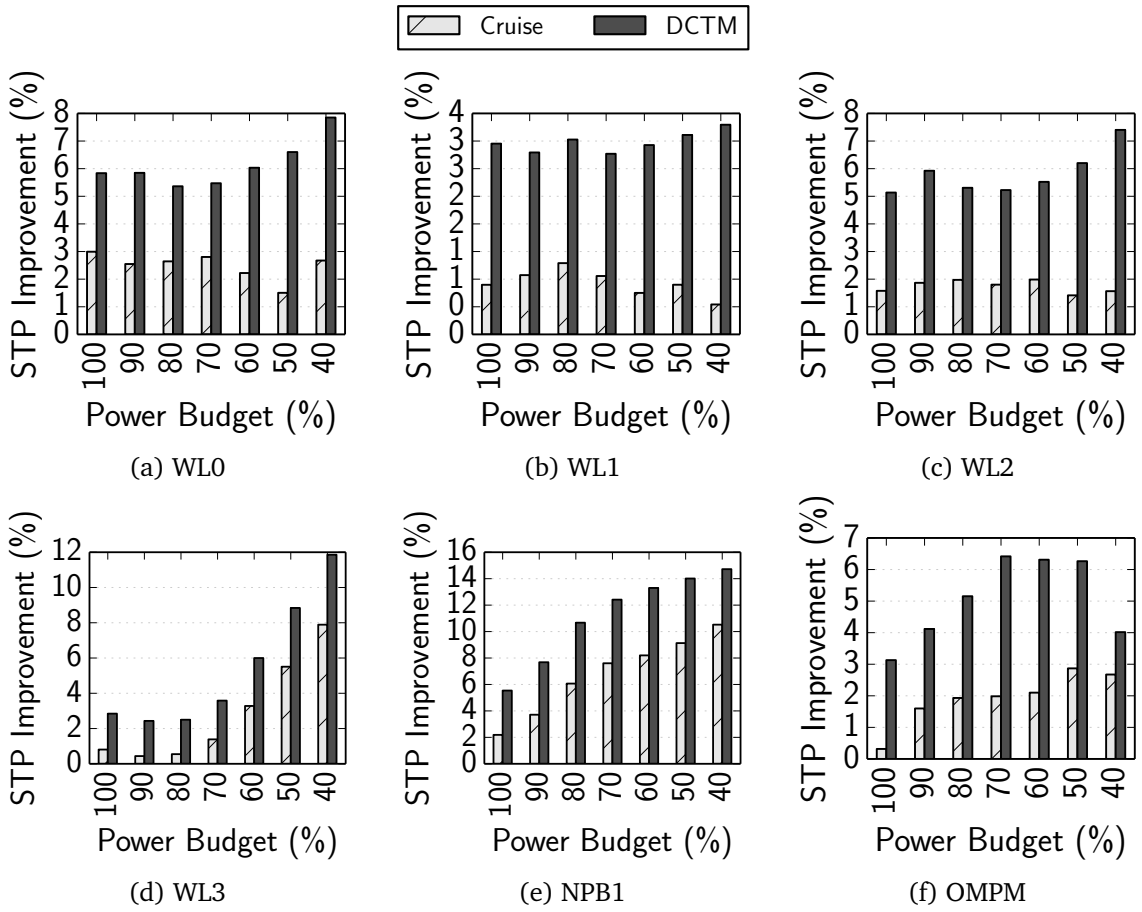
We now evaluate DCTM on our power-constrained tiled many-core architecture. Unless mentioned otherwise, results are obtained using fine-grained hardware adaptation at 1 ms intervals, while thread migration is performed at 20 ms intervals. Each experiment fixes the available power budget to a fraction of the chip's nominal power consumption (see TDP in Table 4.2). We quantify performance in terms of system throughput (STP), which includes power management overhead.

The evaluation is done in a number of steps. We first evaluate the scalability of two-tier hierarchical power management compared to centralized power management. We next compare DCTM against Cruise, demonstrating the importance of being frequency-aware. We then evaluate the importance of dynamic thread migration, followed by a number of sensitivity analyses with respect to the thread migration interval and power distribution.

4.6.1 Hierarchical vs. Centralized Power Management

We first evaluate the scalability of two-tier hierarchical power management versus a centralized approach. We consider the following power management policies: (i) *Centralized* which assumes centralized power management along with per-core DVFS; (ii) *Hierarchical* which is our two-tier hierarchical power manager, with random mapping of threads to tiles, and per-tile DVFS; and (iii) *DCTM* which is our two-tier hierarchical power manager that migrates threads across tiles in a DVFS and LLC aware manner.

Figure 4.5 quantifies relative STP (normalized to the *Centralized* approach) for the various workloads as



(a) WL0 (b) WL1 (c) WL2

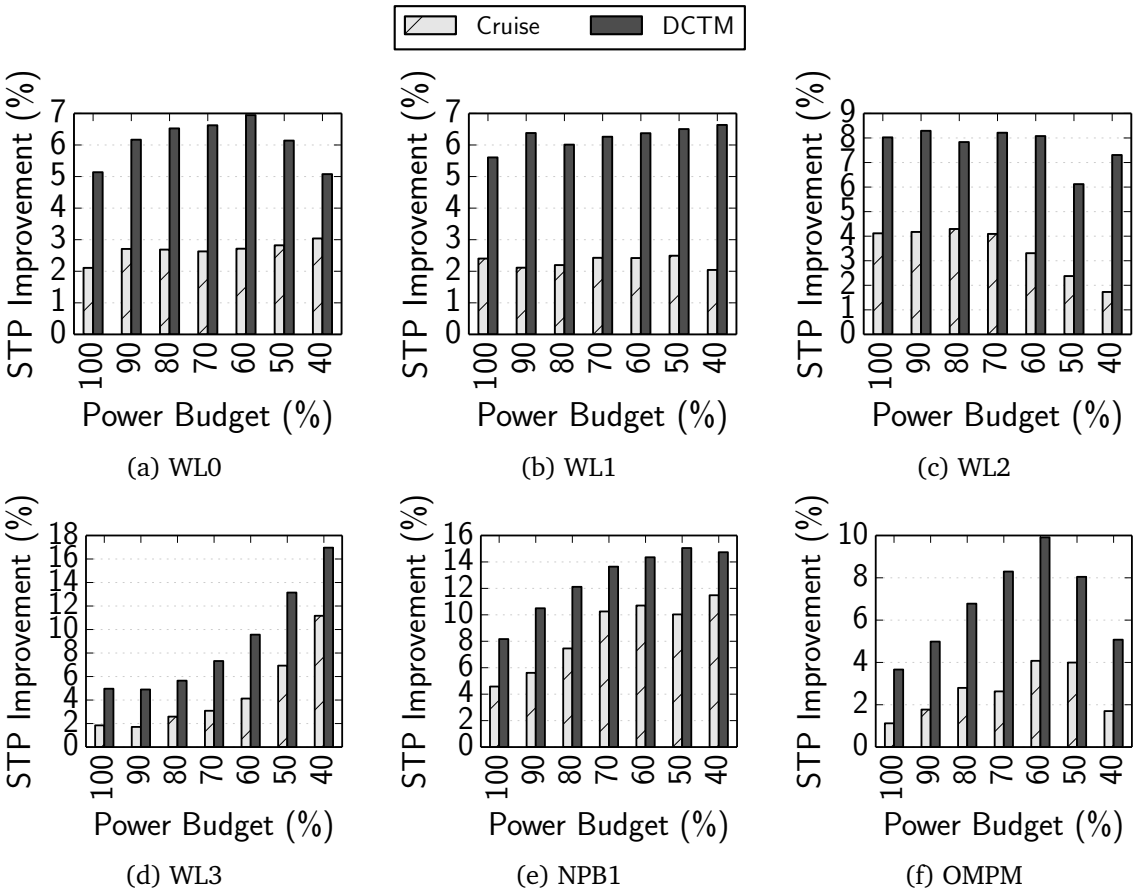
(d) WL3 (e) NPB1 (f) OMPM

Figure 4.6: STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 64-core setup.

a function of core count at a 60% power budget. The *Centralized* approach is quite effective at 64 cores. The overhead of the centralized power manager is limited, and the ability to exploit per-core DVFS yields a performance benefit over the two-tier *Hierarchical* approach with per-tile DVFS, by 7% on average. At larger core counts however, the overhead of the centralized power manager is not offset by the benefit from per-core DVFS, yielding a performance benefit for two-tier hierarchical power management, up to 24% for 256 cores (see *WL1*). The interesting insight here is that at large core counts, per-tile DVFS is in fact beneficial over per-core DVFS, which may seem counter-intuitive at first sight because there is less opportunity for fine-grain adaptation. The reason however is that per-tile DVFS facilitates a two-tier hierarchical power manager which incurs less overhead compared to a centralized power manager for a per-core DVFS architecture.

The results in Figure 4.5 also show that being able to migrate threads such that compatible threads co-execute per tile, as done using *DCTM*, yields a substantial performance benefit over random thread assignment with *Hierarchical*, see for example *WL1*: 32.4% for *DCTM* versus 24% for *Hierarchical*. We observe the performance benefit to be consistent across all workloads.

Overall, we find two-tier hierarchical power management, and DCTM to be beneficial across all workloads. The performance benefit seems to be proportional to the number of compute-intensive benchmarks in the workload, see for example *WL1* (compute-intensive) versus *WL2* (mixed) versus *WL3* (memory-intensive). The reason is that the power manager groups threads based on their sensitivity to LLC size and clock frequency, and allocates a larger fraction of the available power budget to tiles that benefit the most, which are typically the ones running compute-intensive benchmarks.



(a) WL0 (b) WL1 (c) WL2

(d) WL3 (e) NPB1 (f) OMPM

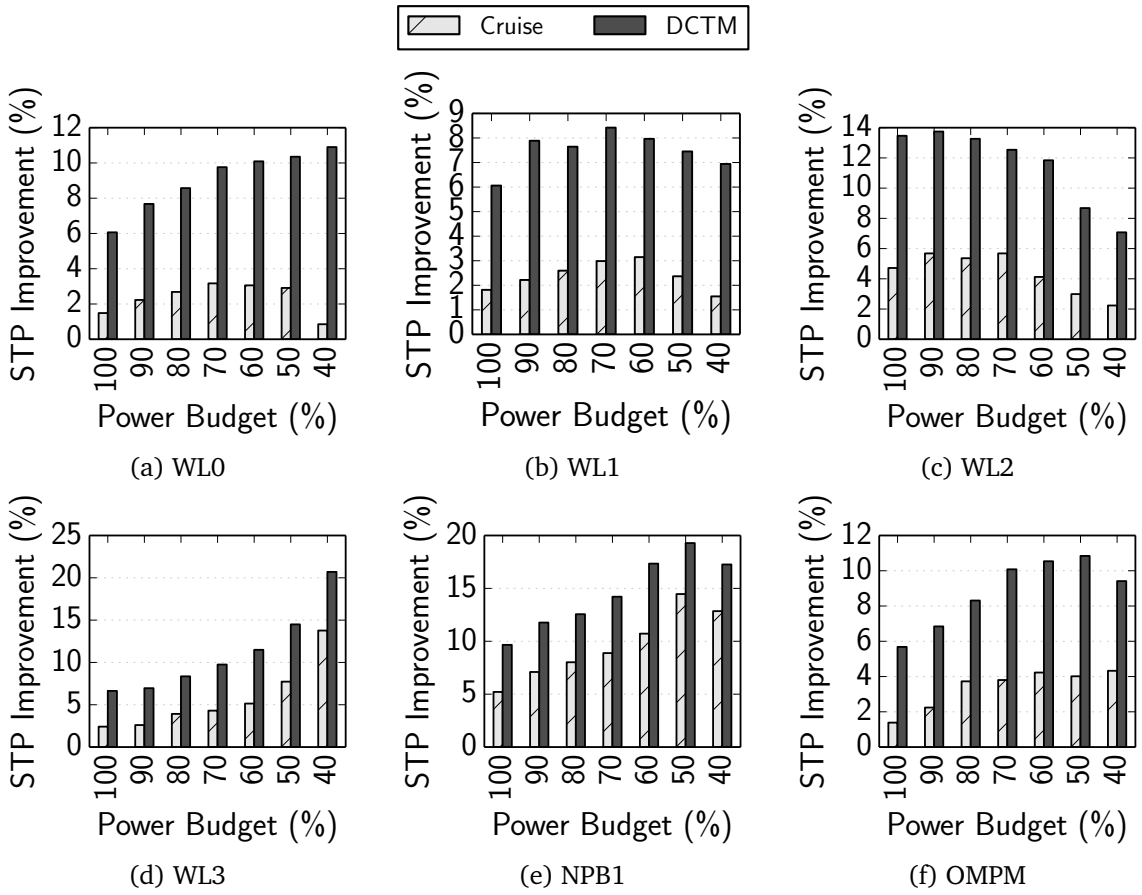
Figure 4.7: STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 128-core setup.

4.6.2 Two-Tier Approach: Performance vs. Power Budget

As we mentioned in Section 4.4.4, application's sensitivity to DVFS could provide better performance than just considering LLC sensitivity. To illustrate this, Figure 4.8 shows the STP improvement (as percentage) of a 256-core setup at different power budgets for *Cruise* and *DCTM*, relative to the *Hierarchical* performance. Both *Cruise* and *DCTM* employ a two-tier hierarchical power manager. *Hierarchical* power management shown as a baseline assumes random benchmark/thread placement on tiles.

The bottomline is that *DCTM* outperforms *Hierarchical* by 10% on average (across all workloads) and by up to 20%. *DCTM* outperforms DVFS-agnostic *Cruise* by 4.2% on average and by up to 12%.

There are a couple interesting trends to be observed for a number of individual workloads. For *WL0* (average SPEC CPU), *DCTM* shows an increasing trend at increasingly smaller power budgets. The reason is that *WL0* includes a wide range of applications with varying characteristics, which can be efficiently exploited using both DVFS and LLC sensitivities. For *WL1* (compute-intensive SPEC CPU), *DCTM* yields a consistent improvement over *Cruise*, but is limited by the available power budget. For *WL3* (memory-intensive SPEC CPU), we observe that both *DCTM* and *Cruise* are able to prevent excessive LLC trashing, which leads the STP improvement over *Hierarchical* to increase at smaller power budgets. However, by being DVFS-aware, *DCTM* still outperforms *Cruise* by 7% on average.



(a) WL0 (b) WL1 (c) WL2

(d) WL3 (e) NPB1 (f) OMPM

Figure 4.8: STP improvement (percentage) for *DCTM* and *Cruise* over *Hierarchical* for the 256-core setup.

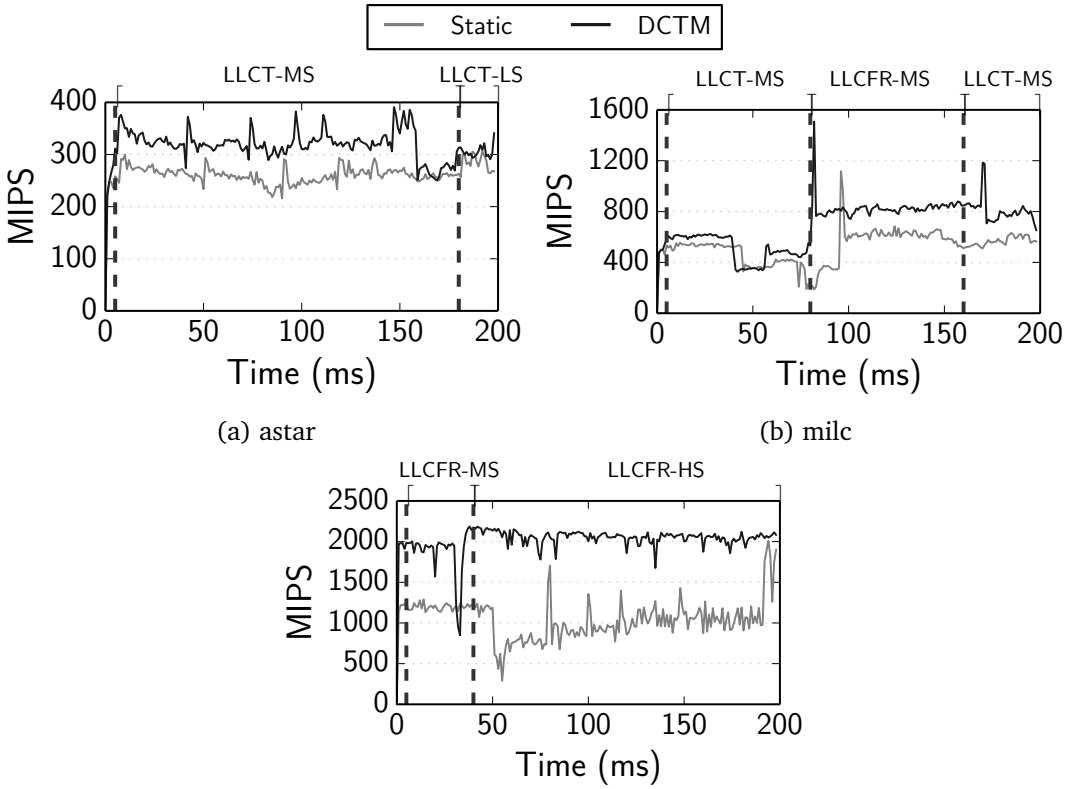
4.6.3 Static Assignment vs. Dynamic Migration

An alternative to performing on-line thread migration could be to statically select a thread schedule a priori based on known average application characteristics. However, in addition to the potential problem of jobs periodically entering and leaving the system, a single application exhibits phase behavior that may cause its classification to change over time. Using an average class leads to suboptimal scheduling, showing that on-line migration is a necessary component of our approach.

4.6.3.1 Workload Behavior Through Time

Before exploring the overall performance benefits of thread migration on the GALS architecture, we illustrate the classification of SPEC CPU2006 application traces based on LLC and DVFS sensitivity. *6* applications exhibit *CCF* behavior, *18* applications exhibit LLC access patterns that fall in LLCT category and *40* applications are classified in LLCFR category. These classifications are used in thread migration scheme when threads are moved to different tiles based on LLC characteristics only (*Cruise*). In the *DCTM* approach, the application classifications are based on both LLC-aware and DVFS-sensitivity as described in Section 4.4. All *CCF* applications are also *HS* leading to *6* applications in the *CCF-HS* category. *11* application traces are categorized as *LLCT-LS* and *8* as *LLCT-MS*. The *LLCFR-MS* category consist of *8* applications and the remaining *32* are classified as *LLCFR-HS* category (also see Figure 4.3).

We ran simulation by pre-classifying the applications traces based on above mentioned classification and



(a) astar

(b) milc

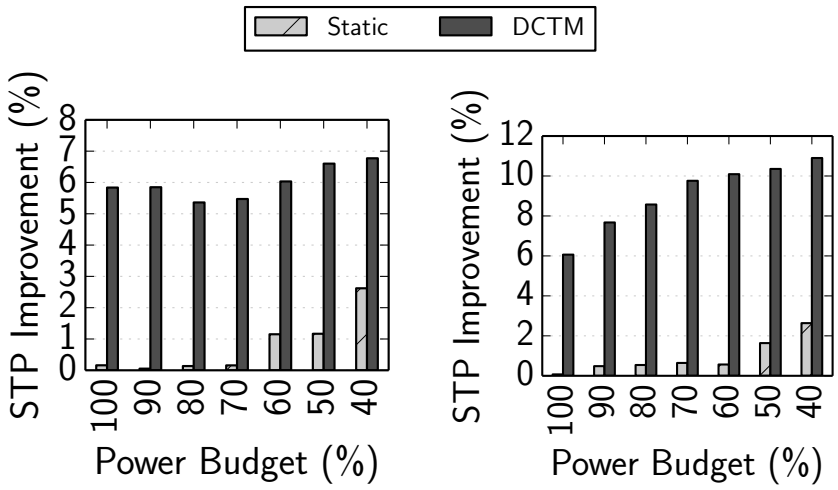
(c) perlbench

Figure 4.9: *Static* assignment and *DCTM* through time for selected SPEC CPU2006 applications on 64-core setup at 80% Power Budget.

rules in Section 4.4 without thread migration. Results are shown in Figure 4.9 for a selection of applications, both using static classification (*Static*) and when performing runtime thread migration (*DCTM*).

LLCT with LS and MS phase The number of application traces with LLCT cache behavior show runtime DVFS sensitivity as LS or MS based on application behavior. Based on entire trace estimates, these application can be classified as LLCT-LS, but it can be observed for these application traces like astar, exhibit clear sets of phase behavior. Figure 4.9(a) plots the MIPS though time for application trace astar at 80% power budget (64-core setup). It can be observed that astar initially shows moderate DVFS sensitivity followed by a phase of low sensitivity to DVFS variations. Application traces with these phases benefit from *DCTM* due to runtime re-scheduling based on DVFS-sensitivity unlike *Static* co-scheduling. The experimental logs suggest that such application traces with distinct phase behavior switches from LLCT-MS to LLCT-LS phase. Being aware on these phases, the *DCTM* is able to initially co-schedule astar with LLCT-MS or CCF-HS applications (Section 4.4, Rule 2) and provide better MIPS. The *Static* approach, being unaware of phases, does co-schedule this application trace with other LLCT-LS threads. Hence we observe lower performance for a significant fraction of time in Figure 4.9(a).

LLCT-MS with LLCFR-MS phases Figure 4.9(b) plots the MIPS though time for application trace milc at 80% power budget (64-core setup). When averaging characteristics over the entire application trace, this workload is classified as LLCT with moderate sensitivity to DVFS. The *Static* approach co-schedules these application traces with other LLCT-MS or CCF applications based on co-scheduling policy in Section 4.4. Although the generic classification is LLCT (streaming behavior), milc shows phases in execution where



(a) 64 cores (b) 256 cores

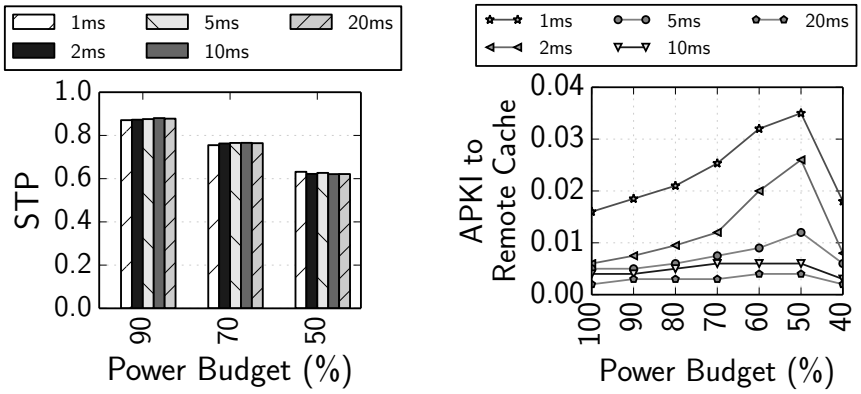
Figure 4.10: *Static* versus *DCTM* relative to *Hierarchical* for *WL0*.

it can be classified as LLCFR due to a reduced working set which does fit into the LLC. During this phase, if the application remains co-scheduled with an LLCT application which will cause milc's working set to be evicted, performance will suffer compared to a situation where the LLCT thread is migrated away in favor of another LLCFR or a CCF application. Application traces with variations in cache access behavior can be benefit if the cache access patterns are evaluated at runtime. Unlike the *Static* approach, *DCTM* is able to observe changes in cache access behavior at runtime and re-schedule accordingly, leading to higher performance of milc during its LLCFR phase.

LLCFR-MS with LLCFR-HS phases A number of LLCFR applications show runtime phases where the accesses to cache are reduced sufficiently (not as low as CCF), making them increasingly compute bound. Figure 4.9(c) plots the MIPS though time for one such application, perlbench, at 80% power budget on a 64-core setup using both *Static* and *DCTM*. After appropriately 80 million instructions, the computation-bound phase is observed and based on *DCTM*, perlbench is re-classified as highly sensitivity to DVFS variation. From that point on, *DCTM* co-schedules it with other LLCFR-HS or CCF-HS applications. The GPM can subsequently redistribute power to these tiles as these application traces are able to provide better power-performance ratio than other tiles, on which groups of LS applications have been clustered and which can now safely reduce power without affecting performance. This can be observed in Figure 4.9(c), where *DCTM* is able to provide significantly higher performance than *Static*. Unable to change the application trace classification at runtime, the *Static* co-schedules this application with other LLCFR-MS. Hence the *Static* approach cannot increase the DVFS setting for perlbench while remaining inside the total power budget.

4.6.3.2 System Performance

We now evaluate the performance benefit of *DCTM* against static classification in a more systematic way. Static classification follows the same classification and scheduling rules as *DCTM*; the only difference is that static classification does so based on the application's average execution behavior. Figure 4.10 shows the performance improvement of static assignment and *DCTM* over *Hierarchical*. At moderate to low power budgets, static assignment provides some improvement over random assignment as the restricted power budget requires significant reductions in both DVFS and LLC size, which can be tolerated better



(a) STP vs power budget

(b) Remote cache accesses

Figure 4.11: Sensitivity to DCTM's thread migration interval for *WL0* on 64 cores and L2 invalidates: *Hierarchical* vs *DCTM* for single-thread *WL0* workload.

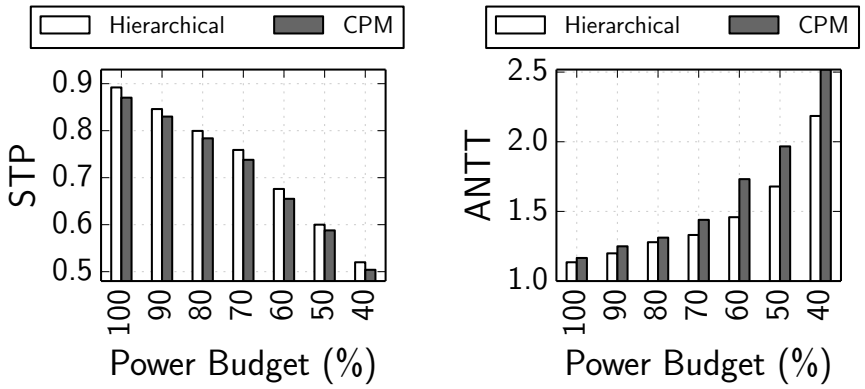
when compatible applications are co-scheduled. At higher power budgets, however, the architecture operates much closer to its full configuration, and static assignment fails to provide a significant advantage. In contrast, *DCTM* is able to exploit phase behavior in the applications, and can obtain the optimum co-schedule at each point in time. This gives *DCTM* a significant margin over both *Hierarchical* and static assignment, showing that runtime migration can greatly improve power efficiency of large many-core systems.

4.6.4 Sensitivity to Migration Interval

Previous experiments considered a coarse-grain thread migration interval of 20 ms, while hardware adaptation was performed every millisecond. This is consistent with an implementation where adaptations are performed in a hardware power manager, while thread migrations is done by the operating system (with input from the power manager and/or performance counters to do the classification).

In Figure 4.11, we vary the *DCTM* thread migration interval between 1 and 20 ms, while leaving the power-aware hardware adaptation interval fixed at 1 ms. Migration itself is assumed to take 1,000 cycles to transfer register state and restart execution at a remote core, in addition to potential cold misses that transfer the thread's working set to the local caches (using the standard coherency protocol, which our simulations model in detail). Figure 4.11 plots STP (relative to full configuration), when running the SPEC average multi-program workload *WL0* on a 64-core system at various power settings. No significant difference in performance is observed, showing that a 20 ms migration interval is sufficient. It is therefore possible to implement this layer in the operating system: hardware thread migration is not required and it is even possible to spend a significant amount of time and effort to compute the best schedule.

In contrast, doing thread migration too frequently can in fact be harmful, as Figure 4.11(b) illustrates. For short migration intervals (below 5 ms), the amount of cache-to-cache transfers increases significantly, showing that working sets frequently have to be transferred across the chip, trailing the migrating threads. These cache-to-cache transfers cause both a reduction in thread performance and consume additional power, which has to be amortized by the improved schedule.



(a) STP (b) ANTT

Figure 4.12: Fine-grain (*Hierarchical*) versus coarse-grain (*CPM*) power redistribution for the two-tier hierarchical power manager for *WL0* on 64-core setup.

4.6.5 Fine-grained Power Redistribution

Our *Two-Tier Hierarchy Approach* reallocates power between the tiles in each hardware adaptation interval (1 ms timescale), in addition to making adaptations local to each tile. An alternative would be to run the global component (GPM) less frequently, while running TPM at a small timescale. Such approach is in fact proposed by Coordinated Power Management (CPM) [128], where a fast per-tile DVFS controller performs local optimization constrained to each tile's power budget, and a global manager redistributes power across the tiles every ten adaptation intervals. We implement a similar design with 1 ms and 10 ms time scales for the local (TPM) and global (GPM) power management, respectively. Figure 4.12 shows STP (relative to full configuration) as a function of the available power. Our *Hierarchical* design (GPM and TPM adaptation at 1 ms timescale) outperforms CPM by 1–8% in STP, in addition to providing better fairness between the running threads (measured in average normalized turnaround time, ANTT [54]). This is because in CPM, each tile needs to manage power over- and undershoots using a fixed power budget over a 10 ms interval. In contrast, when doing global adaptation at 1 ms time scales, power can be redistributed across tiles much faster, allowing compute-bound threads to borrow power from memory-bound threads running on different tiles within just 1 ms, so the system can respond much more quickly to changes in workload behavior.

4.7 Related Work

4.7.1 Micro-architecture Adaptation

A variety of prior work has explored techniques to improve power-efficiency by adapting micro-architecture structures on a per core basis. Some proposals adapt the instruction window [12] and the issue logic [59] to provide greater power/energy efficiency while showing a small reduction in application performance. ForwardFlow core [63] is proposed as a way to trade off core performance for power. Albonesi [4] and Yang et al. [178] evaluated shutting down portions of the cache, either a number of ways or a combination of ways and sets for improved energy efficiency or to trade off performance for power and energy. Eckert et al. [51] combined drowsy caches with front-end pipeline gating and demonstrate better performance-power scaling than dynamic frequency scaling, and even DVFS in some cases. Although their work shows that one can reconfigure the system to perform better than DVFS, they do not perform runtime optimizations of large many-cores in power-constrained environments. Finally,

Dubach et al. [49] used machine-learning models (trained using profiling) to perform online adaptation of a single core at a time.

4.7.2 Centralized Dynamic Power Management

Several prior works explore centralized dynamic power management. For example, Isci et al. [80] investigated a global power controller to determine different per-core DVFS values to maximize chip-wide MIPS. Teodorescu and Torrellas [167] proposed variation-aware power-management DVFS algorithms for application scheduling on a CMP to save power or improve throughput at a given power budget. Deng et al. [42] proposed CoScale, a mechanism that deals with co-optimizing DVFS settings for both the CPU and DRAM. Other proposals used machine learning and neural networks to perform global DVFS with per-core adaptation [84] or global resource allocation [22]. Meng et al. [123] proposed DVFS adaptation along with cache adaptation for a 4-core system. Chrysso (see Chapter 3) dynamically adjusts the capabilities of an out-of-order core, private cache and per-core DVFS at fine-grained time slice (10 ms) using simple analytical models and a centralized power manager to improve performance under given power budgets. Finally, Flicker [139] dynamically adjusts the capabilities of an out-of-order core at coarse-grained time slice (100 ms) using sampling-based global genetic algorithm to improve performance compared to core gating at moderate power budgets.

4.7.3 Tiled Architecture and Hierarchical Power Manager

In RCS [62], the authors proposed mechanisms to uniformly change core resources with the number of cores (up to 12) to exploit application variability at a fixed power budget. The proposed scheme uses SVM-based machine-learning mechanisms to obtain the number of active cores (with corresponding micro-architectural variation) for each interval. PEPON [154] used 10 DVFS adaptations for core and NoC along with selective-way resizing of a single shared LLC to provide feasible working configuration till moderate power budgets. Other proposals [61, 128] used the concept of two-level power management schemes, viz. master-slave and global-local, respectively. Mishra et al. [128] used one of the 10 DVFS values per island (2/4 cores per island) under the given power budget. The mechanism uses a 2-level power manager — GPM-LPIC (digital controller per LPIC) called every 25/50 ms and 2.5/5 ms, respectively. Prior work has explored power management techniques on network-on-chip [170] to provide significant reduction in power dissipation of NoCs. A hierarchical control-theory based power manager [131] employs multiple PID controllers (one for each cluster and one for each application) in a synergistic fashion and manages to achieve optimal power-performance efficiency while respecting the TDP budget. This approach has poor scalability with increasing number of clusters and price-theory based demand-supply approach. Additionally, the coarse-grain power management could ensue thermal-throttling due to instantaneous power over-shoots.

To the best of our knowledge, none of the above works have evaluated three-way micro-architectural adaptation along with a thread migration layer for optimal shared resource utilization using a hierarchical power manager on a power-constrained tiled many-core architecture.

4.8 Summary

An integrated and scalable many-core power management is clearly needed as we move towards increasingly tighter power budgets. In this work, we leverage a two-tier hierarchical power manager due to its low overhead and high scalability on a tiled many-core architecture with shared LLC and per-tile DVFS at fine-grain time slices. We use (i) analytical performance and power models for the shared architecture

and its adaptation, and (ii) we distribute power across tiles using GPM and then within a tile (in parallel across all tiles). We observe that thread scheduling is essential in such an architecture to account for thread sensitivity towards shared resources. We leverage DVFS and cache-aware thread migration (DCTM) to ensure optimum per-tile co-scheduling of compatible threads at runtime over the two-tier hierarchical power manager. Based on our evaluations, we show that DCTM outperforms Cruise [83] by 4.2% on average (and up to 12%) for both multi-program and multi-threaded workloads. Compared to a centralized power manager, DCTM improves performance by 10% on average (and up to 20%) while using 4$\times$ less on-chip voltage regulators.

Part II

Reducing Die-Stack DRAM Hit Latency

Chapter 5
Tag Caching for Gigascale Die-Stacked DRAM Caches: A High-Performance Many-Core Perspective

Whereas core count in high-performance processors continues to grow, off-package memory bandwidth has been growing at a much slower pace. This has led to the emergence of large amounts of on-package memory, often implemented as die-stacked gigascale DRAM Caches, providing much higher memory bandwidth at lower energy. Yet, managing these large DRAM Caches has proven challenging, as it is unfeasible to keep all cache tags in fast, on-chip SRAM storage. Practical designs therefore suffer from high latencies as tags need to be fetched from DRAM.

In this chapter, we analyze the hit and miss latencies of several recently proposed DRAM Cache designs, and we comprehensively explore how associativity affects both latency and hit rate in the context of high-performance many-core processor architectures. We then propose Tag Cache, an on-chip distributed structure that caches DRAM Cache tags, to reduce hit access latency.

5.1 Introduction

Off-chip memory accesses have become a performance-limiting factor with increasing core count on a single die, from both a latency and bandwidth perspective. The off-chip DRAM-DIMMs have not scaled to match the demands of modern servers leading to the well-known bandwidth wall problem [147]. Recent memory technologies such as 3D-stacked DRAM [23] have emerged as a promising alternative wherein DRAM memory dies are stacked on top of, or next to, a processor die using high-bandwidth through-silicon-vias (TSVs) [100]. A typical 3D stacked DRAM has multiple (4–8) data channels that can access many banks (8–16 per channel). This technology offers gigascale DRAM capacity at very high bandwidth and low energy, alleviating the off-chip memory wall constraint. For instance, AMD's recent GPU Radeon Fury X replaces all GDDR5 memory with 4 GB of on-package high-bandwidth memory (HBM) [53], while Intel's Knights Landing uses 8–16 GB of HMB configured as a cache in front of traditional off-package DDR4 memory [7, 74, 78, 159].

Designing gigascale DRAM as a hardware-managed cache faces several challenges, including designing a tag storage of several tens of megabytes. For example, a 1 GB cache with 64 B blocks would need 64 MB of storage for the tag array. Provisioning such a large tag array using on-chip SRAM is impractical. Increasing cache block size to reduce the tag overhead is generally not a good solution as it increases pressure on an already saturated off-chip memory link, while wasting bandwidth when workloads have low spatial locality. Sectored caches [150] can avoid the bandwidth wastage and tag area cost, but often lead to poor performance due to higher miss rates [60, 172]. Tags therefore need to be stored in DRAM,

yet a naive approach will significantly increase access latency as each request now involves two DRAM accesses, one for the tags and, in case of a hit, one for the data.

Recent approaches address the fundamental issue of high-latency DRAM Cache tag accesses by storing the tags along with the data in the DRAM array [116, 144]. Prior work proposing the Tags-in-DRAM concept include the Loh-Hill Cache [116] and the Alloy Cache [144]. Both these approaches mitigate the high latency access issue by avoiding DRAM Cache accesses on misses, either by an additional tracking structure on the logic die such as Loh-Hill Cache's MissMap table or through prediction as in the Alloy Cache. In particular, Alloy Cache organizes tags and data together to form a Tag And Data (TAD) entry in a direct-mapped DRAM Cache, and transfers a TAD per request to avoid the tag serialization penalty. Alloy Cache thus optimizes for hit latency instead of hit rate. However, it suffers from a significant amount of conflict misses as core count increases. On the other hand, the Loh-Hill Cache preserves a high level of associativity for its stacked DRAM Cache — 29 ways in a single DRAM row buffer to leverage row buffer locality, hence it is likely to be a more advantageous design in the context of many-core processors. Yet reading the tags from DRAM in a separate column access still incurs a significant latency cost. Moreover, existing tracking solutions use a centralized entity to either bypass the DRAM Cache on misses, as is the case for the MissMap [116], or store tag information for faster access, see for example [60, 75, 181]. Unfortunately, these centralized structures may become a bottleneck with increased core count.

To combine high associativity with low hit latency, we propose Tag Cache, an on-die distributed structure that caches tags that otherwise need to be read from the DRAM Cache. The Tag Cache utilizes the DRAM Cache replacement policy to store the tag information of the most recently used DRAM Cache ways to exploit temporal locality and increase the number of low-latency DRAM Cache hits. Being distributed in nature, the Tag Cache not only avoids being a bottleneck in a many-core processor, it also prevents tag information from being polluted by memory-intensive applications when co-running with other applications. We make the following contributions:

- We explore different variations of DRAM Caches with conventional cache block size on a tiled many-core processor architecture.
- We propose Tag Cache, a distributed mechanism to alleviate the hit latency of multi-way set-associative DRAM Caches.
- We perform a sensitivity analysis on the distributed Tag Cache structure with respect to caching associativity in terms of hit rate, overall memory access latency, performance and energy.
- We provide a comprehensive evaluation of Tag Cache using multi-program and multi-threaded workloads on a tiled many-core processor architecture and compare it against prior solutions and an ideal Tags-in-DRAM Cache (with zero-latency Oracle Tag).

We evaluate the distributed Tag Cache on a tiled many-core processor with 64 cores (16 tiles, 4 cores per tile) with a 2 GB on-die stacked DRAM Cache running multi-programmed single-threaded and multi-programmed multi-threaded workloads. When combined with the Loh-Hill Cache, we find that it provides an on average 6.4% performance increase over the Loh-Hill Cache with MissMap, and 20.6% higher performance compared to the Alloy Cache (with an ideal memory access predictor). For a 2 GB Tags-in-DRAM Cache, the distributed Tag Cache uses 1-way per shared-per-tile L2/LLC cache (2 MB in total), while the MissMap table needs approximately 8 MB — a $4\times$ reduction in occupied LLC space. The Loh-Hill Cache with a distributed Tag Cache also enables 20.6% lower energy compared to the MissMap-enabled Loh-Hill Cache. Compared to ATCache [75], Tag Cache provides on average 2.6% higher performance with 7.5% lower energy consumption.

5.2 Background and Motivation

In pursuit of mitigating the memory wall issues in the many-core era, processor designers face a key design decision on how to optimally utilize the large capacity in heterogeneous memory systems. An

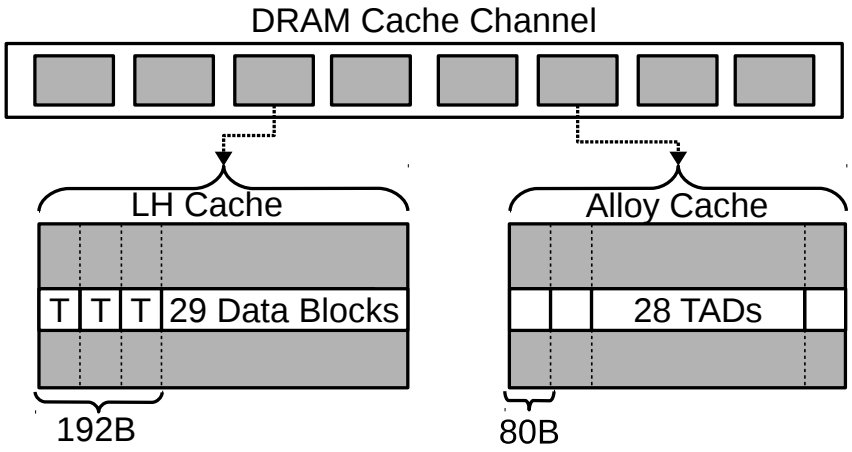


Figure 5.1: Tags-in-DRAM Cache Design: Loh-Hill Cache (left) and Alloy Cache (right).

attractive proposition is to use high-bandwidth memory as a cache, also referred to as DRAM Cache, as it can be designed to be transparent to the software stack. Software-agnostic high-performance DRAM Caches have received considerable attention over the past few years. For the 3D-stacked DRAM Cache, previous works have assumed that the DRAM array has a lower latency than off-chip memory. (Some prior proposals use the 3D-stacked DRAM Cache timing latencies as approximately half of that compared to conventional off-chip DRAM [75, 116, 144], whereas other previous works have assumed ratios of 1:3 [44] and 1:4 [56, 181].) In addition, stacked DRAM also supports more channels, more banks and wider buses per channel [100], providing increased memory bandwidth.

5.2.1 Impact of Associativity

Some prior works have advocated direct-mapped DRAM Caches to reduce the hit latency [40, 144], while others have advocated multi-way set-associative designs [67, 75, 90, 116, 181]. Direct-mapped caches have a fast hit time compared to set-associative caches, due to the restricted look-ups in tag and data. However, they incur more conflict misses compared to a set-associative cache, because of various addresses competing for the same cache line. For example, Zhao et al. [181] report that an 8-core CMP setup with a 16-way DRAM Cache reduces the miss rate by 20–40% over a direct-mapped DRAM Cache. One of the objectives of this work is to explore to what extent DRAM Cache associativity affects performance in a many-core processor architecture setting.

5.2.2 Cache Designs with Tags-in-DRAM

The Loh-Hill Cache [116] proposes a set-associative Tags-in-DRAM Cache. The Loh-Hill Cache implements a 29-way cache in a single DRAM row buffer, as shown in Figure 5.1 (left). The authors propose storing the tags for the 29 cache ways in the first three cache ways of the 2 KB row buffer. To service a cache hit, the Loh-Hill Cache first reads and checks the tags using Compound Access Scheduling to access both the tag and data via multiple column accesses once the row has been activated. Upon a tag match, the data is read from the row buffer and sent to the respective core. This proposal targets improving cache hit rate using higher associativity at the expense of increased hit latency.

Qureshi and Loh propose the direct-mapped Alloy Cache [144] to tackle the hit latency issue, see Figure 5.1 (right). Alloy Cache targets optimizing both cache access latency and bandwidth by integrating the tag and data to form a Tag And Data (TAD) entry. This proposal stores 28 TAD entries on a 2 KB row buffer. To service a cache hit, the Alloy Cache reads the TAD entry from the DRAM Cache row buffer and

checks for a tag match. If the tags match, the associated data entry within the TAD is forwarded to the requesting core. The Alloy Cache improves hit latency at the expense of a lower hit rate.

5.2.3 Latency Mitigation Methods

To optimize the overall DRAM Cache access latency, Loh and Hill [116] propose the MissMap table, which keeps track of all blocks residing in the DRAM Cache. Before accessing the DRAM Cache, they first access the MissMap, which allows them to bypass the DRAM Cache and directly access off-chip memory upon on a MissMap (and DRAM Cache) miss. The MissMap lookup provides full coverage of the DRAM Cache, e.g., 2 MB to track 650 MB of DRAM Cache. The MissMap table is stored in the LLC, thereby reducing the effective LLC size. Qureshi and Loh [144] propose the Memory Access Predictor (MAP) table, 96 B per core, to predict whether an LLC miss is likely to result in a DRAM Cache hit or off-chip memory access. Chou et al. [40] further reduce bandwidth consumption in the Alloy Cache by using structures to store neighboring tags, and a bypass mechanism on an 8-core setup (collective size of 19.2 KB). ATCache [75] leverages the Loh-Hill Cache architecture and proposes a small SRAM cache that stores the DRAM Cache tags. To keep the size of the ATCache small, a predefined caching ratio of 256 is used to cache 1 K DRAM Cache sets — e.g., for a 256 MB DRAM Cache, the ATCache uses 46 KB. The size of the ATCache increases proportionally to the number of sets in the DRAM Cache. Moreover, ATCache suffers from lower hit rate as the associativity of ATCache tags is the same as for the DRAM Cache. Tag Tables [60] use 'base-plus-offset' encoding to compact on-chip DRAM Cache tag arrays. Tag Tables occupy a portion (as cache ways) of the L3 (LLC) cache using a set-dueling policy, thereby limiting the effective capacity of the LLC.

Most of the above proposed structures are designed to bypass the DRAM Cache in case of a miss, or reduce the access latency by either storing partial tags or full tags on-chip, while focusing on a multi-core setup with a relatively small number of cores (4–8 cores). In most proposals, such as MissMap, ATCache, and Tag Tables, the size of the on-die tag-storage structures is proportional to the size of the DRAM Cache. And, in addition, these structures are centralized. This leads to a design that may be feasibly implemented for a small to moderately sized DRAM Cache in a multi-core architecture with a relatively small number of cores. However, in a many-core architecture — 64 cores and beyond with gigascale die-stacked DRAM Cache — these centralized structures would not only suffer from scalability issues but also act as a source of bottleneck and contention for all memory requests going to the die-stacked DRAM Cache. Moreover, many-core processors also lack a large shared LLC, the location where structures such as MissMap and Tag Tables reside. This calls for a distributed approach with minimal resource utilization.

5.2.4 Ideal Tags-in-DRAM Cache

Before introducing our solution, we first discuss an ideal Tags-in-DRAM Cache organization. An ideal Tags-in-DRAM Cache design would combine the best features of both the Alloy Cache and Loh-Hill Cache. The memory requests would incur the latency equivalent to the time taken to access the data block only from the DRAM Cache row buffer (i.e., closed-page access latency), which is similar to a direct-mapped Alloy Cache. On the other hand, the ideal Tags-in-DRAM Cache would provide the hit rate of a multi-way associative Loh-Hill Cache.

In practice, creating an ideal Tags-in-DRAM Cache (Loh-Hill Cache with zero-latency Oracle Tag) is obviously unfeasible. For a gigascale DRAM Cache, a large SRAM tag array (several MBs) would be needed, which eventually would suffer from high access latency overhead due to its size. The practical alternative is to create a small but fast (few cycles) tag lookup structure that avoids the expensive DRAM Cache tag lookup operation. We propose a distributed tag caching mechanism in the next section as the middle ground between the multi-way Tags-in-DRAM Cache and an ideal Tags-in-DRAM Cache (multi-way set-associative with Oracle Tag) — our contribution in this work.

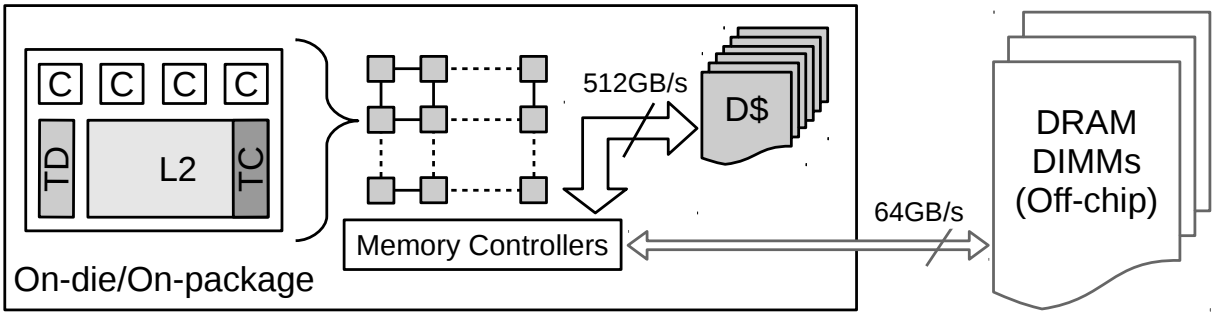


Figure 5.2: Tiled Many-core Architecture with an on-package DRAM Cache and an on-die distributed Tag Cache.

5.3 Tag Cache

We propose an on-chip SRAM-based Tag Cache structure with the main objective to cache tag information and avoid expensive tag lookup operations in the DRAM Cache, thereby reducing the hit access latency for multi-way Tags-in-DRAM Caches. The organization of the Tag Cache is governed by the following objectives: (i) Tag Cache is a distributed structure in order not to be a source of hot-spot contention in many-core architectures; (ii) Tag Cache uses existing SRAM-based cache resources without incurring additional area overhead; and (iii) Tag Cache exploits temporal locality for providing *near-zero* tag lookup latency for DRAM Caches.

To avoid the Tag Cache from being a single source of contention in a many-core processor architecture (first objective), we use a distributed Tag Cache array, on a per-tile basis. This is similar to the idea used in the distributed tag directory (TD) implementation seen in existing tile-based many-core processors like Intel Xeon Phi [159]. Each segment of the distributed Tag Cache is co-located with the shared-per-tile L2 Cache (LLC) in a tile-based many-core architecture, see Figure 5.2. Each segment of the distributed Tag Cache is designed to handle a segment of the DRAM Cache capacity. The segmented access to the DRAM Cache limits the unwanted tag misses due to tag pollution caused by applications' memory patterns (especially cache trashing applications).

To keep the area of the chip constant (second objective), we store each segment of the distributed Tag Cache in the respective L2 (LLC). Because the Tag Cache stores tag information (6 B for tag information plus additional bits — described later in this section), it becomes unviable to store all the tag information in L2. We employ way partitioning in L2 — 15 ways for data blocks and 1 way for Tag Cache entries — because of its simple hardware design. In contrast, using selective sets for storing tag information would require changing the number of L2 tag bits [178] — this would lead to higher hardware complexity.

Owing to space constraints, the Tag Cache selectively stores tag information by leveraging the workload's memory access patterns (third objective). Leveraging the memory replacement policy of the (multi-way) DRAM Cache, it can be expected that a significant fraction of DRAM Cache hits occur at the most recently used (MRU) ways. This observation is used to populate the Tag Cache structure to store the tag information pertaining to data blocks in the most recently accessed ways in the DRAM Cache. The MRU way information can be obtained from the tree-based or state-based MRU tracking mechanism used by the replacement policy in the DRAM Cache. The MRU way information is then passed along when the data is sent back from the DRAM Cache to the memory controller. To keep track of the MRU way information of the DRAM Cache data block, each Tag Cache entry thus needs to store 5 bits as the DRAM Cache way identifier (5 bits to identify up to 32 ways in the DRAM Cache). Considering the 6 bytes for the DRAM Cache tag along with the above mentioned 5 bits, the size of each Tag Cache entry would be 6 bytes and 5 bits; this means that within a 64-byte cache block size, we can keep track of 8 MRU DRAM

Cache ways. Maintaining 8 MRU DRAM Cache ways in the Tag Cache effectively only requires 53 bytes of storage out of the 64-byte cache block; the remaining bytes are used to keep track of the MRU positions of the respective tags, as we will describe later.

Tag Cache accesses can easily be done in parallel with the LLC tag accesses (speculating on an LLC miss) because the Tag Cache, the LLC slices, and the tag directories are all distributed in the same fashion. By doing so, the Tag Cache structure would not add any additional latency. Note though that in the evaluation in this work we consider serialized Tag Cache accesses, i.e., we first access the LLC, and upon a miss, we then access the Tag Cache — this is the worst operational condition to report conservative performance numbers.

Finally, it is important to note that Tag Cache not only reduces the hit latency of a multi-way DRAM Cache. It also reduces energy consumption by avoiding the first CAS operation related to a tag lookup on a DRAM Cache hit.

5.3.1 Tag Cache Operations

Having described Tag Cache based on its key features, we now describe the operation of the Tag Cache.

5.3.1.1 Memory Requests

The Tag Cache performs a fast lookup of the tag corresponding to the requested address and forwards the message to the memory controllers. In case of a Tag Cache hit, the forwarded message contains additional information about the requested memory request. The additional information contains the way identifier information pertaining to the requested address in the DRAM Cache. At the DRAM Cache, the message is parsed, and the data is retrieved. In case of a Tag Cache miss, the memory access request to the DRAM Cache is sent without any additional information (null). At the DRAM Cache, this request is treated as a normal memory operation where a tag lookup operation is carried out before data can be located.

5.3.1.2 Tag Updates

Updating the tag information in the Tag Cache is done as follows. In case of a Tag Cache hit, the Tag Cache updates the MRU bits of the tags accordingly. (Note that the tags nor the way identifiers need to be updated, only the MRU bits need to be updated.) In case of a Tag Cache miss, we leverage the memory response message received from the memory controller. The memory controller sends the request to the DRAM Cache and keeps record of this memory request. The DRAM Cache, upon returning the data, includes the way identifier of the cache block in the DRAM Cache. The cache block and the (5-bit) way identifier are then forwarded to the memory controller, which in turn forwards it to the Tag Cache (along with the corresponding tag). The Tag Cache now needs to replace the least recently used tag with the new tag — the least recently used tag is easily identified using the MRU bits. This mechanism ensures that the Tag Cache keeps track of the, in our case 8, most recently used cache blocks in the DRAM Cache.

5.3.2 DRAM Cache Access Latency with Tag Cache

A typical DRAM access requires activating a row of cells, sensing the charge and finally transmitting the sensed data over the bus. Since row activation has drained the corresponding capacitors, and to restore the charge back, a precharge operation is required to disconnect the cells from the bitlines and prepare the sense amplifiers and bitlines for subsequent activation.

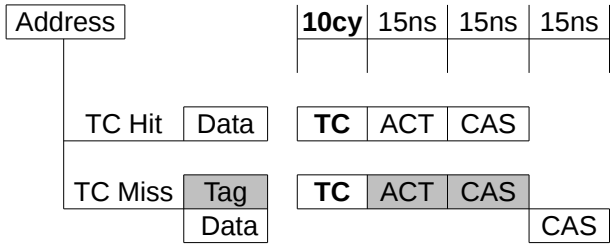


Figure 5.3: Closed-page latency with Tag Cache (TC) hit/miss. (DRAM Cache tag accesses are shown in gray.)

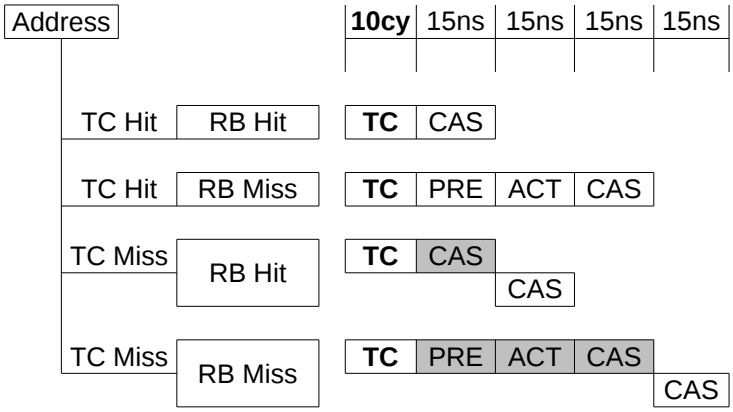


Figure 5.4: Open-page latency with Tag Cache (TC) hit/miss and row buffer (RB) hit/miss. (DRAM Cache tag accesses are shown in gray.)

5.3.2.1 Closed-page Policy

A closed-page policy ensures that a DRAM Cache row-buffer (RB) is closed after every read or write. This may not allow for maximum memory bandwidth, but makes the access latency predictable. The closed-page policy can be effective in situations where many different DRAM Cache rows are accessed frequently. Figure 5.3 highlights the DRAM Cache access latency with Tag Cache overhead. Although the Tag Cache access latency overhead of 10 cycles is added to each DRAM Cache access — the 10 cycles equals the access latency to L2 in our setup — the overall effect is minor. The main benefit can be observed in case of a Tag Cache hit where the DRAM Cache access latency can be reduced from 45 ns to 30 ns.

5.3.2.2 Open-page Policy

In an open-page policy, the DRAM Cache row-buffer is left open for a fixed amount of time after a read/write. This allows for both row-buffer hits and misses. The row-buffer hit signifies that the DRAM Cache row is already present in the bank row-buffer, hence neither precharge nor row activation operations are needed. On the contrary, the row-buffer miss signifies that the requested DRAM Cache row is different from the DRAM Cache row that is currently present in the bank row-buffer. To get the requested row to the bank row-buffer, the row-buffer cells are disconnected and row bitline sense amplifiers are prepared (precharge) followed by activation. Allowing the row buffer to be left open permits the possibility of row-buffer hits due to spatial locality of data, thereby reducing the overall access latency. Figure 5.4 shows all the valid combinations of row-buffer status along with Tag Cache hit/miss. In case of a Tag Cache miss, the overall access latency is not affected. A Tag Cache hit in conjunction with a row-buffer

Core	
Core type	4-way issue OOO, 128-entry ROB
Load/store queue	48 load entries, 32 store entries
L1-I cache	32 KB, 4-way, 64 B line, 3 cycles
L1-D cache	32 KB, 4-way, 64 B line, 3 cycles
Tile	
Tile size	4 cores
Core count	64
Tile count	16
L2 cache	2 MB, 16-way, 64 B cache line 10 cycle access time
L2 prefetcher	stride-based 8 independent streams
Chip wide configuration	
Coherence protocol	directory-based MESI distributed tags
Network on Chip	mesh 4×4, 32 GB/s/link
DRAM Cache	0.5 GB, 1 GB, 2 GB PRE-ACT-CAS: 15-15-15 ns ACT-CAS: 15-15 ns LRU replacement policy 64 B cache line, 2 KB row buffer 512 GB/s total
Memory controllers	8
Main memory	80 ns latency, 64 GB/s total
Frequency	1.2 GHz
Vdd	0.8 V
Technology	22 nm

Table 5.1: Simulated architecture configuration.

hit reduces the DRAM Cache access latency to 15 ns.

5.4 Experimental Setup

5.4.1 Simulation Framework

We use the Sniper multi-core simulator [37], version 6.0 for evaluating the Tag Cache mechanism on a tiled 64-core architecture. Each tile consists of 4 cores with a shared L2, see Table 5.1. We consider two Tags-in-DRAM Cache organizations, Loh-Hill Cache and Alloy Cache, see Table 5.2. We use McPAT v1.0 [111] to estimate power and energy consumption for a 22 nm technology node. We extend our simulation infrastructure to model a DRAM Cache in detail, assuming that the DRAM Cache is a memory-side cache and hence is not considered within the coherence space/traffic. Similar to previous works, we assume that die-stacked DRAM array has lower latency than off-chip main memory using conventional DIMM technology. The DRAM Cache is implemented in a interleaved manner in accordance to the memory controllers (8 memory controllers in our setup) to enable concurrent accesses across channels. The energy/power of the DRAM Cache is estimated using DDR3 technology [126] and scaled as per HMC v1 [85].

Component	Parameters
Alloy Cache (AC)	direct-mapped 30 ns hit latency (close-page) (80 B, one bulk access) Extra cycles for bulk transfer
Loh-Hill Cache (LHC)	29-way associative 30 ns + 15 ns hit latency (close-page) tag + data, compound access

Table 5.2: Tags-in-DRAM Cache Configurations.

(a) Multi-program workloads (SPEC CPU2006)

Workload	**Description**	**Benchmarks**
WL0	SPEC Average	all 55 + 9 uniform random
WL1	Compute	8 compute bound, $\times$8
WL2	Mixed	8 compute + 8 memory, $\times$4
WL3	Memory	8 memory bound, $\times$8

(b) Multi-program multi-threaded workloads

Workload	**Benchmarks**	**Input set**	**#Threads**
NAS Parallel Benchmark suite			
NPB1	BT, CG, FT, IS	Class B	16 each
NPB2	LU, MG, SP, UA		

Table 5.3: Workloads.

Component	Parameters
Tag Cache (TC)	Distributed, co-exist on L2 1-way per L2 (128 KB) 4/8-way; 16 K entries LRU replacement policy

Table 5.4: Tag Cache Configuration.

5.4.2 Workloads

5.4.2.1 Multi-program Workloads

We run a number of multi-program workloads composed of SPEC CPU2006 benchmarks; there are 29 programs in total, which along with all reference inputs leads to 55 benchmarks in total. We select representative simulation points of 750 million instructions each using PinPoints [134]. Four multi-program workloads with 64 benchmarks each are constructed by combining these 55 benchmarks as indicated in Table 5.3 (a). Each benchmark is pinned to a core unless mentioned otherwise. We run the simulation for 1000 ms to keep total simulation time under feasible limits. When a benchmark completes before this time, it is restarted on the same core. We quantify weighted speedup [158] or system throughput (STP) [54] which quantifies the aggregate throughput achieved by all cores in the system.

5.4.2.2 Multi-program Multi-threaded Workloads

We create workloads by combining multiple multi-threaded applications from the NAS Parallel Benchmarks (NPB) [14] suite, see Table 5.3 (b). We use the *class B* input set for NPB. We construct two workloads, each running 64 threads in total: *NPB1* consists of four NPB applications (16 threads each) running concurrently; *NPB2* consists of a different set of NPB applications. Execution of all multi-threaded applications in a workload begins after the last application has reached the region of interest (ROI). We run each workload for 1000 ms to keep total simulation time under feasible limits.

5.5 Evaluation and Results

We now evaluate the efficacy of the Tag Cache. This is done in a number of steps. We first evaluate Tags-in-DRAM Cache designs in the context of our tiled many-core processor. We then show the effectiveness of the Tag Cache in the context of the Loh-Hill Cache. Next, we show sensitivity analyses of the Tag Cache with respect to associativity and DRAM Cache size. Finally, we evaluate the DRAM Cache access policies, namely closed-page versus open-page policies, for the direct-mapped Alloy Cache (with ideal MAP) and the multi-way Loh-Hill Cache (*LHC*). And we evaluate the impact on energy consumption.

Unless mentioned otherwise, we consider a Tag Cache configuration of associativity of 8, i.e., it caches tags for the 8 MRU blocks in the DRAM Cache, and the Tag Cache occupies one way of the shared L2, see also Table 5.4. The Tag Cache contribution to the overall DRAM Cache hit rate is marked by contrasting shades in the figures. Further, we consider a closed-page DRAM Cache with 2 GB capacity, unless mentioned otherwise.

5.5.1 Tags-in-DRAM Cache Performance

We first evaluate different Tags-in-DRAM Cache designs on a many-core processor.

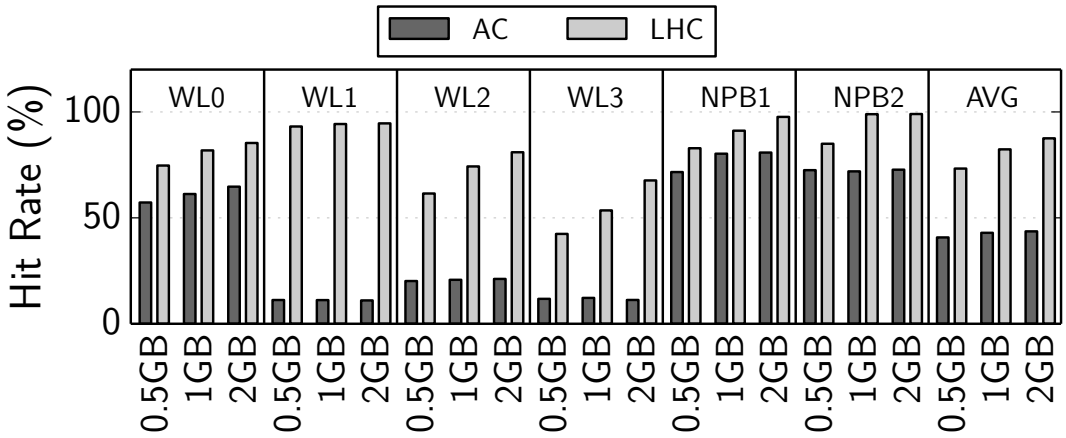


Figure 5.5: Hit rate (%) for the Alloy Cache and Loh-Hill Cache.

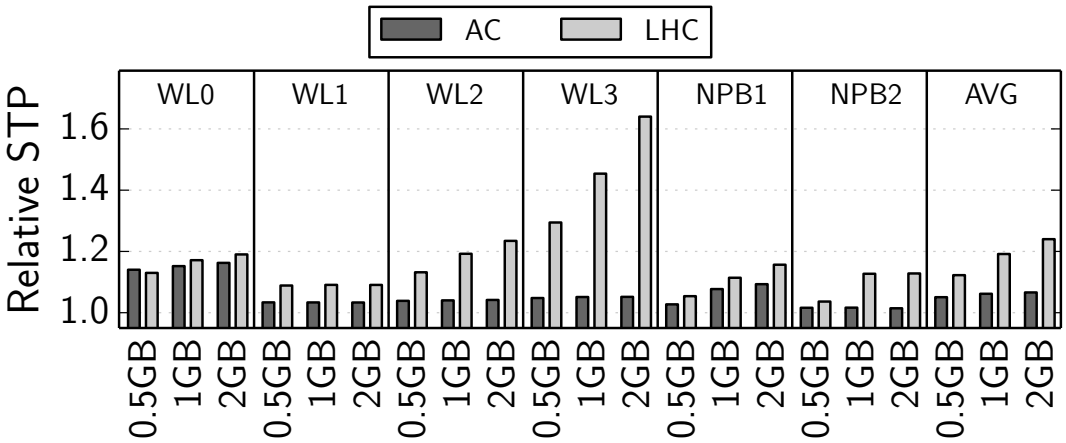


Figure 5.6: STP (relative to no DRAM Cache) for the Alloy Cache and Loh-Hill Cache.

5.5.1.1 Direct-Mapped versus Multi-Way Associative

We evaluate the Alloy Cache (with ideal MAP) versus the Loh-Hill Cache (without MissMAP) for different DRAM Cache sizes. Figure 5.5 reports hit rate. On average, the multi-way associative Loh-Hill Cache shows about 2× higher hit rate than the direct-mapped Alloy Cache. Even though the direct-mapped Alloy Cache provides a lower hit rate due to a larger number of conflict misses, it provides much lower access latency. Lower access latency does amortize the performance loss to some extent due to a significantly higher miss rate. We can observe this in particular for average SPEC CPU2006 workload, WL0, see Figure 5.6. Overall, we observe that in the context of a many-core architecture, the Loh-Hill Cache shows a 17% higher relative system throughput compared to the direct-mapped Alloy Cache. This trend increases with DRAM Cache size. The key take-away here is that in a many-core architecture context, multi-way associativity in the DRAM Cache is more important than short DRAM Cache hit latency. This conclusion contradicts prior results by Qureshi and Loh [144] which considered a multi-core setups with a limited number of cores, which leads to far fewer conflict misses in a direct-mapped DRAM Cache.

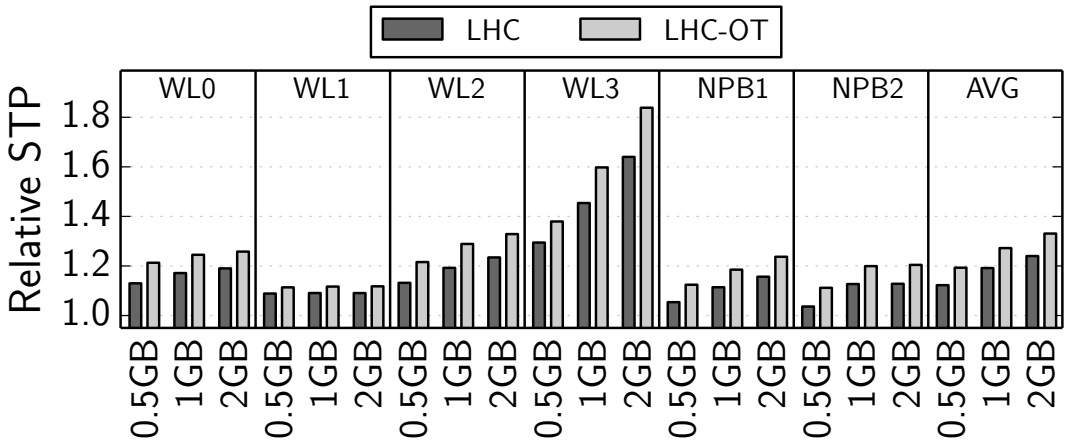


Figure 5.7: Performance comparison with tag lookup and Oracle Tags for Loh-Hill Cache.

5.5.1.2 Tags-in-DRAM Cache with Oracle Tags

To understand the potential performance improvement from tag caching, we now consider an oracle tag mechanism. Figure 5.7 plots the relative STP (with respect no DRAM Cache) for the unassisted Loh-Hill Cache with tag lookup (*LHC*) versus the Loh-Hill Cache with zero-latency Oracle Tags (*LHC-OT*) for all multi-programmed workloads on a 64-core setup. *LHC-OT* provides the access latency of the direct-mapped Alloy Cache with the hit rate of the 29-way Loh-Hill Cache. *LHC-OT* outperforms *LHC* by 9% on average on a 2 GB DRAM Cache. Compared to the Alloy Cache, a 2 GB 29-way set-associative DRAM Cache with Oracle Tags provides on average 26.5% higher system throughput compared to the Alloy Cache of the same size with an ideal MAP.

5.5.2 Effectiveness of Tag Cache

To show the effectiveness of the Tag Cache mechanism, we compare the performance of the Tag Cache enabled Loh-Hill Cache (*LHC-TC*) versus the Loh-Hill Cache with MissMap (*LHC-MM*) and the Loh-Hill Cache with ATCache (*LHC-ATC*). As a point of reference, we also show results for the unassisted Loh-Hill Cache (*LHC*) and the Loh-Hill Cache with Oracle Tags (*LHC-OT*). To represent the entire content on a 2 GB of DRAM Cache, a total of 8 MB is used for the MissMap in *LHC-MM*. The MissMap table is distributed across all the L2 caches per tile, thereby decreasing the effective capacity of the available L2 by 4 ways or 512 KB (only 12 ways out of 16 ways are available for data blocks). Alike MissMap, the Tag Cache is stored in L2, but it only occupies one way per L2 (15 ways out of 16 ways are available for data blocks). The ATCache structure (caching ratio of 256) is also distributed across all the L2 caches per tile. Alike Tag Cache, the ATCache structure reduces the effective capacity of the available L2 by one way (15 ways are available for data blocks).

5.5.2.1 Effect on Shared LLC

Before we delve into the overall performance numbers of the Tag Cache enabled Loh-Hill DRAM Cache, let us first look into the impact of using the Tag Cache, ATCache and MissMap table on the shared L2. Figure 5.8 plots the average shared L2 cache miss rate normalized to *LHC*. As mentioned above, the MissMap table occupies 4 ways whereas the ATCache and Tag Cache occupies only 1 way in L2. The reduction in L2 capacity affects the miss rate, in particular for applications with cache friendly and cache fitting behavior. This trend is observed for workloads WL1, WL2, NPB1 and NPB2. The average L2 miss

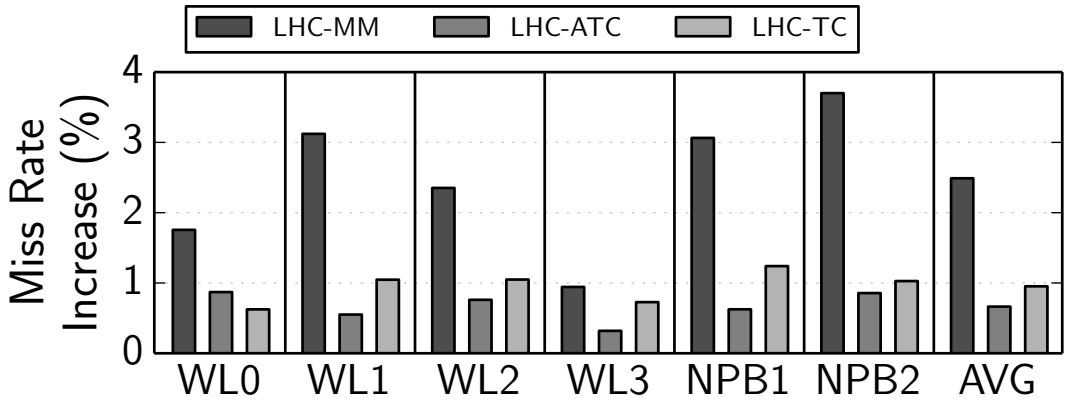


Figure 5.8: Shared L2 miss rate (*LHC* normalized).

rate for these workloads increases by 2.5% on average for *LHC-MM* compared to *LHC*, and up to 3.8%. In contrast, Tag Cache and ATCache do not increase the L2 cache miss rate as much; we measure an average increase by 0.95% and 0.65% only, and by at most 1.15% and 0.95%, respectively.

5.5.2.2 LLC Miss Latency Breakdown

Figure 5.9 shows the average LLC miss latency breakdown for average SPEC CPU2006 workload WL0 (top) and average for all workloads (bottom). The latency breakdown plot includes the latency values due to remote cache operations along with accesses to the tag directory, NoC, DRAM Cache and off-chip DRAM. The ideal *LHC-OT* incurs zero-latency tag lookup for each DRAM Cache request (DramCacheTag=0). The Loh-Hill Cache variations, *LHC*, *LHC-MM*, *LHC-ATC* and *LHC-TC*, suffer from tag lookup operations. The latency contribution due to tag lookup operation in DRAM Cache reduces in *LHC-TC* compared to *LHC* and *LHC-MM* by 11.8 ns on average, see DramCacheTag in Figure 5.9. This reduction in DRAM Cache Tag lookup operation is attributed due to DRAM Cache tag lookup bypassing thanks to the Tag Cache (on average 45.4% of the DRAM Cache hits are contributed by the Tag Cache). For ATCache, latency is reduced by 3.4 ns on average compared to *LHC*, which is significantly lower than *LHC-TC* due to a much lower DRAM Cache hit contribution (8% on average).

5.5.2.3 Overall Memory Miss Latency

To understand the effect of reduced shared L2 capacity along with Tag Cache assisted tag lookup bypassing, we analyze the average memory miss latency for all Loh-Hill DRAM Cache variants. In Figure 5.10, we plot the average miss latency observed at the L1 D-cache. The D-L1 miss latency includes the latency incorporated due to all un-core components such as the shared L2, TD, NoC, DRAM Cache and off-chip DRAM. Across all workloads, the average D-L1 miss latency for *LHC* equals 57.3 ns. In an ideal DRAM Cache configuration, Loh-Hill Cache with Oracle Tags *LHC-OT*, the average miss latency reduces to 51.8 ns, which is about 5.5 ns smaller than *LHC* which is in accordance with the LLC miss latency breakdown, as shown in the previous section. Unlike *LHC* and *LHC-OT*, the other Loh-Hill Cache variants use a portion of the L2 cache to reduce the overall DRAM Cache access latency. This increases the miss rate of the L2 (as seen in Section 5.5.2.1) thereby adding more latency to each memory miss request from the core. Using the Tag Cache (*LHC-TC*), we observe an average miss latency of 54.1 ns which is 3.2 ns lower than *LHC*. ATCache reduces the average miss latency of *LHC* by 0.6 ns only. On the other hand, *LHC-MM* incurs the average latency of 60.45 ns, which is 3.11 ns higher than *LHC*, due to a significantly higher miss rate in the L2 caches.

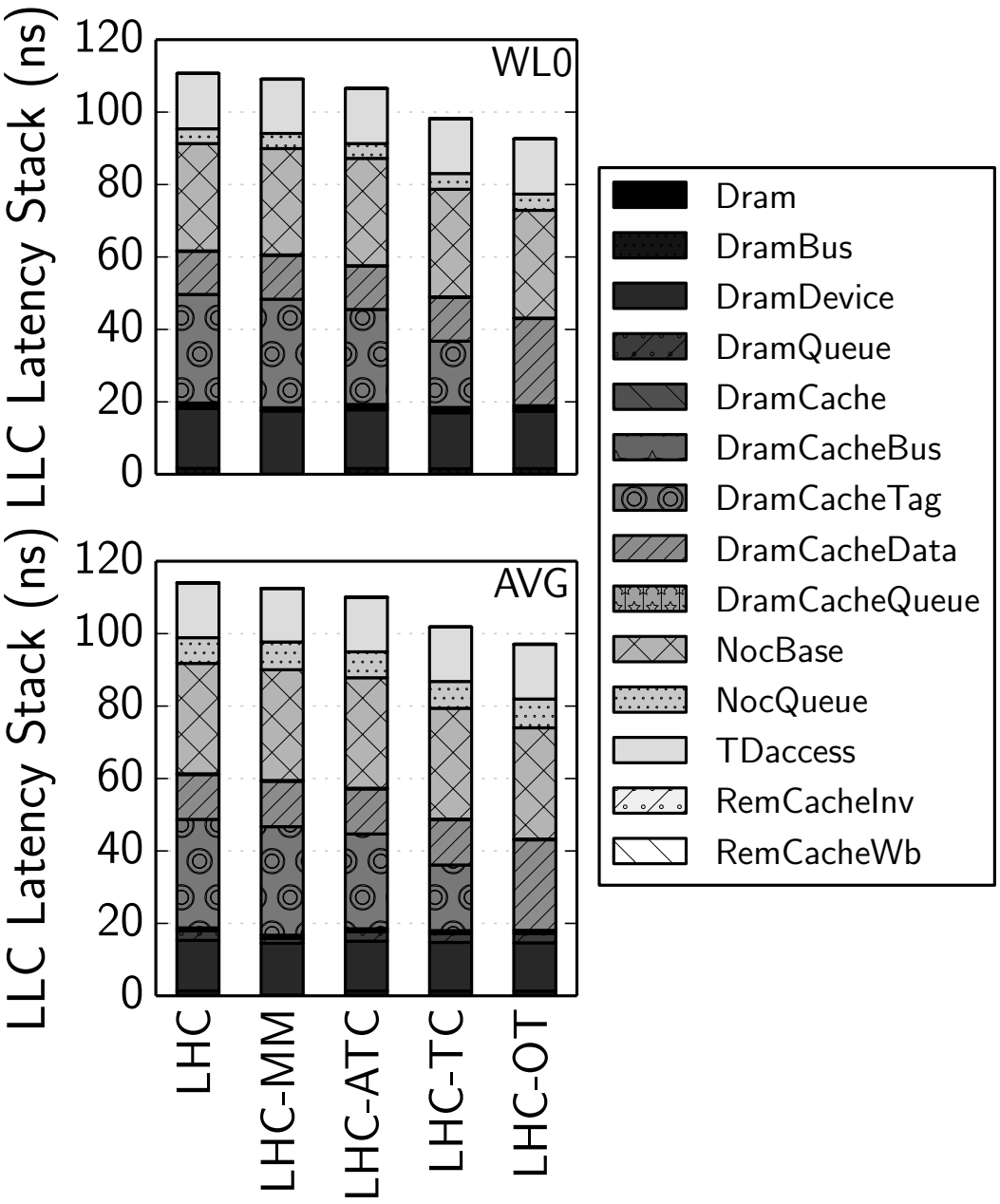


Figure 5.9: Average LLC miss latency breakdown.

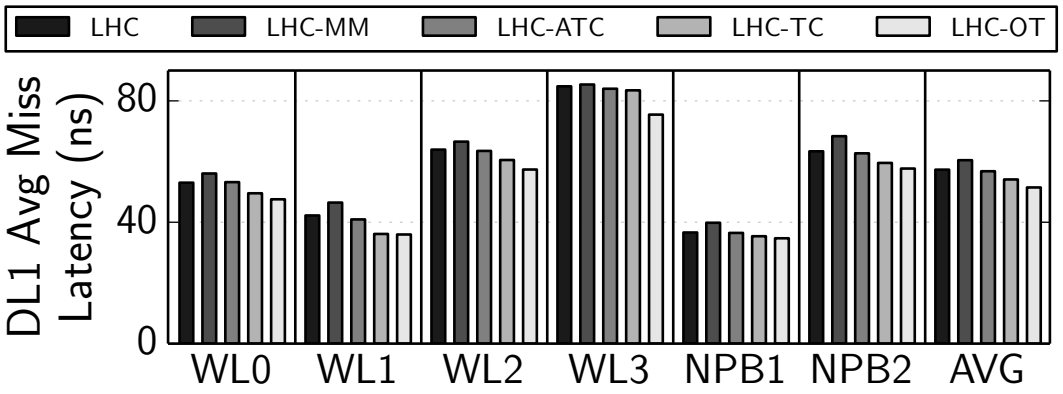


Figure 5.10: Average DL1 miss latency for the 2 GB Loh-Hill Cache variants.

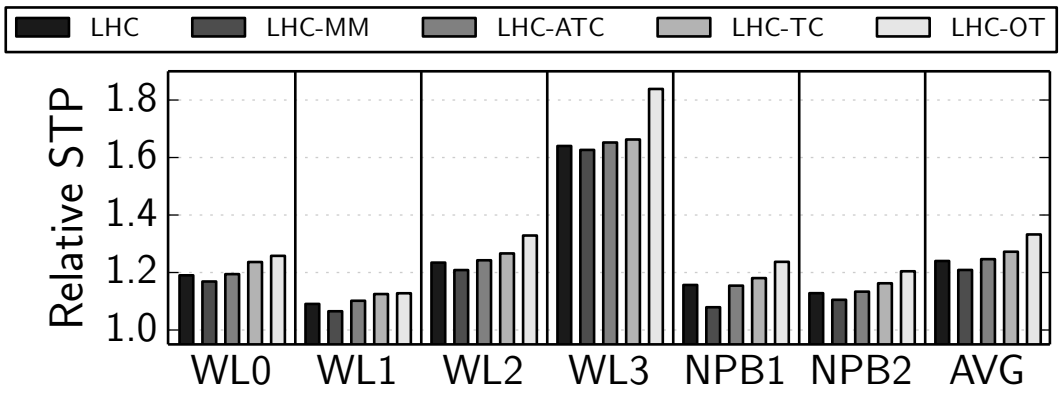


Figure 5.11: Relative STP for the 2 GB Loh-Hill Cache variants.

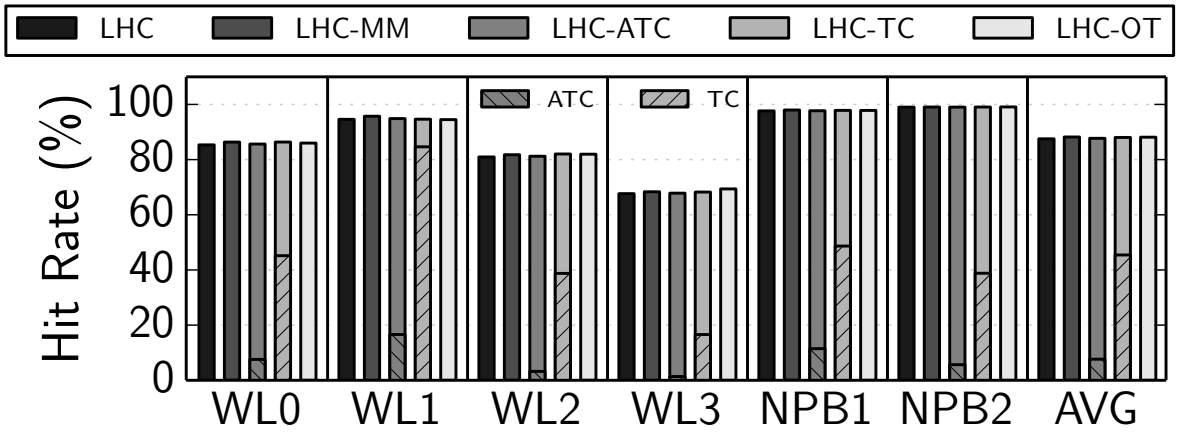


Figure 5.12: Hit rate for the 2 GB Loh-Hill Cache variants.

5.5.2.4 Overall Performance

Figure 5.11 shows relative STP (with respect to no DRAM Cache) for 5 variations of the Loh-Hill DRAM Cache. We observe that *LHC-MM* experiences an average performance loss of 3.1% compared to *LHC*. This is due to an increase in the number of L2 misses as the MissMap occupies one fourth of the L2. The increase in the number of L2 misses causes an increase in overall miss latency, as described in the previous section. *LHC-ATC* yields 0.6% higher performance (on average) compared to *LHC*. On the contrary, *LHC-TC* yields 3.3% higher performance (on average) compared to *LHC*, and 6.4% higher performance compared to *LHC-MM*. The increase in performance due to the Tag Cache can be attributed to a reduction in the tag lookup latency in the Loh-Hill Cache, which in its turn leads to a reduction in the overall memory access latency, as reported in the previous sections.

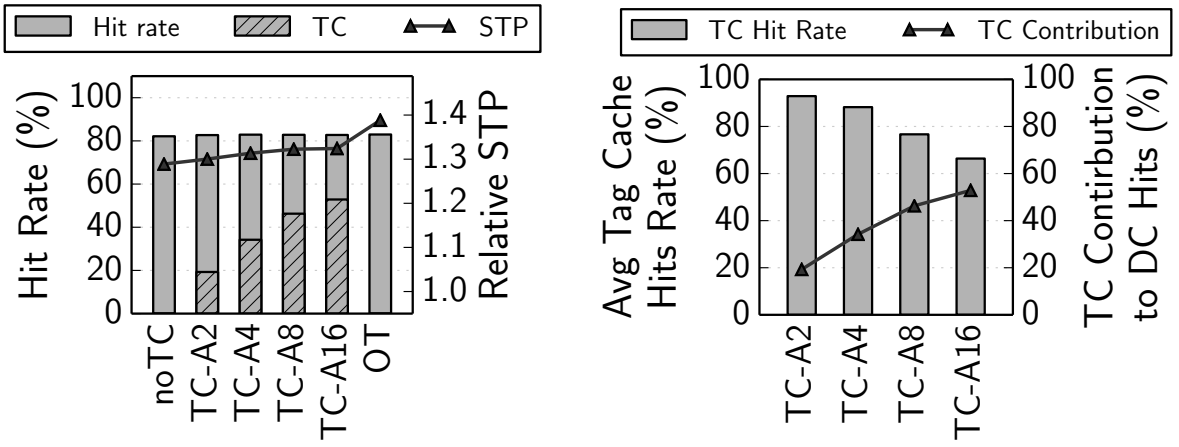
Overall, *LHC-TC* is within 6% compared to *LHC-OT*. The largest gap is observed for workload WL3, which is due to the large working set of the memory-intensive applications in WL3 which results in a relatively modest benefit from the Tag Cache mechanism.

The performance gain using Tag Cache can thus be attributed to lower DRAM Cache latency as a result of Tag Cache hits. Figure 5.12 reports DRAM Cache hit rate and, for the Tag Cache, it also reports the hit rate in the Tag Cache (shaded gray). The latter is the fraction of DRAM Cache accesses that bypass the tag lookup in the DRAM Cache, thereby shortening the overall DRAM Cache access latency. Overall, the Tag Cache contributes to on average 45.4% of the overall DRAM Cache hits. The lowest hit rate (16.6%) is observed for WL3 as this workload is composed of a number of memory-trashing applications.

5.5.3 Tag Cache Sensitivity

5.5.3.1 Sensitivity to Associativity

We now conduct a sensitivity study to understand the effect of tag caching for 2, 4, 8 and 16 MRU DRAM Cache ways while using 1-way of the L2 for the Tag Cache (containing 16 K entries), see Figure 5.13. The associativity configurations are highlighted as *TC-Ax*, where *Ax* denotes the above mentioned associativity values of the Tag Cache. Using this nomenclature, Figure 5.13(a) plots the total DRAM Cache hit rate (%) along with the Tag Cache hits (as a percentage of the total DRAM Cache hit rate) on the left y-axis and relative STP on right y-axis, averaged across all workloads. The values for unassisted *LHC* and *LHC-OT* are also included to show the minimum and maximum performance range. The Tag Cache hit contribution



(a) Hit Rate vs. STP

(b) Tag Cache efficiency

Figure 5.13: Tag Cache sensitivity wrt. associativity for a 2 GB DRAM Cache.

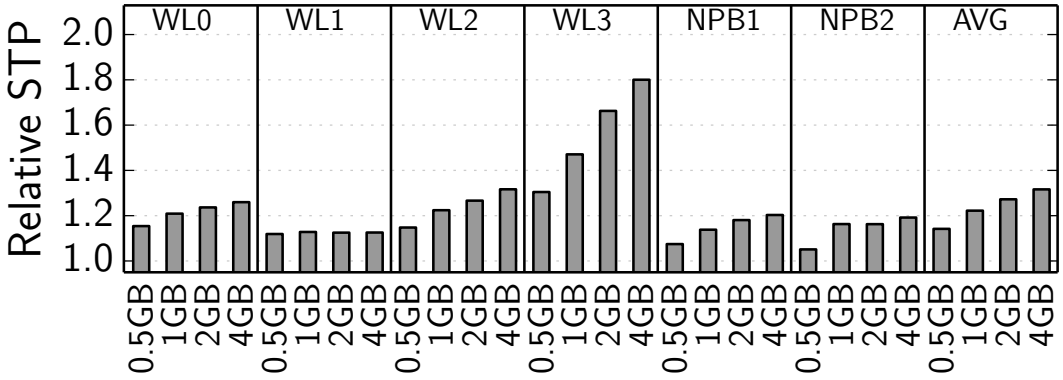


Figure 5.14: Relative STP versus DRAM Cache size.

to the overall DRAM Cache hit rate increases from 19% to 52% for 2-way versus 16-way associativity. We observe a significant rise in hit rate between 2-way and 4-way, namely by 2×. However, we observe a saturating trend, and the overall rise in Tag Cache hit rate with increasing associativity does not provide corresponding benefits in reducing average miss latency and gain in overall performance. The relative STP shows a saturating effect with increasing Tag Cache associativity. Figure 5.13(b) corroborates this finding as it shows the Tag Cache associativity trade-off as a function of efficiency (hits in Tag Cache) and Tag Cache contribution to the DRAM Cache hit rate (*LHC-TC*). Lowering the associativity of Tag Cache increases its efficiency at the cost of lower DRAM Cache hits, and vice versa. We consider the associativity of 8 as it provides a reasonable trade-off overall.

5.5.3.2 Sensitivity to DRAM Cache Size

We also conduct a sensitivity study by varying the size of the DRAM Cache while keeping the Tag Cache structure constant. Figure 5.15 plots the DRAM Cache hit rate for all multi-programmed workloads; the shaded part accounts for the Tag Cache's contribution. We observe that overall DRAM Cache hit rate (%) increases from 74.1% to 91.6% on 0.5 GB to 4 GB DRAM Cache. The increase in hit rate is attributed to fewer capacity misses. The same effect is observed with the Tag Cache contribution on the total DRAM

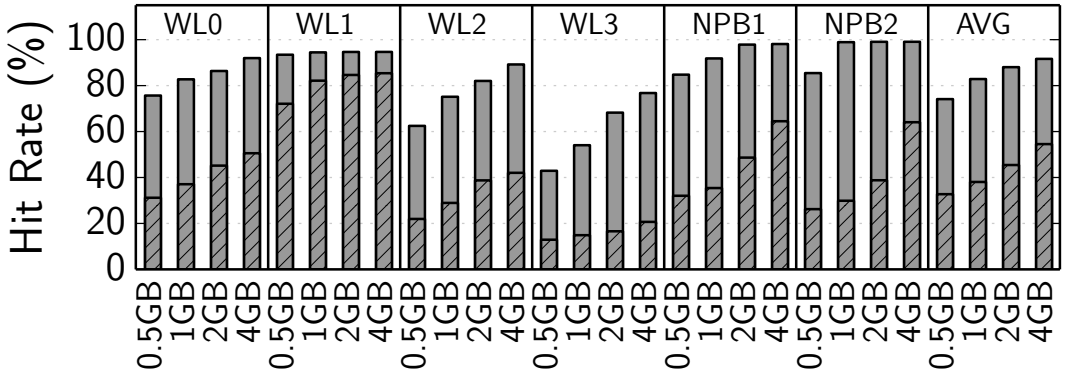


Figure 5.15: DRAM Cache (w/ Tag Caching) hit rate sensitivity to size.

Cache hit rate. On average, the Tag Cache contribution to DRAM Cache hit rate increases from 32.7% to 54.5% on 0.5 GB to 4 GB DRAM Cache, respectively. This increase in Tag Cache contribution to DRAM Cache hit rate translates into a performance improvement as shown in Figure 5.14. Although the Tag Cache size is constant, the Tag Cache contribution to DRAM cache hits shows an increasing trend with DRAM Cache size; the DRAM Cache is experiencing more capacity hits which the Tag Cache is more likely to keep track of by maintaining the MRU tags. Workload WL3, which is composed of a set of memory-intensive applications, is affected most by DRAM Cache size. STP improves by 31% and up to 80% for a 0.5 GB and 4 GB DRAM Cache, respectively, compared to no DRAM Cache; DRAM Cache hit rate increases from 42.9% to 76.8% for a 0.5 GB and 4 GB DRAM Cache, respectively. The Tag Cache still contributes about 12.9% to 20.7% of the overall DRAM Cache hit rate.

5.5.4 Closed versus Open Policy

We now evaluate Tag Cache in the context of an open-page policy. The DRAM Cache is assumed to have a row buffer of size 2 KB with 8 channels and 8 banks per channel. We compare closed-page and open-page policies for the Alloy Cache with ideal MAP (*AC*), unassisted Loh-Hill Cache (*LHC*), Loh-Hill Cache with MissMap (*LHC-MM*), Loh-Hill Cache with ATCache (*LHC-ATC*), and the Loh-Hill Cache with Tag Cache (*LHC-TC*). Figure 5.16 reports relative STP (with respect to no DRAM Cache). In spite of having the shortest hit latency, Alloy Cache's performance is worse than the Loh-Hill Cache for both the closed and open-page policies (see Figure 5.17). This is due to its much lower DRAM Cache hit rate. On average, *AC* provides 6.6% and 3% higher system throughput than an architecture without DRAM Cache for the closed and open-page policies, respectively. On the other hand, the 29-way *LHC* provides on average 17.4% and 14.8% higher performance, respectively. Using the distributed Tag Cache mechanism, the performance of the Loh-Hill Cache (*LHC-TC*) improves by 6.4% and 5.8% on average over *LHC-MM*, respectively, and by 20.6% and 17.7% on average over Alloy Cache, respectively. On the contrary, *LHC-ATC* provides 2.6% and 2.2% lower performance on average compared to *LHC-TC*, respectively.

It is interesting to note that the Alloy Cache yields the highest performance increase for workload *WL0*, which is the average SPEC CPU workload. The reason is that this workload includes a wide range of applications with varying memory characteristics, which the Alloy Cache is well capable of exploiting. Yet, still, high associativity in the DRAM Cache, in spite of incurring a higher hit latency, is key to further improve performance, especially for workloads that are more memory-intensive.

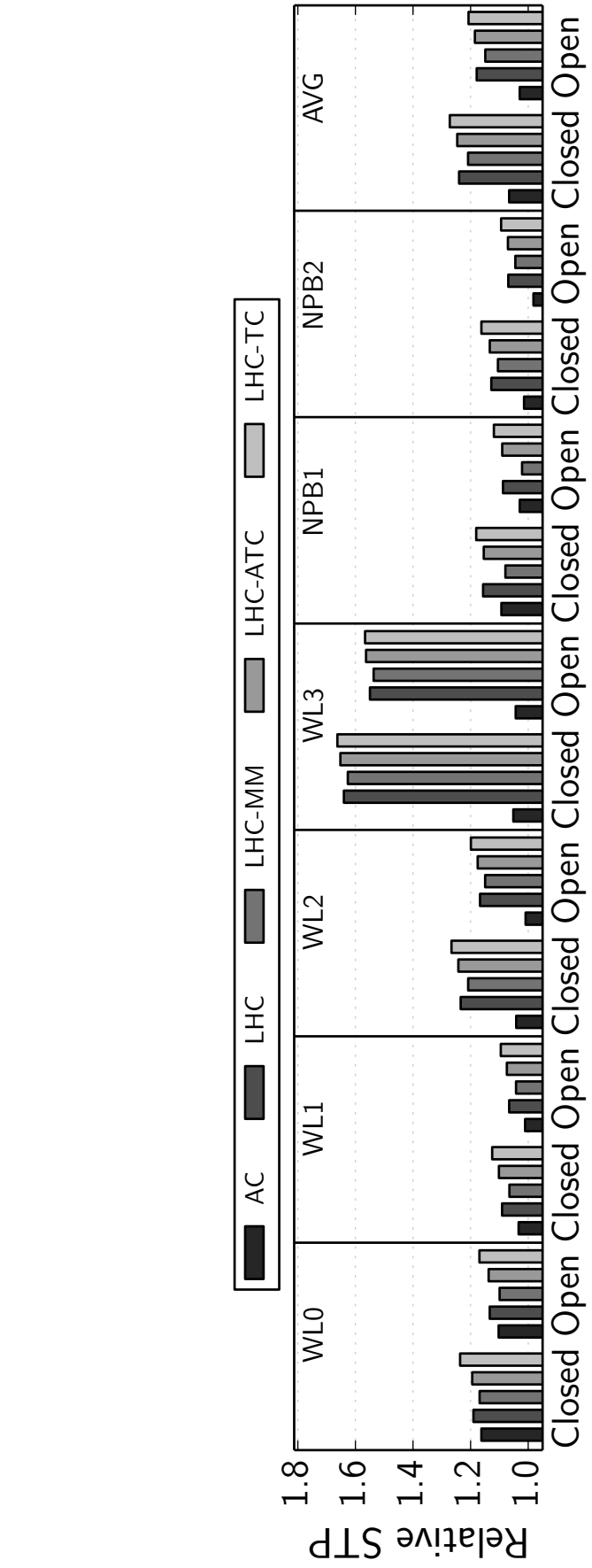


Figure 5.16: Relative STP for all multi-program workloads on 64-core setup with a 2 GB DRAM Cache considering closed and open-page policies.

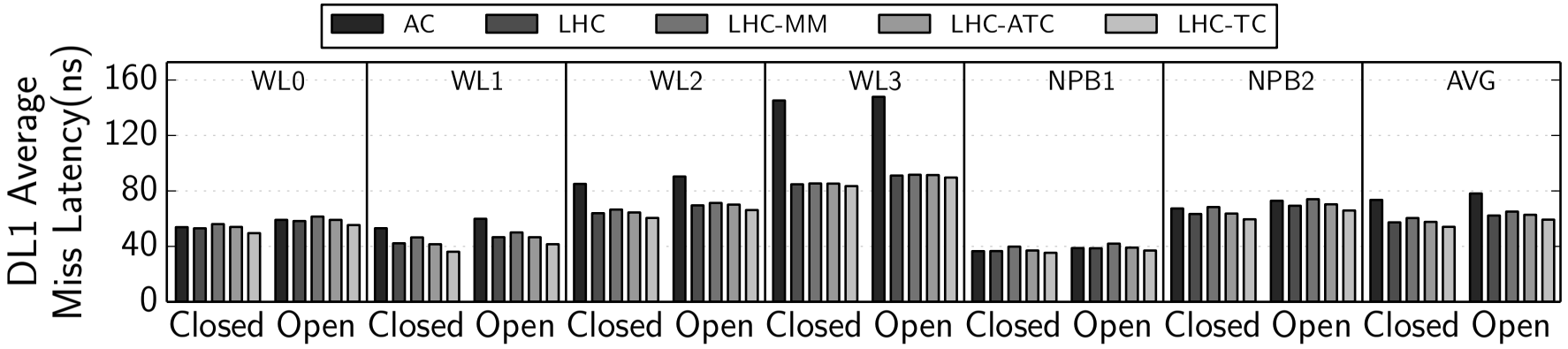


Figure 5.17: Average DL1 miss latency for all multi-program workloads on 64-core setup with a 2 GB DRAM Cache considering closed and open-page policies.

5.5.5 Energy Consumption

Tag Caching not only helps improving performance, it also helps reduce energy consumption. As mentioned in Section 5.3.2, the CAS operation to fetch the tag lines in the Loh-Hill Cache to read the tag is eliminated upon a Tag Cache hit. This not only reduces DRAM Cache access latency, it also reduces energy consumption. Based on the numbers provided in [85, 126], we calculate the static and dynamic energy (excluding refresh energy) of a 2 GB DRAM Cache for *LHC*, *LHC-MM*, *LHC-ATC* and *LHC-TC*. In Figure 5.18, we show the reduction in energy consumption (both static and dynamic) normalized to *LHC* for *LHC-MM*, *LHC-ATC* and *LHC-TC* for the closed and open-page policies. On average, *LHC-TC* reduces DRAM Cache energy consumption by 20.6%, 10.4% and 7.4% over *LHC-MM*, *LHC* and *LHC-ATC*, respectively. The reason is twofold: (i) *LHC-TC* achieves better performance and thus shorter execution time, which in turn reduces energy consumption, and (ii) no need to read the tags in the DRAM Cache in case of a Tag Cache hit. In contrast, the MissMap-enabled Loh-Hill Cache, *LHC-MM*, experiences a slowdown in overall performance which causes an increase in energy consumption by 8.8% on average over *LHC*.

The highest reduction in energy consumption is observed for WL1, which is the compute-intensive SPEC CPU2006 workload. Energy is reduced by 19.6% compared to *LHC* and by 38.5% compared to *LHC-MM*, because of the high hit rate in the Tag Cache, as previously reported in Figure 5.12. The smallest reduction in energy is observed for WL3 (4.7% compared to *LHC*, and 7.2% compared to *LHC-MM*), which is the memory-intensive SPEC CPU2006 workload. This is due to the fact that the Tag Cache is too small to capture the large working set for this workload.

While energy consumption is reduced in the DRAM Cache, energy consumption increases in L2 because of the Tag Cache lookups. According to our measurements this amounts to an average increase in L2 energy by 7.3% on average. It is important to note that the total energy consumption in the DRAM Cache is 5 to 6 times higher than in L2, hence it is safe to conclude that Tag Cache leads to net energy savings.

5.6 Related Work

Prior works in DRAM Caches with conventional cache block sizes have mostly focused on managing tag storage overhead, cache hit rate, and/or cache access latency. For small-sized DRAM Caches, Zhao et al. [181] proposed on-die partial tags as a lookup for full tags in the DRAM Cache. With significantly larger die-stacked DRAM Cache size, the size of partial tags grows proportionally large. The Loh-Hill Cache [116], as mentioned before, proposed to use a MissMap table to bypass the requests to off-chip DRAM in case of DRAM Cache misses. The size of the MissMap grows proportionally with DRAM Cache size. For example, for a 2 GB of die-stacked DRAM Cache, the MissMap occupies 8 MB of LLC space, nearly half the size of an L3 typically supported in modern processors. Our distributed Tag Cache on Loh-Hill Cache occupies less then 2 MB and is not affected by DRAM Cache size. An extension of the Loh-Hill Cache is the Mostly-Clean DRAM Cache [156], which uses a miss predictor to reduce the MissMap storage overhead, and which uses parallel accesses to avoid the tag access serialization penalty. Tag Tables [60] used an on-demand page-table structure to store cache tag information for the DRAM Cache. The number of entries in Tag Table grows proportionally with DRAM Cache size at the cost of reduced LLC capacity. ATCache [75] proposes a single entity SRAM array to store set information of a multi-way associative DRAM Cache based on caching ratio. While keeping the caching ratio the same and increasing the DRAM Cache capacity, the size of ATCache grows proportionally.

Prior approaches toward reducing tag data storage overhead have managed the DRAM Cache using large cache lines on the order of kilobytes [44, 90]. Seznec [153] proposed to eliminate the need to store full tag data by using a smaller sector of a large cache block as a bit vector. These techniques still store tag data for all of DRAM, increasing bandwidth consumption and pollution of the DRAM Cache, increasing false-sharing probability, and limiting scalability. Meza et al. [125] proposed tags in DRAM and caching

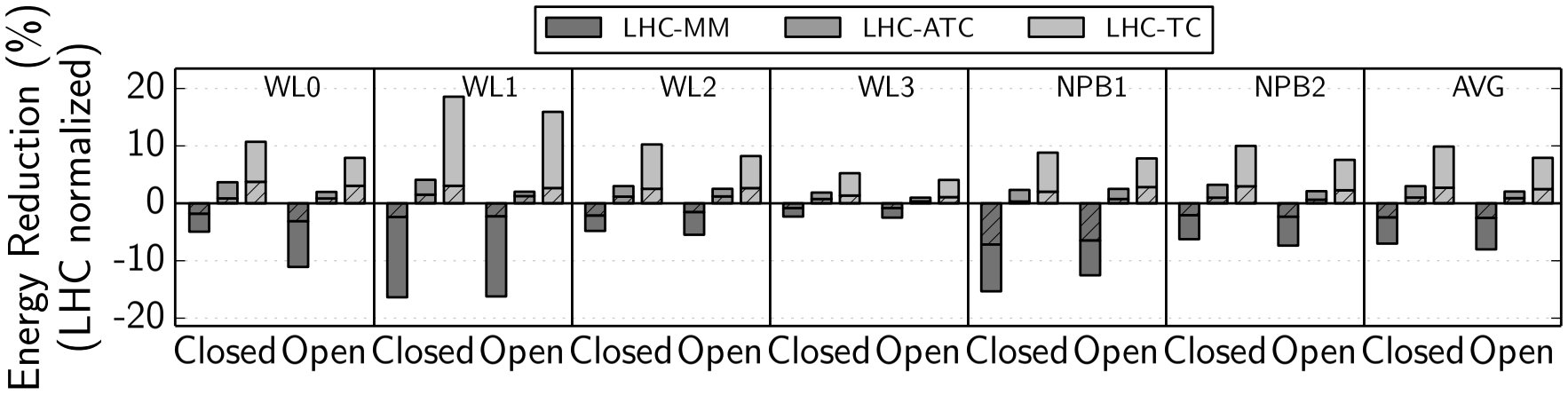


Figure 5.18: Energy reduction (*LHC* normalized) for a 64-core setup with 2 GB DRAM Cache (shaded region denote static energy); higher is better.

full tag information in a small direct-mapped buffer to exploit row buffer spatial locality. Footprint cache is also another DRAM Cache design, which is a sector cache design with prefetcher [88]. It uses SRAM storage to store the tag arrays; however, this is not a scalable solution, since the tag storage of 1 GB DRAM Cache can be as large as 6 MB. Also, enabling prefetch requests might exacerbate the bandwidth bloat problem in DRAM Cache due to the extra bandwidth consumed by inaccurate prefetches. Unison Cache's [87] embedded tags (like Alloy Cache) in Footprint Cache-like large allocation units are used to exploit spatial locality. Gulur et al. [67] proposed to embed tags for co-located conventional block and large block on separate tag bank of DRAM Cache to increase row-buffer hits at the cost of multiple precharge and activation operations for each read/write operations (leading to higher dynamic energy). It also proposes a centralized way locater on DRAM Cache to reduce hit latency. Unfortunately, the latency, energy and bandwidth consumption increases when large blocks are evicted and replaced in case of misses.

The existing prior works on gigascale DRAM Caches have focused on relatively small multi-core setups with 4 to 8 cores. In many-core processors, a DRAM Cache solution needs to be scalable towards larger core counts, which calls for a distributed solution. Our distributed Tag Cache mitigates these issues on a many-core processor while reducing the DRAM Cache hit latency and incurring limited space overhead. Moreover, the area overhead of distributed Tag Cache mechanism is independent of the DRAM Cache size unlike previously proposed mechanisms.

5.7 Summary

Large die-stacked DRAM Caches are clearly needed for many-core processors to bridge the memory wall problem. This poses a problem of how to efficiently manage the tags associated with gigascale DRAM Caches. Prior works have proposed innovative Tags-in-DRAM designs to efficiently co-locate tags with data. Unfortunately, Tags-in-DRAM designs with multi-way associativity suffer from high latency. To mitigate the latency, prior works have proposed structures that either mitigate tag lookup latency in DRAM Cache or bypass the request to off-chip DRAM. We observe that in many-core processor with gigascale die-stacked DRAM Cache, these approaches would not only require more resources but could potentially become a source of contention, thereby limiting overall system performance.

We propose Tag Cache, an on-chip distributed mechanism with limited space and latency overhead to bypass the tag read operation in multi-way DRAM Caches, thereby reducing hit latency. Each Tag Cache, stored in L2, stores tag information of the most recently used DRAM Cache ways. We show that the Tag Cache is able to exploit temporal locality of the DRAM Cache, thereby contributing to on average 46% of the DRAM Cache hits while running multi-programmed workloads. Our experimental results show that we achieve 6.4% higher performance and 20.6% lower energy compared to the previously proposed Loh-Hill Cache with MissMap on a 2 GB DRAM Cache, while using 4$\times$ less LLC resources. Our approach provides on average 2.6% higher STP with 7.5% lower energy consumption than ATCache-based Loh-Hill Cache. Compared to the direct-mapped Alloy Cache, our approach provides on average 20.6% higher performance.

Chapter 6
Conclusions and Future Work

6.1 Overview

With high performance many-core processors gaining much traction, it is imperative to have a power management mechanism that is low cost, less complex, scalable and capable of analyzing the complex behaviors and interactions across applications to optimize performance at different power envelopes.

On the other hand, successive technological advances in micro-architecture and process technology have sustained tremendous performance scaling, while the memory performance has not scaled to the same level. The proposal of near-memory architecture aims at mitigating the memory wall issue. While near-memory DRAM-based architecture aims to reduce the latency of memory operations by reducing the critical distance, it still suffers from high latency.

The work of this thesis introduces, develops, and analyses a novel method to address the issue of power wall and memory wall of a state of the art many-core high performance processors. The goal of the thesis was to provide a low overhead, low computational complexity and scalable mechanism to address the above mentioned issues. Although there exists many solutions that provide micro-architectural mechanisms to address the power wall and memory wall issues, none of them has solved the problem of providing full operability at constrained power envelops.

With high performance many-core processors gaining much traction, it is imperative to have a power management mechanism that is low cost, less complex, scalable and capable of analysing the complex behaviors and interactions across applications to optimize performance at different power envelopes.

On the other hand, successive technological advances in micro-architecture and process technology have sustained tremendous performance scaling, while the memory performance has not scaled to the same level. The proposal of near-memory architecture aims at mitigating the memory wall issue. While near-memory DRAM-based architecture aims to reduce the latency of memory operations by reducing the critical distance, it still suffers from high latency.

6.2 Summary of Research

In the first part of the thesis, we have addressed the power-wall issue with respect to many-core processors. As transistor budgets outpace the power envelope, various techniques have been previously introduced that allow for run-time adaptation to improve power efficiency. Yet the combination of several of these techniques, including core adaptation, cache adaptation and per-core DVFS, will be required to meet ever more stringent power budgets. This quickly increases power management complexity which

must dynamically allocate power among cores and threads based on workload behavior and run-time conditions.

To address this problem we introduce Chrysso in Chapter 3, an integrated and scalable model-driven power management that quickly select the best combination adaptation method out of core and uncore micro-architecture adaptation, per-core DVFS, or any combination thereof. This decision is based on workload behavior for each core on a large many-core system, while optimizing the chip's total performance and defined power usage. Chrysso's global optimization algorithm can quickly search the adaptation space by making performance and power projections to identify Pareto-optimal configurations, effectively pruning the search space. This makes Chrysso scale favorably to large numbers of cores and configurations, significantly outperforming isolated adaptation techniques in isolation w.r.t. total system throughput under a broad range of power budgets. At stringent power budgets, Chrysso achieves 1.9× better chip performance over core-level gating for multi-programmed workloads, and 1.5× higher performance for multi-threaded workloads. The critical/bottleneck thread identification in Chrysso-Critical is able to reduce the execution time further by 0.2–10% on the average over Chrysso.

Most existing power management schemes use a centralized approach (like Chrysso) to regulate power dissipation based on power monitoring and performance characteristics. Unfortunately, the complexity and overhead of centralized power management increases significantly with core count rendering it inviable at fine-grain time slices (sub 10 ms time quanta). Moreover, per core on-chip voltage regulators adds significant area overhead leading to a sub-optimal chip design. In Chapter 4, we leverage a two-tier hierarchical power manager due to its low overhead and high scalability on a tiled many-core architecture with shared LLC and per-tile DVFS at fine-grain time slices (1 ms time quanta for micro-architectural adaptations). We use the Chrysso analytical performance and power models and tune it for the shared architecture and its adaptation. The power is distributed using two-tier power manager where global power is first distributed across tiles using GPM and then within a tile (in parallel across all tiles). We also show that thread scheduling is essential in such an architecture to account for thread sensitivity towards shared resources. We leverage DVFS and cache-aware thread migration (DCTM) to ensure optimum per-tile co-scheduling of compatible threads at runtime over the two-tier hierarchical power manager. Using simple two-tier hierarchical power manager and *DCTM* in tandem, we report that system performance improves by 10% on average, and up to 20%. *DCTM* outperforms existing solutions by 4.2% on average (and up to 12%) while using the same underlying two-tier hierarchical power manger on adaptive many-core processor.

In the second part of the thesis, we have addressed the memory-wall issue with respect to many-core processors. With the advancements in the core micro-architectural techniques and technology scaling, the performance gap between the computational component and memory component is increasing significantly. To bridge this gap, the architecture community is pushing forward towards multi-core architecture with on-die near-memory DRAM cache memory (faster than conventional DRAM). Large die-stacked DRAM caches are clearly needed for many-core processors to bridge the memory wall problem. This poses a problem of how to efficiently manage the tags associated with gigascale DRAM Caches. Prior works have proposed innovative Tags-in-DRAM designs to efficiently co-locate tags with data. Unfortunately, Tags-in-DRAM designs with multi-way associativity suffer from high latency. To mitigate the latency, prior works have proposed structures that either mitigate tag lookup latency in DRAM Cache or bypass the request to off-chip DRAM. In Chapter 5, we have shown that in many-core processor with gigascale die-stacked DRAM Cache, these approaches would not only require more resources but could potentially become a source of contention, thereby limiting overall system performance. We propose Tag Cache, an on-chip distributed tag caching mechanism with limited space and latency overhead to bypass the tag read operation in multi-way DRAM Caches, thereby reducing hit latency. Each Tag Cache, stored in L2, stores tag information of the most recently used DRAM Cache ways. We show that the Tag Cache is able to exploit temporal locality of the DRAM Cache, thereby contributing to on average 46% of the DRAM Cache hits while running multi-programmed workloads.

6.3 Potential Future Work

In this section we discuss the potential future work. With the results obtained an based on our analysis in the previous section, we find numerous opportunities for further research. In this section, we discuss briefly our proposal for further work.

6.3.1 Adaptive SMT Cores

In the first part of the thesis presented the analytical performance and power models for adaptive micro-architecture. We evaluated these models for cores without simultaneous multi-threading (SMT). A possible extension of this work could be to extend the core performance and power model for simultaneous multi-threading (SMT) cores. This is particularly interesting as most high performance core support simultaneous multi-threading. The adaptation of SMT core in a power constrained scenario would open challenges in optimising performance. This would also call for a more advanced thread migration scheme on a many-core tiled architecture. The thread migration module would need to optimize thread placement based on three available shared resources, namely, core (SMT), DVFS and shared LLC, to maximize overall performance.

6.3.2 Power Management

We have shown that runtime micro-architectural adaptations provide various operation power-performance points for various power envelopes without shutting down the cores. This provide better overall performance with much higher operational flexibility. Perhaps an implementation of an FGPA/firmware-based power manager would provide the first hand insight towards scope and operational properties. Moreover, an experimental prototype can also help to improve the analytical models to provide better accuracy. This would assist the power manager to converge faster thereby increasing scalability. Additional die temperature values as controlled information can be added as feedback to the PI/PID controller to improve the power distribution. This would also enable temperature-aware micro-architectural adaptation to not only manage heat dissipation but also prevent hot-spots for increasing the lifetime of the many-core processor. Moreover, further work can be carried towards making the analytical model-driven power management (both centralized and hierarchical) more generic. For instance, extending the scope to not only include variations of CPUs but also on-die GPUs, accelerators, large DRAM Caches, etc.

6.3.3 Resizeable DRAM Cache

Although the near-memory architecture is aimed to reduce the performance gap between the computing component and memory component, it still dissipate a substantial amount of power. Further research could be focused towards creating analytical models to project core/application performance and power values during DRAM Cache adaptation at different power envelopes.

6.4 Work published

[CF 2015] Sudhanshu Shekhar Jha, Wim Heirman, Ayose Falcón, Trevor Carlson, Kenzo Van Craeynest, Jordi Tubella, Antonio González, and Lieven Eeckhout "*Chrysso: An Integrated Power Manager for Constrained Many-Core Processors*", in proceedings of the ACM International Conference on Computing Frontiers (CF 2015).

[CF 2016] Sudhanshu Shekhar Jha, Wim Heirman, Ayose Falcón, Jordi Tubella, Antonio González, and Lieven Eeckhout "*Shared Resource Aware Scheduling on Power-Constrained Many-Core Tiled Processors*", in proceedings of the ACM International Conference on Computing Frontiers (CF 2016 Poster).

[Under Review] Sudhanshu Shekhar Jha, Wim Heirman, Ayose Falcón, Jordi Tubella, Antonio González, and Lieven Eeckhout "*Shared Resource Aware Scheduling on Power-Constrained Many-Core Tiled Processors*", under review.

[Under Review] Sudhanshu Shekhar Jha, Ayose Falcón, Jordi Tubella, Antonio González, and Lieven Eeckhout "*Tag Caching for Gigascale Die-Stacked DRAM Caches: A High-Performance Many-Core Perspective*", under review.

References

[1] Jaume Abella and Antonio González. Power-aware adaptive issue queue and register file. In *Proceedings of the High Performance Computing (HiPC)*, pages 34–43. 2003.

[2] David H Albonesi. Dynamic IPC/clock rate optimization. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 282–292, 1998.

[3] David H Albonesi. The inherent energy efficiency of complexity-adaptive processors. In *Proceedings of the Power-Driven Microarchitecture Workshop*, pages 107–112, 1998.

[4] David H Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*, pages 248–259, November 1999.

[5] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, pages 49–58, December 2003.

[6] AnandTech. Intel readying 15-core Xeon E7 v2. http://www.anandtech.com/show/7753/intel-readying-15core-xeon-e7-v2, February 2014.

[7] AnandTech. Quick Note: Intel Knights Landing Xeon Phi & Omni-Path 100 @ ISC 2015. http://www.anandtech.com/show/9436/quick-note-intel-knights-landing-xeon-phi-omnipath-100-isc-2015, 2015.

[8] Carl J Anderson, J Petrovick, JM Keaty, J Warnock, G Nussbaum, JM Tendier, C Carter, S Chu, J Clabes, J DiLullo, et al. Physical design of a fourth-generation POWER GHz microprocessor. In *Proceedings of the International Symposium on Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 232–233, 2001.

[9] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[10] Vishal Aslot and Rudolf Eigenmann. Performance characteristics of the SPEC OMP2001 benchmarks. *ACM SIGARCH Computer Architecture News*, pages 31–40, December 2001.

[11] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the Conference on Supercomputing*, pages 176–186, 1991.

[12] R Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 218–229, June 2001.

[13] Robert Bai, Nam-Sung Kim, Dennis Sylvester, and Trevor Mudge. Total leakage optimization strategies for multi-level caches. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, pages 381–384, 2005.

[14] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, pages 63–73, September 1991.

[15] R Jacob Baker. *CMOS: circuit design, layout, and simulation*, volume 1. John Wiley & Sons, 2008.

[16] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Dynamic memory hierarchy performance optimization. In *Workshop on Solving the Memory Wall Problem*, 2000.

[17] Rajeev Balasubramonian, David H Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Transactions on Computers,*, 52(10):1243–1258, 2003.

[18] Amirali Baniasadi and Andreas Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 16–21, 2001.

[19] Christopher Batten, Ajay Joshi, Jason Orcutt, Anatoly Khilo, Benjamin Moss, Charles Holzwarth, Milos Popović, Hanqing Li, Henry Smith, Judy Hoyt, Franz Kärtner, Rajeev Rajeev, Ram, Vladimir Stojanovìc, and Krste Asanovi. Building many core processor-to-dram networks with monolithic silicon photonics. In *Proceedings of the 16th IEEE Symposium on High Performance Interconnects (HOTI)*, pages 21–30. IEEE, 2008.

[20] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[22] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*, pages 318–329, November 2008.

[23] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, David McCauley, Pat Morrow, Donald W Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die stacking (3D) microarchitecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–479, 2006.

[24] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[25] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, pages 746–749, 2007.

[26] Pradip Bose, D Brooks, Alper Buyuktosunoglu, P Cook, K Das, P Emma, Michael Gschwind, H Jacobson, Tejas Karkhanis, Prabhakar Kudva, et al. Early-stage definition of LPX: a low power issue-execute processor. In *Power-Aware Computer Systems*, pages 1–17. Springer, 2002.

[27] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. Hierarchical scheduling windows. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, pages 27–36, 2002.

[28] Dominik Brodowski and Nico Golde. Linux cpufreq governors. *Linux Kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt*, 2013.

[29] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 171–182, January 2001.

[30] David M Brooks, Pradip Bose, Stanley E Schuster, Hans Jacobson, Prabhakar N Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[31] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. FIVR – Fully integrated voltage regulators on 4th generation Intel® Core™ SoCs. In *Proceedings of the 29th Applied Power Electronics Conference and Exposition (APEC)*, pages 432–439, 2014.

[32] J Adam Butts and Gurindar S Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 191–201, 2000.

[33] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, pages 73–78, 2001.

[34] Alper Buyuktosunoglu, David H Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. Power-efficient issue queue design. In *Power Aware Computing*, pages 35–58. Springer, 2002.

[35] Alper Buyuktosunoglu, Tejas Karkhanis, David H Albonesi, and Pradip Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 147–156, June 2003.

[36] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, November 2011.

[37] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[38] Anantha P Chandrakasan and Robert W Brodersen. *Low power digital CMOS design*. Springer Science & Business Media, 2012.

[39] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the ACM International Conference on Supercomputing*, pages 402–412, 2014.

[40] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 198–210, 2015.

[41] Pat Conway and Bill Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[42] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. In *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO)*, pages 143–154, December 2012.

[43] 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*, pages 256–268, 1974.

[44] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

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

[46] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and ML Scottt. Integrating adaptive on-chip storage structures for reduced dynamic power. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 141–152, 2002.

[47] Kristof Du Bois, Stijn Eyerman, Jennifer B Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[48] Kristof Du Bois, Jennifer B Sartor, Stijn Eyerman, and Lieven Eeckhout. Bottle graphs: visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 355–372, October 2013.

[49] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*, pages 485–496, December 2010.

[50] Thomas Ebi, M Faruque, and Jörg Henkel. TAPE: Thermal-aware agent-based power economy multi/many-core architectures. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 302–309, 2009.

[51] Yasuko Eckert, Srilatha Manne, Michael J Schulte, and David A Wood. Something old and something new: P-states can borrow microarchitecture techniques too. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 385–390, July 2012.

[52] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[53] ExtremeTech. AMD unveils Radeon R9 Nano: HBM and Fury X in a 6-inch GPU. http://www.extremetech.com/gaming/212973-amd-announces-new-radeon-r9-nano-hbm-and-fury-x-in-a-six-inch-gpu, 2015.

[54] Stijn Eyerman and Lieven. Eeckhout. System-level performance metrics for multiprogram workloads. *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*, pages 42–53, May 2008.

[55] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, pages 1–37, May 2009.

[56] Marco Facchini, Trevor Carlson, Anselme Vignon, Martin Palkovic, Francky Catthoor, Wim Dehaene, Luca Benini, and Paul Marchal. System-level power/performance evaluation of 3D stacked DRAMs for mobile applications. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 923–928, 2009.

[57] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 13–23, June 2007.

[58] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 148–157, May 2002.

[59] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 230–239, June 2001.

[60] Sean Franey and Mikko Lipasti. Tag Tables. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 514–525, 2015.

[61] Yang Ge, Qinru Qiu, and Qing Wu. A multi-agent framework for thermal aware task migration in many-core systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(10):1758–1771, 2012.

[62] Hamid Reza Ghasemi and Nam Sung Kim. RCS: Runtime resource and core scaling for power-constrained multi-core processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 251–262, August 2014.

[63] Dan Gibson and David A. Wood. Forwardflow: A scalable core for power-constrained CMPs. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–25, June 2010.

[64] Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. Automatic tuning of two-level caches to embedded applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, page 10208, 2004.

[65] Kate Greene. A new and improved Moore's law. MIT Technology Review. http://www.technologyreview.com/news/425398/a-new-and-improved-moores-law/, 2011.

[66] Michael Gschwind, H Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, (2):10–24, 2006.

[67] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 38–50, 2014.

[68] Stephen Gunther, Frank Binns, Douglas M Carmean, and Jonathan C Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5:1–9, 2001.

[69] Selim Gurun and Chandra Krintz. Autodvs: an automatic, general-purpose, dynamic clock scheduling system for hand-held devices. In *Proceedings of the International Conference on Embedded Software*, pages 218–226, 2005.

[70] T Halfltill. Transmeta breaks x86 low-power barrier. *Microprocessor Report*, 14(2), 2000.

[71] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31:6–15, 2011.

[72] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[73] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, 2007.

[74] HPCWire. Micron, Intel Reveal Memory Slice of Knight's Landing. http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/, 2014.

[75] Cheng-Chieh Huang and Vijay Nagarajan. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 51–60, 2014.

[76] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 878–883, June 2004.

[77] Intel. 2nd generation Intel Core vPro processor family. http://www.intel.com/content/dam/doc/white-paper/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf, September 2008.

[78] Intel. Intel® Xeon Phi™ Processor "Knights Landing" Architectural Overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf, 2015.

[79] IRC Committee. The international technology roadmap for semiconductors: 2013 edition, IRC Overview, 2013.

[80] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, pages 347–358, December 2006.

[81] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 158–168, 2002.

[82] Erik Jacobsen, Eric Rotenberg, and James E Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO)*, pages 142–152, 1996.

[83] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–260, 2012.

[84] Ramkumar Jayaseelan and Tulika Mitra. A hybrid local-global approach for multi-core thermal management. In *Proceedings of the International Conference on Computer-Aided Design (ICCD)*, pages 314–320, November 2009.

[85] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Proceedings of the Symposium on VLSI Technology (VLSIT)*, 2012.

[86] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[87] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 25–37, 2014.

[88] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[89] Sudhanshu S. Jha, Wim Heirman, Ayose Falcón, Trevor E. Carlson, Kenzo Van Craeynest, Jordi Tubella, Antonio González, and Lieven Eeckhout. Chrysso: An integrated power manager for constrained many-core processors. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, May 2015.

[90] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. CHOP: Adaptive filter-based dram caching for CMP server platforms. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[91] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, March 2012.

[92] Joel Hruska (ExtemeTech). Intel formally kills its tick-tock approach to processor development. http://www.extremetech.com/extreme/225353-intel-formally-kills-its-tick-tock-approach-to-processor-development, March 2016.

[93] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[94] Ron Kalla, Balaram Sinharoy, and Joel M Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[95] Tejas Karkhanis, James E Smith, and Pradip Bose. Saving energy with just in time instruction delivery. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 178–183, 2002.

[96] Stefanos Kaxiras and Margaret Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[97] Kamil Kędzierski, Francisco J Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. Power and performance aware reconfigurable cache for cmps. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, page 1, 2010.

[98] Taeho Kgil, Shaun D'Souza, Ali Saidi, Nathan Binkert, Ronald Dreslinski, Trevor Mudge, Steven Reinhardt, and Krisztian Flautner. Picoserver: using 3d stacking technology to enable a compact energy efficient chip multiprocessor. *ACM SIGARCH Computer Architecture News*, 34(5):117–128, 2006.

[99] Chris Kim and Kaushik Roy. Dynamic Vth scaling scheme for active leakage power reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, page 163, 2002.

[100] Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong Ryol Hwang, Sooman Hwang, Byongwook Na, Joungwook Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyoon Cho, Minsoo Jang, Cheolmin

Han, Jung-Bae Lee, Joo Sun Choi, and Young-Hyun Jun. A 1.2 V 12.8 GB/s 2 Gb Mobile Wide-I/O DRAM With 4 128 I/Os Using TSV Based Stacking. *IEEE Journal of Solid-State Circuits*, 47(1):107–116, 2012.

[101] Nam Sung Kim, David Blaauw, and Trevor Mudge. Leakage power optimization techniques for ultra deep sub-micron multi-level caches. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCD)*, page 627, 2003.

[102] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, 2004.

[103] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[104] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–134, February 2008.

[105] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[106] Yuya Kora, Kyohei Yamaguchi, and Hideki Ando. MLP-aware Dynamic Instruction Window Resizing for Adaptively Exploiting Both ILP and MLP. In *Proceedings of the 46th Annual International Symposium on Microarchitecture (MICRO)*, pages 37–48, 2013.

[107] Harish K Krishnamurthy, Vaibhav A Vaidya, Pavan Kumar, George E Matthew, Sheldon Weng, Bharani Thiruvengadam, Wayne Proefrock, Krishnan Ravichandran, and Vivek De. A 500 MHz, 68% efficient, fully on-die digitally controlled buck voltage regulator on 22nm Tri-Gate CMOS. In *Proceedings of the IEEE Symposium on VLSI Circuits Digest of Technical Papers*, pages 1–2, 2014.

[108] Nasser Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P Thomas, Christopher Mozak, Brent Boswell, Manoj Lal, Anant Deval, Jonathan Douglas, Mahmoud Elassal, Ankireddy Nalamalpu, Timothy M. Wilson, Matthew Merten, Srinivas Chennupaty, Wilfred Gomes, and Kumar Rajesh. 5.9 Haswell: A family of IA 22nm processors. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 112–113, 2014.

[109] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, pages 48–51, February 2009.

[110] Jian Li and Jose F Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 77–87, 2006.

[111] Sheng Li, J Ahn, Jay B Brockman, and Norman P Jouppi. McPAT 1.0: An integrated power, area, and timing modeling framework for multicore architectures. *HP Labs*, 2009.

[112] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[113] Song Liu, Brian Leung, Alexander Neckar, Seda Ogrenci Memik, Gokhan Memik, and Nikos Hardavellas. Hardware/software techniques for DRAM thermal management. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 515–525, 2011.

[114] Gabriel H Loh. 3d-stacked memory architectures for multi-core processors. 36(3):453–464, 2008.

[115] Gabriel H Loh. Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 201–212, 2009.

[116] Gabriel H Loh and Mark D Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 454–464, 2011.

[117] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, August 2001.

[118] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 449–460, 2011.

[119] Niti Madan, Li Zhao, Naveen Muralimanohar, Aniruddha Udipi, Rajeev Balasubramonian, Ravishankar Iyer, Srihari Makineni, and Donald Newell. Optimizing communication and capacity in a 3D stacked reconfigurable cache hierarchy. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 262–274, 2009.

[120] Afzal Malik, Bill Moyer, and Dan Cermak. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 241–243, 2000.

[121] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 132–141, 1998.

[122] Diana Marculescu. Profile-driven code execution for low power dissipation. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 253–255, 2000.

[123] Ke Meng, Russ Joseph, Robert P Dick, and Li Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 177–186, 2008.

[124] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[125] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Computer Architecture Letters*, 11(2):61–64, 2012.

[126] Micron. TN-41-01: Technical Note, Calculating Memory System Power for DDR3, 2008.

[127] Timothy N Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, and Radu Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, February 2012.

[128] Asit K Mishra, Shekhar Srikantaiah, Mahmut Kandemir, and Chita R Das. CPM in CMPs: Coordinated power management in chip-multiprocessors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2010.

[129] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 144–116, April 1965.

[130] T.Y. Morad, U.C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, pages 14–17, January 2006.

[131] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 174:1–174:9, 2013.

[132] Koji Nii, Hiroshi Makino, Yoshiki Tujihashi, Chikayoshi Morishima, Yasushi Hayakawa, Hiroyuki Nunogami, Takahiko Arakawa, and Hisanori Hamano. A low power SRAM using auto-backgate-controlled MT-CMOS. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 293–298, 1998.

[133] Venkatesh Pallipadi. Enhanced intel speedstep technology and demand-based switching on linux. *Intel Developer Service*, 2009.

[134] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large Intel Itanium; programs with dynamic instrumentation. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, pages 81–92, December 2004.

[135] J Thomas Pawlowski. Hybrid memory cube (HMC). 2011.

[136] M Peckerar, R Fulton, P Blaise, D Brown, and R Whitlock. Radiation effects in MOS devices caused by X-ray and e-beam lithography. *Journal of Vacuum Science & Technology*, 16(6):1658–1661, 1979.

[137] MC Peckerar, CM Dozier, DB Brown, D Patterson, D McCarthy, and D Ma. Radiation effects introduced by X-ray lithography in MOS devices. *IEEE Transactions on Nuclear Science*, 29(6):1697–1701, 1982.

[138] Pavlos Petoumenos, Georgia Psychou, Stefanos Kaxiras, Juan Manuel Cebrian Gonzalez, and Juan Luis Aragon. Mlp-aware instruction queue resizing: the key to power-efficient performance. In *Architecture of Computing Systems (ARCS)*, pages 113–125. 2010.

[139] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 13–23, June 2013.

[140] Shlomit S Pinter and Adi Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO)*, pages 214–225, 1996.

[141] Fred J Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address). In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*, page 2, 1999.

[142] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Dynamic allocation of datapath resources for low power. *Proceedings of the Workshop on Complexity-Effective Design*, 2001.

[143] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 90–95, 2000.

[144] Moinuddin K Qureshi and Gabriel H Loh. Fundamental latency trade-off in architecting DRAM Caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 235–246, 2012.

[145] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

[146] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO)*., pages 16–27, 1999.

[147] Brian M Rogers, Anil Krishna, Gordon B Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 371–382, 2009.

[148] Alberto Ros, Jose M Garcia, and Manuel E Acacio. *Cache coherence protocols for many-core CMPs*. INTECH Open Access Publisher, 2010.

[149] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, pages 20–27, March 2012.

[150] Jeffrey B Rothman and Alan Jay Smith. Sector cache design and performance. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 124–133, 2000.

[151] Sam Naffziger (AMD). AMD Keynoter Talks CPU-GPU Integration at the VLSI Symposium. http://www.monolithic3d.com/blog/amd-keynoter-talks-cpu-gpu-integration-at-the-vlsi-symposium, August 2011.

[152] Ruchira Sasanka, Christopher J Hughes, and Sarita V Adve. Joint local and global hardware adaptations for energy. *ACM SIGARCH Computer Architecture News*, 30(5):144–155, 2002.

[153] André Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *ACM SIGARCH Computer Architecture News*, volume 22, pages 384–393, 1994.

[154] Akbar Sharifi, Asit K Mishra, Shekhar Srikantaiah, Mahmut Kandemir, and Chita R Das. PEPON: Performance-aware hierarchical power budgeting for NoC based multicores. In *Proceedings of the 21st International conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 65–74, 2012.

[155] Abel Guilhermino Silva-Filho and Filipe Rolim Cordeiro. A combined optimization method for tuning two-level memory hierarchy considering energy consumption. *Journal on Embedded Systems EURASIP*, 2011:2, 2011.

[156] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 247–257, 2012.

[157] Kirk Skaugen. Petascale to Exascale: Extending Intel's HPC commitment. In *Proceedings of the International Supercomputing Conference (ISC)*, volume 10, 2010.

[158] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, November 2000.

[159] Avinash Sodani. Knights Landing: 2nd Generation Intel “Xeon Phi" Processor. In *Proceedings of the 27th Hot Chips Symposium (HC)*, August 2015.

[160] Jared Stark, Mary D Brown, and Yale N Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 57–66, 2000.

[161] N. Sturcken, M. Petracca, S. Warren, P. Mantovani, L.P. Carloni, A.V. Peterchev, and Kenneth L. Shepard. A switched-inductor integrated voltage regulator with nonlinear feedback and network-on-chip load in 45 nm soi. *IEEE Journal of Solid-State Circuits*, pages 1935–1945, Aug 2012.

[162] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[163] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with Asymmetric Multi-core Architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, March 2009.

[164] Karthik T Sundararajan, Vasileios Porpodas, Timothy M Jones, Nigel P Topham, and Böjrn Franke. Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs. In *Proceedings of the Annual International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[165] Yuan Taur and Tak H Ning. *Fundamentals of modern VLSI devices*. Cambridge university press, 2013.

[166] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[167] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 363–374, June 2008.

[168] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403, 1995.

[169] Osman S Unsal and C Mani Krishna. Cool-Fetch: Compiler-enabled power-aware fetch throttling. *IEEE Computer Architecture Letters*, 1(1):5–5, 2002.

[170] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Siram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 98–589, 2007.

[171] Keshavan Varadarajan, SK Nandy, Vishal Sharda, Amrutur Bharadwaj, Ravi Iyer, Srihari Makineni, and Donald Newell. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 433–442, 2006.

[172] Hong Wang, Tong Sun, and Qing Yang. CAT — Caching Address Tags: A technique for reducing area cost of on-chip caches. *ACM SIGARCH Computer Architecture News*, 23(2):381–390, 1995.

[173] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 314–324, 2009.

[174] Jonathan A Winter and David H Albonesi. Addressing thermal nonuniformity in SMT workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):4, 2008.

[175] Dong Hyuk Woo, Nak Hee Seong, Dean L Lewis, and Hsien Hsin S Lee. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[176] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 248–259, 2004.

[177] Wm A Wulf and Sally A McKee. Hitting the Memory Wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[178] Se-Hyun Yang, Michael D Powell, Babak Falsafi, and TN Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 151–161, February 2002.

[179] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014.

[180] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 136–146, June 2003.

[181] Li Zhao, Ravi Iyer, Ramesh Illikkal, and Don Newell. Exploring dram cache architectures for cmp server platforms. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 55–62, 2007.