ABSTRACT

Title of dissertation: GREEDY COORDINATE DESCENT CMP MULTI-LEVEL CACHE RESIZING

Inseok Stephen Choi
Doctor of Philosophy, 2014

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Hardware designers are constantly looking for ways to squeeze waste out of architectures to achieve better power efficiency. Cache resizing is a technique that can remove wasteful power consumption in caches. The idea is to determine the minimum cache a program needs to run at near-peak performance, and then reconfigure the cache to implement this efficient capacity. While there has been significant previous work on cache resizing, existing techniques have focused on controlling resizing for a single level of cache only. This sacrifices significant opportunities for power savings in modern CPU hierarchies which routinely employ 3 levels of cache. Moreover, as CMP scaling will likely continue for the foreseeable future, eliminating wasteful power consumption from a CMP multi-level cache hierarchy is crucial to achieve better power efficiency.

In this dissertation, we propose a noble technique, *greedy coordinate descent CMP multi-level cache resizing*, that minimizes a power consumption while maintaining a high performance. We simutaneously resizes all caches in a modern CMP

cache hierarchy to minimize the power consumption. Specifically, our approach predicts the power consumption and the performance level without direct evaluations. We also develop *greedy coordinate descent* to search an optimal cache configuration utilizing *power efficiency gain* (PEG) that we propose in this dissertation.

This dissertation makes three contributions for a CMP multi-level cache resizing. First, we discover the limits of power savings and performance. This limit study identifies the potential power savings in a CMP multi-level cache hierarchy when wasteful power consumption is eliminated. Second, we propose a prediction-based *greedy coordinate descent* (GCD) method to find an optimal cache configuration and to orchestrate them. Third, we implement online GCD technique for a CMP multi-level cache resizing. Our approach exhibits 13.9% power savings and this achieves 91% of the power savings of the static oracle cache hierarchy configuration.

GREEDY COORDINATE DESCENT CMP MULTI-LEVEL CACHE RESIZING

by

Inseok Stephen Choi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:

Professor Donald Yeung, Chair/Advisor
Professor Manoj Franklin
Professor Gang Qu
Professor Rejeev Barua
Professor Alan Sussman

To my wife, Katie, and my daughter, Emily.

To my parents and brother.

Acknowledgments

First and foremost, I really thank my advisor, Dr. Donald Yeung, for his long waiting with patience, for giving me an invaluable opportunity to work on challenging projects, and for providing me an enormous amount of computing resources to support my extensive simulations. He brought me to better-shaped and more matured reasoning processes all the time. It has been a pleasure to work with and learn from him. I would like to thank Dr. Manoj Franklin. He was my academic advisor and provided many guidelines in my graduate study and TA opportunities for his classes. I can not forget his heart-felt advice from his life about work-life balance. I would also like to thank Dr. Gang Qu. He provided and shared many interesting research topics in low-power systems and trustworthy computing. Thanks are due to Dr. Rajeev Barua and Dr. Alan Sussman for being my committee members and sparing their invaluable time for serving in the committee.

During my graduate life in Maryland, I was so lucky to meet incredible people. I am truly thankful for being my friend to Sunwoo Kim and Sukhyun Song sharing many perspectives of life and joys and sorrows of Ph.D. life. My fellow PhD students should have my special thanks to offering good friendship and being together during the tough time.

Most importantly, I owe my utmost gratitude to my wife, Seoyeon. I am truly blessed to have such a wonderful wife. I would like to thank my lovely daughter, Emily Dahay, for being my source of happiness and joy. I am extremely grateful to Dr. Luca Vricella for his great heart surgery for Emily. I specially thank my parents

and brother for their endless love. I was able to finish my long journey because of their enormous amount of support and endless prayers to God for me. I sincerely thank God for his endless love and care so that I can grow in his grace with Him.

Table of Contents

List of Tables viii

List of Figures ix

List of Abbreviations xii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Important Problems in Cache Resizing . . . 2
  - 1.2.1 Multi-Level Cache Resizing . . . 3
  - 1.2.2 Power Efficient Cache Partitioning . . . 4
  - 1.2.3 CMP Multi-level Cache Resizing . . . 4
- 1.3 Thesis Statement and Contributions . . . 5
- 1.4 Thesis Organization . . . 8

2 Background and Motivation 9
- 2.1 Chip Multi-Processors . . . 9
- 2.2 Multi-level Caches . . . 12
- 2.3 Cache Resizing . . . 12
  - 2.3.1 Multi-Level Cache Resizing . . . 14
  - 2.3.2 Optimal Multi-level Caches . . . 16
  - 2.3.3 Dynamic Reconfiguration . . . 17
- 2.4 Power Efficient Cache Partitioning . . . 20
- 2.5 Multi-Level Cache Resizing in CMPs . . . 21

3 Related Work 23
- 3.1 Cache Resizing . . . 23
- 3.2 Cache Partitioning . . . 25
- 3.3 Multi-level Cache Optimization . . . 27
- 3.4 Circuit-level and Design-time Optimization . . . 28

4 Multi-Level Cache Resizing Behavior Analysis 32
- 4.1 Complexity of Multi-Level Cache Resizing . . . 32
- 4.2 Power and Performance Impact of Varying Cache Sizes . . . 33
- 4.3 Experimental Methodology . . . 35
  - 4.3.1 Single-program Workload Generation . . . 35
  - 4.3.2 Performance Simulation . . . 37
  - 4.3.3 Power Simulation . . . 39
- 4.4 Limits of Multi-Level Cache Resizing . . . 41

5 Greedy Coordinate Descent Method 50
5.1 Analytical Model for MCR . . . 50
5.2 Performance Approximation . . . 52
5.3 Stack Distance Measurement . . . 52
5.3.1 Stack Distance . . . 52
5.3.2 Way Counters . . . 54
5.4 Power Efficiency Gain . . . 55
5.5 Greedy Coordinate Descent Method . . . 58

6 GCD Uniprocessor Multi-Level Cache Resizing 64
6.1 GCD Uniprocessor MCR Algorithm Design . . . 64
6.1.1 GCD MCR Algorithm Description . . . 64
6.1.2 Scalability of GCD MCR Algorithm . . . 67
6.1.3 Hardware Overhead . . . 72
6.1.4 Implementation . . . 74
6.2 Experimental Methodology . . . 77
6.3 Evaluation of the GCD Uniprocessor MCR . . . 80
6.3.1 Offline Analysis . . . 80
6.3.2 Discussion . . . 86
6.3.3 Online Analysis . . . 91
6.3.4 Discussion . . . 98

7 PEG-based Last Level Cache Partitioning 108
7.1 Last Level Cache Partitioning Problem . . . 108
7.2 Experimental Methodology . . . 110
7.2.1 Simulator . . . 110
7.2.2 Metrics . . . 111
7.2.3 Multiprogrammed Workloads . . . 112
7.3 Limits of Last Level Cache Partitioning . . . 112
7.4 PEG-based Cache Partitioning Algorithm Design . . . 118
7.5 Evaluation of PEG-based Cache Partitioning . . . 122
7.5.1 Offline Analysis . . . 122
7.5.2 Online Analysis . . . 128

8 GCD CMP Multi-Level Cache Resizing 131
8.1 Limits of CMP Multi-Level Cache Resizing . . . 131
8.2 Dividing CMP Multi-level Cache Resizing into Subproblems . . . 141
8.3 GCD CMP Multi-Level Cache Resizing Algorithm Design . . . 146
8.3.1 Algorithm Description . . . 147
8.3.2 Implementation . . . 148
8.4 Evaluation of GCD CMP Multi-level Cache Resizing . . . 152
8.4.1 Offline Analysis . . . 152
8.4.2 Online Analysis . . . 155

9 Conclusion 158
9.1 Summary and Conclusion 158
9.2 Future Work 159

Bibliography 162

List of Tables

2.1 Server processors specifications. 11
2.2 Cache hierarchy of modern commodity processor. 12

4.1 Multi-level cache resizing problem space scaling. 33
4.2 Benchmarks classification based on misses per kilo instructions (MPKI) of 2MB LLC. 38
4.3 Architectural configuration. 40
4.4 Cache parameters for the baseline multi-level caches. 42
4.5 Reconfigurable cache configurations (set:ways). For example, #7 configuration of L2 is 512:7 (set number is omitted if it is same to the set number of a previous configuration). In total, 67,584 simulations are conducted. 43
4.6 Statically optimal multi-level configurations. 44

5.1 Optimization technique classification. 59

6.1 Storage overhead of shadow tags. 74
6.2 AMAT and IPC of static optimal (SO) and ideal GCD MCR (I-MCR). 91
6.3 Cache configurations of static optimal (SO), GCD MCR (MCR), and ideal GCD MCR (I-MCR). 92
6.4 Number of cache reconfigurations of GCD MCR 99
6.5 Parameters to test the correlation of the improvement of the power savings in online GCD MCR. 101
6.6 Parameters to test the correlation of the performance degradation levels in online GCD MCR. 102
6.7 Correlation coefficient of each metric. * indicates statistical significance at 10% level. 103

7.1 Multi-program workloads. Each benchmark is shown in abbreviations according to the benchmark classification based on MPKI in Chapter 4. Benchmark classifications are shown in Table 4.2. For example, G2-A workload consists of m_0 and m_2 and these are *bzip2* and *zeusmp* as shown in Table 4.2, respectively. 113
7.2 Cache partition and power consumptions of LLC and DRAM. 124
7.3 Cache partition from UCP, U+PD, and I-PCP-2. 128
7.4 Throughput of UCP according to the epoch size for G2 workloads. . . 129

8.1 Performance and power of static optimal CMP cache configurations from exhaustive search and Nelder-Mead simplex search with 1% performance degradation. All values are normalized to the result of the even partitioning (EP). 134
8.2 CMP cache configurations from exhaustive search and Nelder-Mead simplex search with 1% performance degradation. 135

8.3 Average relative power consumption per caching level and per dynamic/static power. Values are normalized to the power consumption of UCP. . . . 138
8.4 Cache configurations from uniprocessor static optimal and CMP static optimal. . . . 142
8.5 Performance per workload groups. Normalized difference is calculated by Diff/(1-D-HG). . . . 157

List of Figures

2.1 Power consumption per core and per clock speed of modern commodity processors. [1] . . . 10
2.2 System power consumption breakdown of the workloads. . . . 15
2.3 Cache power consumption breakdown of the workloads. . . . 15
2.4 Lowest Power Consumption with Limited Performance Degradation. 18
2.5 Different optimal cache hierarchy configurations across SPEC CPU 2006 benchmarks. Due to the difficulties in plotting 3-dimensional result, we plot result for a 2-level cache hierarchy. Each trend line represents the power consumption of a corresponding L2 cache configuration. . . . 19

4.1 Best and worst performance of workloads by changing cache configurations. IPC is normalized to the IPC of the baseline configuration. . 34
4.2 Best and worst power consumption of workloads by changing cache configurations. Power is normalized to the power consumption of the baseline configuration. . . . 35
4.3 Simulation framework. We use AlphaVM, SimPoint, and Sim-EIO to generate our workloads. We use SimpleScalar for performance simulation. Lastly, we use MASTAR, CACTI, and McPAT to simulate power consumption of the target system. . . . 36
4.4 Power and performance comparison between exhaustive and per-level exhaustive searches. (For example, E-L1 only searches by resizing L1 cache capacity.) . . . 45
4.5 Power breakdown of workloads with low MPKI (first group). . . . 48
4.6 Power breakdown of workloads with low MPKI (second group). . . . 48
4.7 Power breakdown of workloads with medium MPKI. . . . 49
4.8 Power breakdown of workloads with high MPKI. . . . 49

5.1 LRU stack, stack distance, and distance counter. . . . 53
5.2 Way counter workflow. . . . 56
5.3 Way counter implementation. . . . 57
5.4 Comparison between *Coordinate Descent*, *Nelder-Mead*, and *Gradient Descent* method. Contour lines of objective function $f(x)$ and constraint function $g(x)$ are plotted in solid lines and dotted lines, respectively. . . . 61

6.1 MCR framework. 65
6.2 Power and performance comparison between random, Nelder-Mead, and MCR.) 81
6.3 Number of evaluations comparison between Nelder-Mead and MCR. 82
6.4 GCD multi-level cache resizing solution-search steps. 84
6.5 Relative power savings of GCD MCR (MCR), GCD MCR with priori information (P-MCR), and ideal GCD MCR (I-MCR) compared to the power savings of static optimal (SO). 85
6.6 Relative performance degradation level in percentile of GCD MCR (MCR), GCD MCR with priori information (P-MCR), and ideal GCD MCR (I-MCR). 85
6.7 Power consumption breakdown comparisons of static optimal (SO) and MCR for INT benchmarks. 87
6.8 Power consumption breakdown comparisons of static optimal (SO) and MCR for FP benchmarks. 88
6.9 Power and performance comparison between dynamic MCR and Nelder-Mead (with static MCR and static optimal). 93
6.10 Relative power savings of online GCD MCR (D-MCR) and static GCD MCR (S-MCR) compared to the power savings of static optimal (SO). 94
6.11 Relative performance degradation level in percentile of online GCD MCR (D-MCR) and static GCD MCR (S-MCR) compared to the power savings of static optimal (SO). 95
6.12 Power consumption breakdown comparisons of online GCD MCR for INT benchmarks. 96
6.13 Power consumption breakdown comparisons of online GCD MCR for FP benchmarks. 97
6.14 Dynamic cache reconfigurations of online GCD MCR in *libquantum* workload. 104
6.15 Dynamic cache reconfigurations of online GCD MCR in *bwaves* workload. 105
6.16 Dynamic cache reconfigurations of online GCD MCR in *calculix* workload. 105
6.17 Dynamic cache reconfigurations of online GCD MCR in *xalan* workload. 106
6.18 Dynamic cache reconfigurations of online GCD MCR in *h264ref* workload. 107

7.1 Power and system throughput comparison between exhaustive search to find the optimal partitioning for the best throughput (E), UCP (U), exhaustive LLC resizing search to find the optimal partitioning for the best power savings based on UCP with the performance degradation level of 1% (U+PD), COOP (C), and our PEG-based cache partitioning (P). Power and system throughput are normalized to the results of even partitioning (EP). 115

7.2 Power consumption breakdown for performance-oriented optimal (EXH), UCP, and exhaustive LLC resizing search to find the optimal partitioning for the best power savings with the performance degradation level of 1%(U+PD). 117
7.3 Power savings in LLC dynamic and static power consumption and dynamic power consumption of DRAM (compared to UCP). 117
7.4 Fairness comparison between exhaustive search to find the optimal partitioning for the best throughput (E), UCP (U), exhaustive LLC resizing search to find the optimal partitioning for the best power savings based on UCP with the performance degradation level of 1% (U+PD), COOP (C), and our PEG-based cache partitioning (P). Fairness is normalized to the fairness of even partitioning (EP). . . . 125
7.5 Power and system throughput comparison between PCP, I-PCP-1, and I-PCP-2. Power savings is normalized to the power savings of U+PD. Throughput is normalized to the throughput of UCP. 127
7.6 Power and system throughput comparison between online PCP, COOP, and PCP. 130

8.1 Power consumption and throughput of CMP multi-level cache resizing. Approximated by Nelder-Mead simplex method. 136
8.2 Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G2 workloads. 139
8.3 Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G4 workloads. 139
8.4 Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G8 workloads. 140
8.5 Cache power breakdown for baseline and static optimal (BL:baseline, UM:uniprocessor MCR, SO: static optimal with exhaustive search in CMP) 142
8.6 CMP multi-level cache resizing and variable grouping. 144
8.7 Convergence rate comparison between no gropuing, vertical grouping, and horizontal grouping. 146
8.8 GCD CMP multi-level cache resizing framework. 151
8.9 Power and system throughput comparison between static quasi-optimal (SQO), vertical grouping (VG), and horizontal grouping (HG). . . . 153
8.10 Power and system throughput comparison between static quasi-optimal (SQO), ideal vertical grouping (I-VG), and ideal horizontal grouping (I-HG). 155
8.11 Power and system throughput comparison between static quasi-optimal (SQO), static vertical grouping (S-VG), static horizontal grouping (S-HG), dynamic vertical grouping (D-VG), and dynamic horizontal grouping (D-HG). 156

List of Abbreviations

AMAT	Average Memory Access Time
CMP	Chip Multi-Processor
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
FBB	Forward Body Bias
GCD	Greedy Coordinate Descent
ILP	Instruction Level Parallelism
LLC	Last Level Cache
MCR	Multi-Level Cache Resizing
MTCMOS	Mutli-Threshold CMOS
NM	Nelder-Mead simplex
QoS	Quality-of-Service
RAM	Random-Access Memory
RBB	Reverse Body Bias
SDRAM	Synchronous Dynamic Random-Access Memory
TLP	Thread Level Parallelism
ZBB	Zero Body Bias

Chapter 1

Introduction

1.1 Motivation

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 [1]. Unfortunately, this problem will only get worse in the future as process technologies continue to scale to smaller feature sizes. Moreover, CMPs are prevalent to provide better performance in a power envelope; computer architects utilize efficient cores rather than exploiting instruction level parallelism (ILP) which incurs high power dissipation but yield only modest performance gains. As a result, CMP scaling–*i.e.* increasing the number of cores–will continue in the foreseeable future as transistor count increases. As such, power efficiency will remain an extremely important design goal.

A key place to look for better power efficiency is in an 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, 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. On the other hand, a cache hierar-

chy is crucial to achieve high performance in CMPs by eliminating off-chip traffics when efficiently utilized. As such, it is required that hardware designers continue to make efforts to squeeze wasteful power consumption out of the cache hierarchy and maintain high performance at the same time.

1.2 Important Problems in Cache Resizing

As both low power consumption and high performance are required, on-chip cache hierarchies face three challenging problems. First, controlling sizes of multi-level caches is imperative to eliminate wasteful power consumption because multi-level cache hierarchies distribute the power consumption across different caching levels. Controlling the size of a single level cache [2, 3, 4, 5, 6, 7, 8, 9, 10] will miss significant opportunities for power savings because L1 caches are the greatest culprit for dynamic power consumption of a cache hierarchy and last level caches (LLCs) are by far the greatest concern for static power consumption due to its large area. Second, resource distribution in an LLC gets crucial to achieve high system throughput by efficient utilization of the LLC. Otherwise, cache misses at the LLC incur more off-chip traffics that results in poorer system throughput. Third, controlling sizes of multi-level caches in CMPs is an *NP-hard* problem. As such, having a scalable algorithm is crucial to alleviate the *power inefficiency* in the cache hierarchy of CMPs.

1.2.1 Multi-Level Cache Resizing

The trend for modern CPUs is towards deeper cache hierarchies, however, which distributes the power consumption across many caching levels. Today, three levels of cache is commonplace. 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.

The current lack of comprehensive multi-level cache resizing is partly due to the latency tolerance of multi-level cache hierarchy. Because there are multi-caching levels, CPU performance is somewhat insensitive to its actual delay caused by additional cache misses when a single level of cache is resized: *e.g.* the additional cache misses from a L2 cache may hit in the next level of cache and result in only small increase in AMAT compared to a two-level cache hierarchy where those additional cache misses result in DRAM accesses that increase AMAT significantly. However, negligent multi-level cache resizing will eventually result in cascaded aggregate off-chip traffics. As a result, an improvident multi-level cache resizing will turn into significant performance degradation.

1.2.2 Power Efficient Cache Partitioning

Cache-partitioning techniques have been widely investigated mainly for higher performance [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. However, power efficient cache-partitioning has not been comprehensively studied yet. In particular, a shared LLC can be considers as a single cache, so we can apply existing cache resizing techniques to the LLC. A recent study [10] applies a cache resizing technique to a shared LLC in CMPs on top of the utility-based cache partitioning scheme [11]. This study explores the wasteful power consumption of an LLC, mostly from the static power consumption, thus reducing the power consumption of the LLC without noticeable performance degradation by not allocating ways with lower utility. However, this approach can not search the global optimum because its allocation paths are already defined in the UCP algorithm. In other words, disabling part of partitions that were distributed to achieve the best performance can not find power-efficient partitions. Therefore, such local-search-based/performance-oriented cache configurations is prone to results in local minima.

1.2.3 CMP Multi-level Cache Resizing

For the foreseeable future, the number of cores will likely grow. Moreover, multi-level cache hierarchy will continue to being crucial for bridging the processor-memory speed gap. Hence, CMP multi-level cache hierarchy is not only crucial to achieve higher performance, but also significant contributor in the total power dissipation for being towards deeper cache hierarchy to bridge the processor-memory

speed gap.

As such, eliminating wasteful power consumption from the CMP multi-level cache hierarchy while maintaining high throughput is extremely important problem. However, the current trend of CMP scaling will only exacerbate the complexity of this problem. Moreover, CMP multi-level cache resizing problem is an *NP-hard*. We study CMPs with a shared last-level cache and finding an optimal partitions in the shared last-level cache has been shown to be *NP-hard* [22]. And finding an optimal partitions is only a subroutine of finding optimal CMP multi-level cache hierarchy configurations. For this reason, we need a scalable solution to orchestrate all caches as by optimizing private caches as well as LLC partitions to eliminate wasteful power consumption in the entire system level, while maintaining high throughput.

1.3 Thesis Statement and Contributions

This thesis investigates *greedy coordinate descent* (GCD) CMP *multi-level cache resizing* (MCR) to minimize power consumption at all caching levels simultaneously at a given performance degradation limit. Our work quantifies the potential power benefits of uniprocessor MCR, LLC partitioning, and CMP MCR, providing insights into where savings come from as well as the challenges that must be overcome in order to attain the full benefits. We also investigate controlling uniprocessor MCR, LLC partitioning, and CMP MCR. Cache hierarchies with multiple reconfigurable caches exhibit a large number of resizing configurations. Our work develops techniques to navigate this complex search space to quickly find the best configura-

tions. In particular, we develop *greedy* search iterations in using coordinate descent method [23] to reduce the complexity of a multi-level cache resizing problem without direct power/performance evaluations. We define *power efficiency gain* (PEG) to select a caching level and to decide the amount of cache resizing in the caching level in searching the solution space. We compute a PEG value by predicting power consumption and performance changes utilizing way counters. In other words, we reduce the evaluation complexity by predicting performance and power by utilizing way-counter hardware. In particular, we reduce the evaluation complexity from $O(k^m)$ to $O(km)$, where k is the number of possible configurations per caching level and m is the number of caching level. More specifically, we make the following contributions.

First, we study the limits of power consumption and performance by discovering statically optimal cache hierarchy configurations to quantify the potential benefits of our approach. Our study presents a static-optimal version of uniprocessor MCR, LLC partitioning, and CMP MCR that use exhaustive off-line search to find the best configurations. In particular, we show that architectural techniques are still crucial to optimize power consumption in a cache hierarchy by employing the state-of-the-art circuit- and device-level techniques to reduce power consumption. We find static-optimal configurations for each application can reduce total cache hierarchy power consumption by 15.4%, 3.3%, and 15.2%, respectively, while degrading the performance by less than 1% across our workloads.

Second, we propose a way-counter-based performance/power prediction technique. Direct performance/power evaluation in a time-varying system introduces

significant errors due to dynamic program phase changes. This makes evaluation-based techniques difficult to succeed because such oscillations of performance and power generate an unstable system. As such, we predict performance and power utilizing way counters.

Third, we develop a novel search technique, *greedy coordinate descent*. Because direct methods or gradient-based methods are not achievable in the CMP multi-level cache resizing problem, we use *coordinate descent* method instead to solve this problem. Moreover, we propose our new technique, *greedy coordinate descent*, to solve this problem efficiently. We define *power efficiency gain* (PEG) to enable efficient and fast search in our technique. PEG not only provides power savings per unit performance degradation, but also considers a balance between two adjacent cacheing levels so that we can optimize towards the global optimal.

Fourth, we implement our GCD approach to find such global optimal configurations for uniprocessor MCR, LLC partitioning, and CMP MCR.

Fifth, we implement ideal GCD by eliminating errors caused by AMAT and power prediction using way counts. To quantify the impact of errors caused by our prediction, we use measured performance, power and AMAT information from our extensive simulations. We show that our ideal GCD approach can approximate the global optimal by 98%, 84%, and 94% compared to the power savings of the global optimal.

Fifth, we study online algorithms to enable our approach at runtime. Our online implementation exhibits fast convergence rates. As a result, our periodically repeated GCD search can adapt to dynamically changing program behavior. Our

results show that online GCD techniques save significant power savings, providing 13.4%, 1.8%, and 13.9% power savings for uniprocessor MCR, LLC partitioning, and CMP MCR, respectively, on average.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 explains the background of our study including chip multi-processors, multi-level caches, cache resizing, power efficient cache partitioning and multi-level cache resizing in chip multi-processors. Chapter 3 lists the previous studies related to our study covering cache resizing, cache partitioning, multi-level cache optimization, and circuit–level and design–time optimization. In Chapter 4 we analyze the multi-level cache resizing behavior to understand the complexity of multi-level cache resizing and the interaction between caching level when we resize them. Chapter 5 presents our greedy coordinate descent approach in details. In Chapter 6, we develop our GCD method for uniprocessor multi-level cache resizing and show the power savings of our GCD method that approximates the global optimal cache configuration both at offline and online. And then, we also apply our GCD method for last level cache partitioning and present the power savings in Chapter 7. In Chapter 8, we further extend our GCD method for CMP multi-level cache resizing and show the power savings. Finally, Chapter 9 concludes this dissertation and suggests future work.

Chapter 2

Background and Motivation

2.1 Chip Multi-Processors

The trend for modern CMPs is towards more cores [24, 25, 26, 27, 28]. However, current CPUs are constrained by power consumption [29, 1, 30]. As a result, power efficiency is one of the most important metric for achieving high performance in future CPUs. As such, modern processors are getting more efficient as power efficiency becomes one of the most important design goals. This trend is playing out in industry. For example, Figure 2.1 shows power consumption of server processors from Intel. The power consumption of server processors is generally decreasing. In particular, the Intel *Gainstown* processor consumes as little as 7.5W per core.

A fixed power envelope and continued technology scaling are the main factors which drive efforts in power efficient designs. We want to fully utilize transistors, which are more available than ever before, but with power constraints in mind. For this reason, we need more power efficient architectures. We cannot continue using performance-oriented techniques which may exhibit lower power efficiency, such as higher clock rates, deeper pipelining, wider instruction windows, etc. In other words, clock rate scaling and ILP exploitation are not solutions to achieve high performance anymore. Therefore, it is extremely important to continue efforts to squeeze inefficient power consumption out of architectures. For this reason, modern

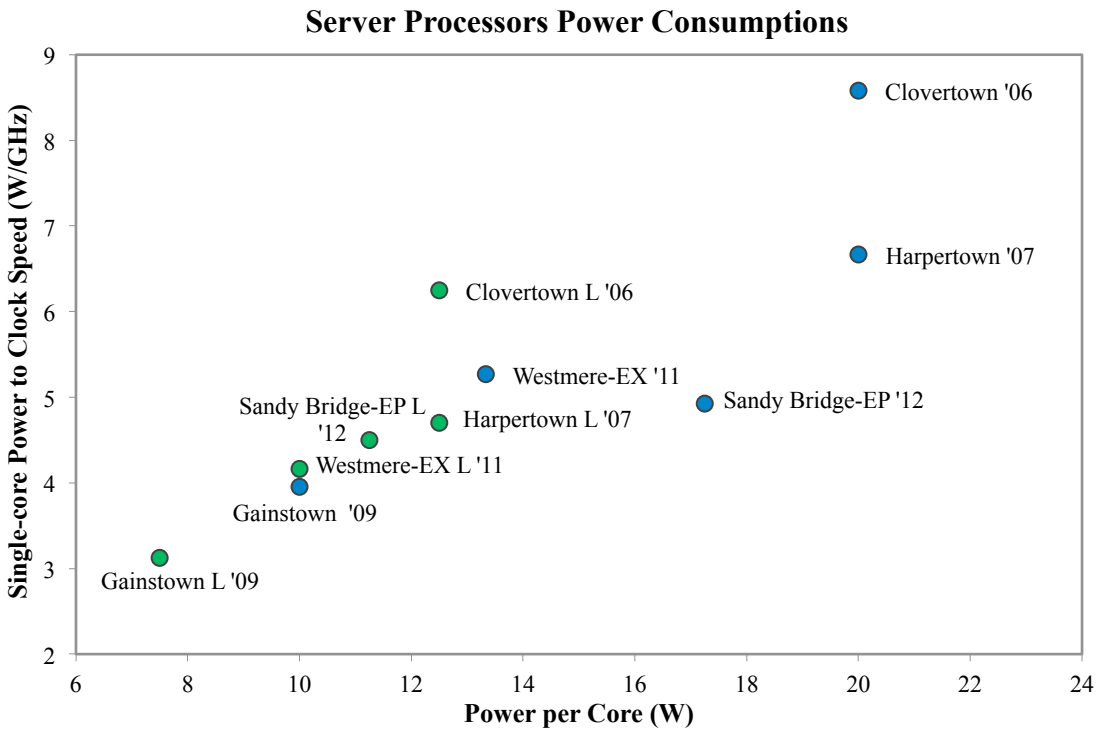


Figure 2.1: Power consumption per core and per clock speed of modern commodity processors. [1]

processors commonly adopt simple core designs that have narrow instruction width and shallow pipeline depth to achieve higher power efficiency.

Our goal in this thesis is to save power consumption in multi-level caches, which can consume as much as half of the overall processor's power. We conduct our study with a power efficient processor, comparable to state-of-art commodity processors. To compare with commodity processors, as can be seen in Table 2.1, our target processor consumes 5W per core on average and runs at a clock rate of 2GHz. We will cover the details of the processor in Section 4.3.2.

[1] All power consumptions are based on Intel's TDP (Thermal Design Power) reports. 'L' stands for a low-voltage model. Specific model names are L5335 and E5345 for *Clovertown*, L5430 and E5472 for *Harpertown*, L5530 and E5540 for *Gainstown*, L5645 and E5649 for $Westmere - EX$ and E3-1265L and E3-1270 for $SandyBridge - EP$.

CPU	Process	# of Cores	Clock Rate (GHz)	TDP (W)
Clovertown L	65nm	4	2	50
Clovertown	65nm	4	2.33	80
Harpertown L	45nm	4	2.66	50
Harpertown	45nm	4	3	80
Gainstown L	45nm	8	2.4	60
Gainstown	45nm	8	2.53	80
Westmerer-EX L	32nm	6	2.4	60
Westmere-EX	32nm	6	2.53	80
Sandy Bridge-EP L	32nm	4	2.5	45
Sandy Bridge-EP	32nm	4	3.5	69

Table 2.1: Server processors specifications.

	Intel Atom	Intel Sandy Bridge	AMD Opteron
L1 Capacity	24KB	32KB	64KB
Latency	3	4	3
L2 Capacity	512KB	256KB	512KB
Latency	15	12	12
L3 Capacity	N/A	0.5 - 2M	0.5 - 2M
Latency		26 - 31	34 -

Table 2.2: Cache hierarchy of modern commodity processor.

2.2 Multi-level Caches

The trend for modern CPUs is towards deeper cache hierarchies to bridge the processor-memory speed gap. Modern multi-level caches mostly consist of three levels of cache with each caching level exhibiting different access times and capacities. Table 2.2 shows cache hierarchies of some modern commodity processors. At lower levels within the cache hierarchy, cache capacities are growing to exploit larger working sets at the expense of higher cache latency.

2.3 Cache Resizing

Cache resizing, an architectural technique to save caches' power consumption while maintaining near-peak performance *i.e.* intelligently turning off/disabling ways and/or sets of under-utilized cache to save power, has been known for several decades. But its application for a multi-level cache hierarchy within a multicore

CPU has not been fully investigated. Although there has been significant work on cache resizing, existing techniques are limited to single-level cache resizing. In particular, most studies consider resizing a single level of cache for a uniprocessor only [2, 5, 6, 7, 8, 9], (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_t devices [31, 32], adaptive body bias (ABB) [18, 33], and dynamic voltage scaling (DVS) [34, 35] 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–for example, at the L2 given an L1 with a high miss rate–limiting their application.

2.3.1 Multi-Level Cache Resizing

Deeper cache hierarchies distribute the power consumption across different caching levels. For dynamic power consumption, the L1 cache is the greatest culprit, but for static power consumption, the LLC is by far the greatest concern due to its large area. Figure 2.2 shows the power consumption breakdowns for the SPEC 2006 benchmarks in a 2-way out-of-order core with a three-level cache hierarchy (details will follow in Section 4.3.2). The power consumption of the cache hierarchy accounts for up to 38% (31% on average) of the total system power consumption. Figure 2.3 shows the cache power consumption breakdown. As Figure 2.3 shows, the dynamic power of the L1 cache alone can take up to 47% with average of 26%, and the static power of the LLC can take up to 37% with average of 28%, of the total power consumption of the three-level cache hierarchy. As such, investigating multi-level resizing is mandatory to eliminate all wasteful power consumption in the cache hierarchy.

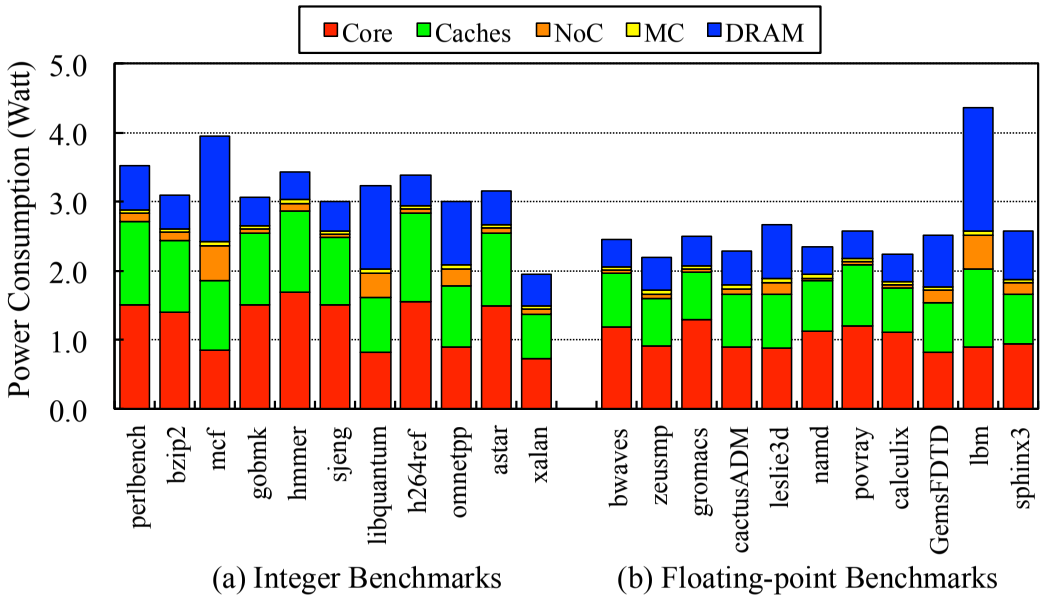


Figure 2.2: System power consumption breakdown of the workloads.

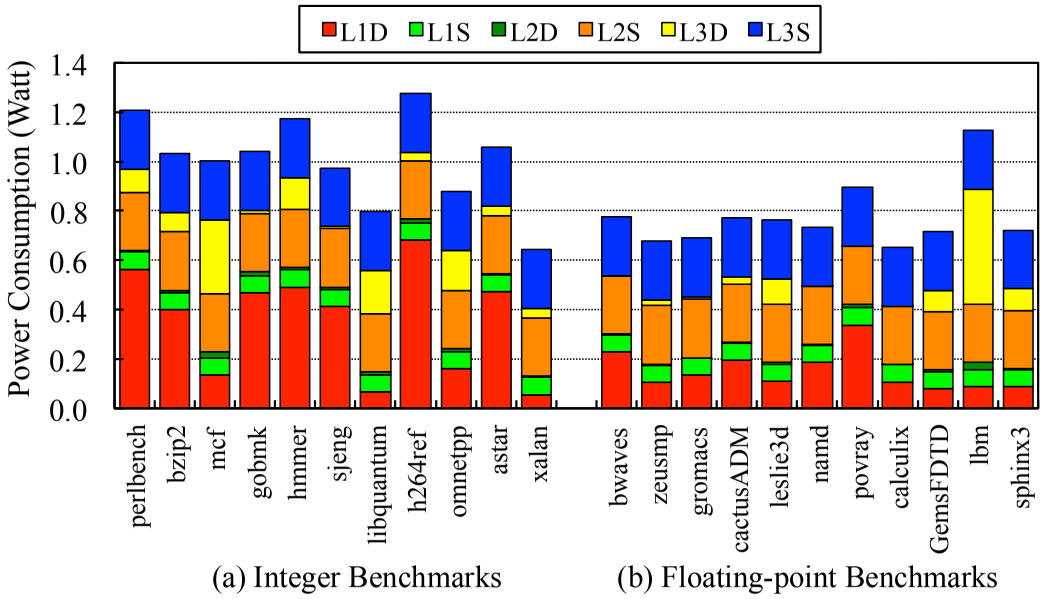


Figure 2.3: Cache power consumption breakdown of the workloads.

2.3.2 Optimal Multi-level Caches

An optimal multi-level cache hierarchy saves power in two ways. First, it eliminates wastful power consumption caused by over-provisioning of each cache. Modern multi-level caches are designed to perform well under worst case conditions. But not all programs have large working sets that require the worst-case amount of cache. Thus, by eliminating unnecessary portions of caches, significant power savings can occur. Second, optimal multi-level hierarchies achieve balance between different adjacent caching levels. A smaller L1 cache can reduce total L1 access energy, but will increase access energy to the L2 energy due to increased L2 traffic. There are three balance points, L1/L2, L2/L3 and L3/Memory, and these are all tied together. Unbalanced caches waste energy. By balancing/changing traffic between caching levels, one can reduce power consumption. Unlike previous single-level cache resizing studies, this is not only a much more challenging problem for the large solution space, but also a more interesting problem to understand the source of power savings, right cache provisioning and traffic controls between caches.

Finding the optimal cache hierarchy configuration requires considering the interactions between resizing decisions across different caching levels. Multi-level cache resizing balances the power consumed by a cache against the power consumption it inflicts on the next level of cache through its cache misses. Notice, a cache's balance point depends on both the upstream and downstream caches (if any), which in multi-level cache resizing are themselves resizable. Thus, the optimal configuration is the one that achieves balance *globally* across all the caches at the same time.

Optimal cache hierarchy configuration not only eliminates wasteful power consumption from the cache hierarchy, but also maintains high performance as well, achieving higher power efficiency. We conduct a limit study to find *Pareto-frontiers* in the performance-power domain. Based on our extensive simulations, which will be discussed in Section 4.4, we search exhaustively over the entire solution space to generate a set of *Pareto-frontiers*. Figure 2.4 shows the power consumption and the performance level per average performance degradation using SPEC CPU 2006 benchmarks. We find that optimal cache configurations outperform both in the power savings and the performance locally compared to other cache hierarchy configurations. For example, within 5% performance degradation, the optimal cache hierarchy configuration can save as much as 20% of the total system power with 3.3% of performance degradation. Moreover, within 10% performance degradation, the optimal cache hierarchy configuration saves power consumption by as much as 20.6% while maintains the performance degradation level of 5.4%.

2.3.3 Dynamic Reconfiguration

Optimal cache hierarchy configurations vary across different benchmarks. Figure 2.5 shows different optimal cache hierarchy configurations of SPEC CPU 2006 benchmarks (due to the difficulties in plotting 3-dimensional result of a 3-level cache hierarchy, we conduct a study for a 2-level cache hierarchy). We find optimal cache hierarchy configurations that consume least power while maintain performance degradation level less than 10% compare to the baseline configuration (baseline cache

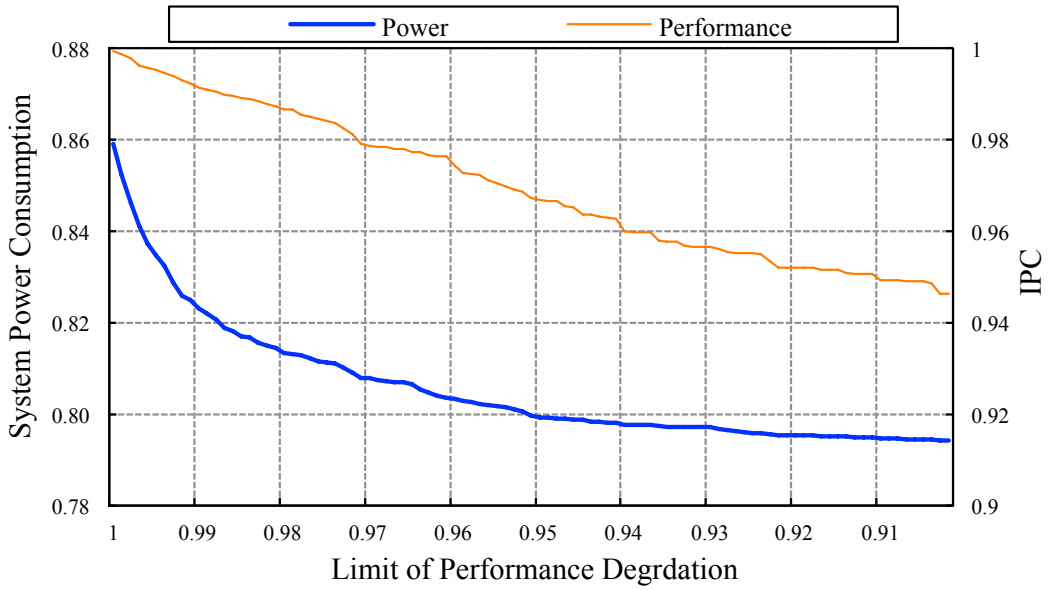


Figure 2.4: Lowest Power Consumption with Limited Performance Degradation.

hierarchy configuration consists of 8-way L1 cache and 8-way L2 cache). In particular, Figure 2.5-(a) shows optimal cache configurations per benchmark and their relative power consumption in the cache hierarchy colored differently. For example, the optimal cache hierarchy configuration of *astar* is 2-way L1 cache and 16-way L2 cache and this optimal cache hierarchy configuration consumes 70-80% of the power consumption of the baseline cache hierarchy configuration. As shown in Figure 2.5-(b),(c), and (d), optimal cache hierarchy configurations of *omnetpp*, *astar*, and *namd* are 4-way L1 cache and 10-way L2 cache, 2-way L1 cache and 16-way L2 cache, and 8-way L1 cache and 2-way L2 cache, respectively. As a result, to eliminate wasteful power consumption from a cache hierarchy, we need a novel architecture that can adapt to diverse application behaviors.

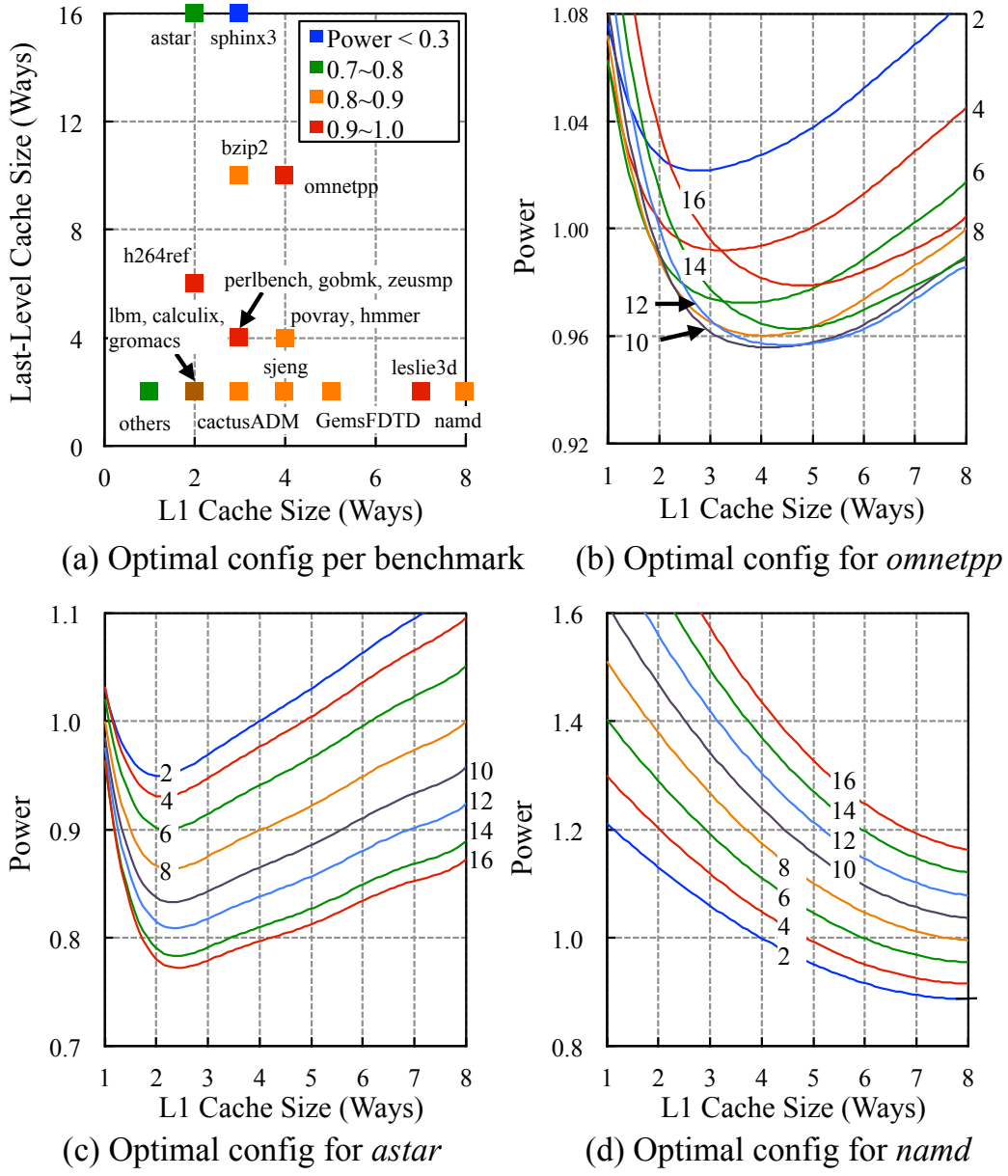


Figure 2.5: Different optimal cache hierarchy configurations across SPEC CPU 2006 benchmarks. Due to the difficulties in plotting 3-dimensional result, we plot result for a 2-level cache hierarchy. Each trend line represents the power consumption of a corresponding L2 cache configuration.

2.4 Power Efficient Cache Partitioning

Cache-partitioning techniques have been widely investigated for higher performance [11, 12, 13, 14, 15, 16, 17, 36, 19, 20, 21]. However, cache partitioning for better power efficiency has not been extensively studied yet. Cooperative Partitioning [10], or *LLC Resizing*, is a previous technique focused on resizing a CMP's shared LLC to save power with a given threshold. It disables ways of the LLC when the *utility* of the way is not high enough. Although it proposes a way to save power consumption in the LLC by disabling ways with low utility, it is derived from the performance optimization technique in [11], and hence, it does not capitalize on all of the potential power savings from LLC partitioning.

Most state-of-the-art LLC partitioning techniques dynamically change partition sizes. As such, resizing private caches above the LLC in CMPs will encounter a dynamically changing LLC partition size. Although LLCs commonly use the serialized access technique, which saves the data-array access power when a cache miss occurs, its tag access power can be significant compared to the potential power savings from private cache resizing. The dynamic power consumption of tag accesses is dependent on the LLC partitioning, the allocated LLC ways and thus, private cache resizing increases the power consumption of the LLC. To alleviate this problem, private cache resizing should be performed with the awareness of the increased dynamic power at the LLC constrained by a threshold to avoid severe performance degradation.

2.5 Multi-Level Cache Resizing in CMPs

A CMP's cache hierarchy is crucial in determining its performance and power consumption. Already, a large body of research on LLC resource distribution or LLC partitioning has been conducted for this reason. However, previous studies did not provide a solution for multi-level cache resizing in CMPs, which resizes the CPU's private caches along with LLC partitioning. Unfortunately, there is no comprehensive study on cache resizing for a modern CMP yet.

As discussed earlier, power consumption in multi-level cache hierarchies is distributed across the different levels. Each level of cache exhibits its own power savings and performance degradation when the waste is squeezed out. Since the overall performance impact and power savings are not trivial, when multiple cache-resizing techniques are combined, it is crucial to predict potential power savings and performance loss to control each caching level to save most of the wasteful power while maintaining high performance.

One possible way to achieve this goal is by applying uni-processor MCR to each core in the CMP assuming an even partitioning of the LLC across cores. Each workload would achieve near-peak performance with bounded degradation. But this simplistic approach would be suboptimal because each LLC partition is limited to the evenly distributed capacity. On the other hand, it is possible to consider all combinations of possible cache configurations in the CMP to find the globally optimal configuration given a bounded performance degradation compared to the achievable maximum performance. However, finding such a solution is infeasible in

a real system due to the huge search space. For this reason, we need a novel way to approximate the global optimum with low complexity.

Chapter 3

Related Work

This chapter summarizes and discusses related studies of power efficient caches.

3.1 Cache Resizing

As we discussed earlier, cache over-provisioning causes significant waste in the power consumption of a cache hierarchy. Moreover, this waste harms the total power efficiency of the system due to the large portion of the cache hierarchy's power consumption. However, such waste can be eliminated by reconfiguring a cache to the minimum size that the cache is not incapacitated.

A large body of work exists on cache resizing and most of the work solves the problem by addressing three fundamental aspects: how to resize a cache , how to decide its size, and which power to save. First, most of the work is based on the idea that caches consist of subarrays. Because most of caches already utilize partitioned organization for performance reasons, minor changes to a conventional cache organization can result in a reconfigurable cache. For example, number of sets or ways, or block sizes can be reconfigured to meet a certain size at different granularities. Second, deciding ideal cache size is another aspect to eliminate the waste. There are static and dynamic approaches. Static approach utilizes off-line profiling information to capture ideal cache sizes. On the other hands, dynamic

approach employs heuristic- or approximation-based searching/learning/prediction methods. Third, cache consumes dynamic and static power. Either one or both can be an objective function for the optimization. We will compare related studies with these perspectives.

Selective cache ways [2] uses off-line profiling to drive disabling of cache ways for dynamic power savings. DRI caches [7, 9] use cache-miss counts to detect over-provisioning, and resize across cache sets. In addition, DRI caches also gate the power supply to unused portions of cache, conserving both dynamic and static power. A hybrid approach, selective sets and ways, was proposed [8] to provide better granularity compared to previous studies. We use selective sets and ways cache organization for L1 caches for its finer granularity and use selective ways cache organization for L2 and L3 to reduce flushing overhead caused by changing sets. Malik *et al* [6] study selective ways in the MCore CPU. Madan *et al* [5] propose resizing L2 caches by dynamically extending their capacity into stacked DRAM. In this study, 3D-stacked SRAM and DRAM dies provide heterogeneous configurations to the cache hierarchy and its impacts were studied. However, interaction between caching levels or controlling sizes of different caching levels are not discussed. All of these prior studies consider resizing a single level of cache only, whereas we address the problem of resizing multiple levels of caches in CMPs. In particular, we develop novel algorithms for solving an *NP-hard* problem in $O(N^2)$ time complexity with a *greedy* approach.

Besides resizing, researchers have studied other adaptive cache techniques as well. Dropsho *et al* [37] propose *accounting caches* which divide a cache's ways

into primary and secondary groups. Each cache access searches the two groups sequentially, accessing the secondary only on a primary miss. This saves power if secondary accesses are infrequent. Zhang *et al* [38] propose *way concatenation* which permits flexible organization of cache banks to form direct-mapped, 2-way, or 4-way set-associative caches. Neither accounting caches nor way concatenation address capacity allocation across different levels of cache, the main focus of our study.

3.2 Cache Partitioning

The limits of exploiting ILP and frequency scaling lead to CMP scaling. The growth in the number of cores in CMPs make shared-resource distribution critical to achieve both high performance and low power consumption. In most CMPs the LLC is shared between threads and its utilization has significant impacts on both performance and power because LLC misses cause off-chip traffics which result in high latency and power consumption. Cache partitioning explicitly allocates shared cache across multiprogrammed workloads, providing cache to those programs that can best utilize it.

The majority of techniques focus on performance [19, 36, 20, 11, 21, 17, 13]. Suh *et al* propose an online monitoring scheme to partition the cache to maximize the overall performance, and implements marginal-gain counters to predict the overall miss rate and improves partitioning scheme based on the miss-rate information [19, 20]. Fairness in cache sharing was studied in [36]. This study implements static and dynamic LLC partitioning algorithms to improve fairness between threads. In

addition, the relationship between fairness and throughput was shown that fairness usually increases throughput, but not vice versa. Qureshi *et al* proposes utitliy-based cache partitioning [11]. This study captures best partitioning at run time by utilizing way-counters based monitoring. The lookahead algorithm considers the marginal utility for all possible ways which can be distributed to each thread. Time-sharing cache partitioning scheme allows each thrashing thread to occupy large portion of the shared cache capacity to achieve significant speedup in turn [17]. In this study, Chang *et al* shows that time-sharing based cache partitioning not only results in better performance, but better fairness whiling maintaining QoS. Liu *et al* [13] conducts off-line study to analyze best cache allocations and develops a run-time cache partitioning technique to victimize aggressor threads to achieve high performance.

More recently, techniques have also tried to reduce power consumption [39, 10] by withholding allocation and shutting down portions of the shared cache, similar to cache resizing. Like our study, cache partitioning also employs reuse distance profiles to drive allocation decisions. But LLC partitioning saves mostly static power consumption compared to our study which also resizes private caches where dynamic power dominates. ReCac, Reconfigurable Cache for CMPs, was proposed [39] not only for performance, but for power savings. ReCac dynamically falls back to performance centric cache partitioning if the power savings are not desirable. Cooperative partitioning utilizes a way-aligned LLC for cache partitioning to save power while maintaining high performance [10]. This study uses utility-based metric and extends the lookahead algorithm [11] to decide partitions. In addition, migra-

tion overhead, caused by cache flush, is reduced by allowing multi threads to read the transitional ways simultaneously, but only a single thread to write, during the transitional period after a partitioning decision made.

We use similar approach to [39, 10] to resize a LLC in our study. In addition, our online monitoring scheme, including PEG monitoring which will be discussed in Chapter 5, is inspired a way counter [19] that approximates stack distance as most of the work [36, 20, 11, 10] utilize it either for dynamic or static partitioning scheme. Although our work and existing techniques are similar in that both approaches are "horizontal" allocation technique, out work is distinguished from previous work by out approach is "vertical" as well. While both can save power, cache partitioning does so by optimizing utility across competing threads whereas we do so by optimizing balance between caching levels and across threads. For this reason, our scheme outperforms previous techniques in both performance and power savings.

3.3 Multi-level Cache Optimization

The trend for modern CPUs is towards deeper cache hierarchies to hide memory latency. As such, multi-level cache hierarchies distribute power consumption across many cache levels. Although there has been significant work on cache resizing and partitioning as we discussed in Section 3.1 and 3.2, most of existing techniques are limited in their optimization scope, *i.e.* single caching level. Unfortunately, there was no comprehensive "vertical" study yet.

Balasubramonian *et al* [3, 4] propose resizing 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 partitionings always utilize all of the available SRAM, only one cache's size is controlled independently. Hence, in this technique, it is impossible to optimize the balance point of different caching levels simultaneously as is done in our study. Moreover, the technique is only limited to uniprocessors. Wang *et al* [40] propose private cache resizing in conjunction with LLC partitioning. This technique requires off-line profiling to generate the profile tables for each task including information of energy consumptions, L1/L2 partitions, and execution times. Although the proposed algorithm considers L1 cache resizing impacts on L2 cache partitions, its computation overhead is significant, $O(MN^2)$ with M cores and N configurations per caching level. Besides, its off-line profiling requirement makes the algorithms not feasible in real world in that off-line profiling is neither always available nor cheap.

3.4 Circuit-level and Design-time Optimization

Finally, significant number of researches has explored circuit-level techniques for reducing a cache's static power consumption. Both feature-size scaling and threshold-voltage scaling result in high leakage energy dissipation, *i.e.* high static power consumption. Gated-V_{DD} [7, 9] employ an extra transistor in V_{DD} of the SRAM cells in caches to gate the power supply. The extra transistor is turned off in the unused portions of the cache and the stacking effect of reverse-biased series-connected transistors reduces the leakage energy in the unused portions. This

technique also adapts DRI, an architectural scheme to reduce power consumption of cache as we discussed in Section 3.1, hence achieves leakage power reduction with an integrated circuit and architecture approach.

As process technologies continue to scaling down feature sizes, below 100 nm, subthreshold leakage power has become a dominant fraction of static power dissipation. Since scaling down voltage increases the leakage power exponentially and improves switching frequencies, there is a trade-off between access time and leakage power. For this reason, there is a large body of work exist to explore the trade-off and to reduce the leakage power while maintaining the performance. Multi-V_t techniques [32, 31] employ low-V_t devices along critical paths and high-V_t devices along non-critical paths to save power while still maintaining performance. Similarly, drowsy cache studies [34, 35] employ dynamic voltage scaling to reduce the leakage power of inactive cache lines by putting the cache lines into a low-power standby mode. Consequently, drowsy caches are able to reduce the leakage power because large portion of the cache lines can be put in a standby mode without significant performance degradation. The state transition can be switched between standby and active modes by scaling the supply voltage.

Unlike these static or design-time V_t assigning, there is a dynamic approach to control V_t as well. Most of studies [33, 18, 41, 42, 43, 44] adopt body biasing technique to change V_t. FBB controls the back-gate 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. RBB works in the opposite way, it lowers V_t in active mode for fast access while suppressing leakage current in standby

mode with high V_t. Nii *et al* propose auto-backgate-controlled MTCMOS [45, 46] which increases V_t in sleep mode by backdate biasing in order to reduce undesirable leakage current during the inactivated time while retaining the data stored in the SRAM [33]. Kim *et al* further improves the scope of leakage power reduction by proposing active leakage reduction [18]. This technique, inspired by DVS [41, 42], scales V_t by employing body bias control. Like DVS, this technique utilizes a slack time in computation to increase V_t, hence reduces active leakage power.

Similar to these, Tschanz *et al* evaluates sleep transistor and body bias to reduce active leakage power [43]. This study finds that both MTCMOS sleep transistor and FBB, which lowers V_t in use while maintaining low-leakage current by applying ZBB or RBB in idle mode, can be used dynamically for active leakage control. Kim *et al* reduces active and standby leakage power in cache memories [44]. This study employs super high V_t devices to reduce standby leakage current and dynamically applies FBB only for active SRAM cells in active mode.

Jacob *et al* derives a closed-form solution as a first-order approximation to provide optimal sizes of each caching level [47]. This lacks of power model and does not capture dynamic phase changes for its static model. Silva-Filho *et al* [48] and Gordon-Ross *et al* [49] study design-time techniques for optimizing 2-level cache hierarchies. This body of work tries to find the best block size and associativity–as well as cache capacity–for two caching levels. They consider a more complex design space than we do, and employ more costly search techniques that are suitable for design analysis only. In contrast, our approach is an architecture-level power management technique. It solves a more constrained problem, but provides algo-

rithms suitable for runtime use. Similarly, Zhang and Vahid [50] search for the best cache architecture using a reconfigurable hardware platform. But they only consider optimizing a single level of cache.

Chapter 4

Multi-Level Cache Resizing Behavior Analysis

A deeper cache hierarchy bridges the memory-processor speed gap. Besides contributing power consumption significantly with each additional caching level, the multi-level cache hierarchy also determines the CPU's overall performance. Therefore, saving wasteful power in the cache hierarchy should be done without harming overall performance. In other words, we wish to identify the best configurations that eliminate virtually all wasteful power consumption while maintaining near-peak performance. In this thesis, we view this problem as a constrained multi-variable optimization problem.

This chapter conducts limit studies to understand the potential of multi-level cache resizing and the difficulties in performance-evaluation based search approaches to solve this constrained multi-variable optimization problem.

4.1 Complexity of Multi-Level Cache Resizing

As we discussed, there is significant potential in multi-level cache resizing. Unfortunately, multi-level cache resizing becomes an *NP-hard* problem as the number of cores scales out. Even in a uniprocessor, optimizing a three-level cache hierarchy is very challenging. If we have a k-way configurable cache per level, then we solve the problem in the space complexity of $O(k^l)$, where l is the number of caching

# cores	1	2	4	8
space complexity	$O(k^l)$	$O(k^{2l})$	$O(k^{4l})$	$O(k^{8l})$
space volume	256	$114,688$	3.1×10^{10}	3.0×10^{20}

Table 4.1: Multi-level cache resizing problem space scaling.

levels in the hierarchy. For example, we have 256 possible configurations for the three level cache hierarchy consists of 8-way configurable L1, 8-way configurable L2, and 4-way configurable L3. In CMP, problem space grows exponentially. Table 4.1 summaries the multi-level cache resizing problem scaling.

4.2 Power and Performance Impact of Varying Cache Sizes

We conduct exhaustive simulations over all possible combinations of cache configurations per level. In particular, we compare 2,048 cache hierarchy configurations to understand the power consumptions and the performance according to a cache hierarchy configuration. Our base line cache hierarchy configuration consists of 32KB L1 cache, 256KB L2 cache, and 2MB LLC. Performance and power consumption values are normalized to the performance and the power consumption of the baseline cache hierarchy configuration. Details of this simulation environment will be discussed in Section 4.4. As shown in Figure 4.1, the different cache configurations significantly affect the overall performance. For example, the optimal cache configuration achieves 1.65X performance compared to the baseline for the *sphinx3* workload. On the other hand, some cache configurations harm the performance.

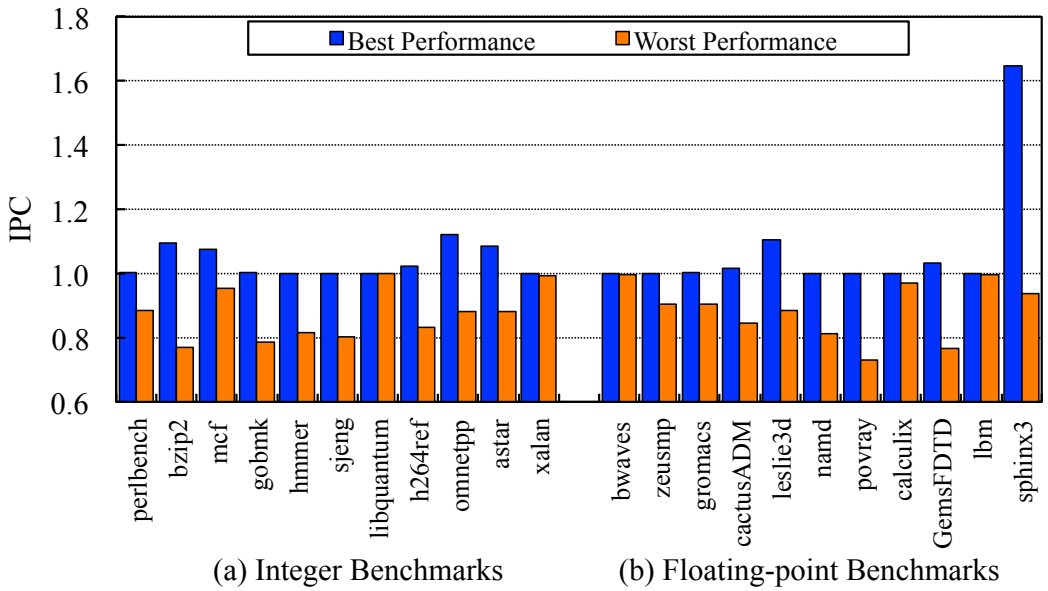


Figure 4.1: Best and worst performance of workloads by changing cache configurations. IPC is normalized to the IPC of the baseline configuration.

For example, the worst configuration achieves 0.73X of the baseline performance for the *povray* workload. On average, we can expect 18% performance improvement comparing the best configurations to the worst configurations.

On the other hand, cache configurations also affect the total system power consumption. As shown in Figure 4.2, an optimal cache configuration reduces the total system power consumption by up to 28% of the power consumption of the baseline. Similar to the performance comparisons, some cache configurations increase the total system power consumption. For example, the worst cache configuration exhibits 1.85X of the power consumption of the baseline configuration for the *hmmer* workload. On average, the best configurations can save 50% of the total system power of the worst configurations.

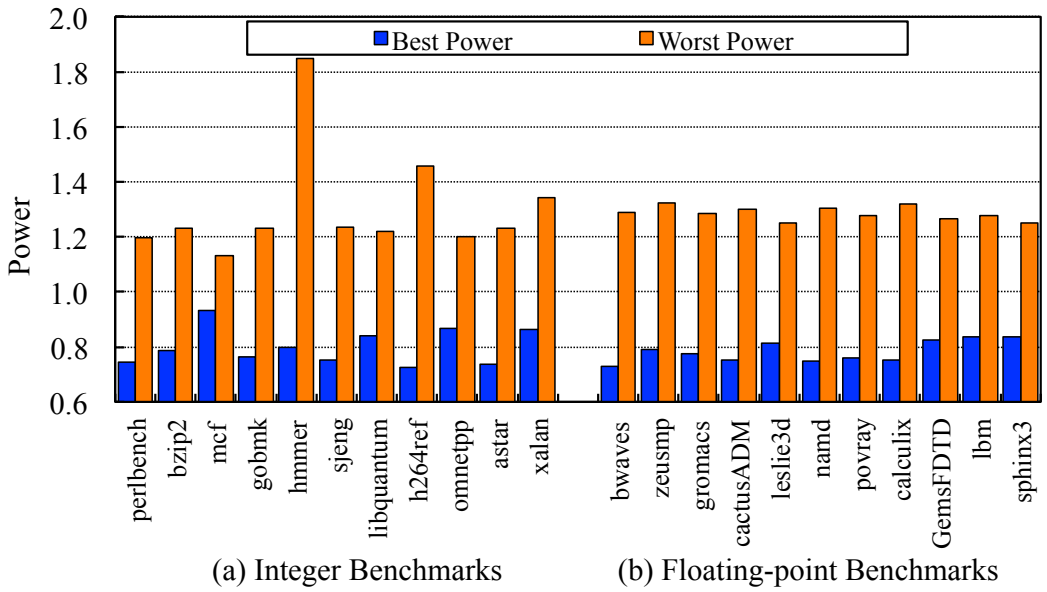


Figure 4.2: Best and worst power consumption of workloads by changing cache configurations. Power is normalized to the power consumption of the baseline configuration.

4.3 Experimental Methodology

In this section, we detail our experimental methodology. Figure 4.3 shows our framework for this study. We generate our workloads using SPEC CPU 2006, and conduct performance and power simulations.

4.3.1 Single-program Workload Generation

We use 22 SPEC CPU2006 benchmarks (11 integer and 11 floating point), as shown in Table 4.2. Because we use a SimpleScalar ported to Alpha/Linux, we compile the SPEC CPU 2006 benchmarks on a native Linux environment. We installed an Alpha CPU emulator [51] on a Windows PC and then installed the *Linux*

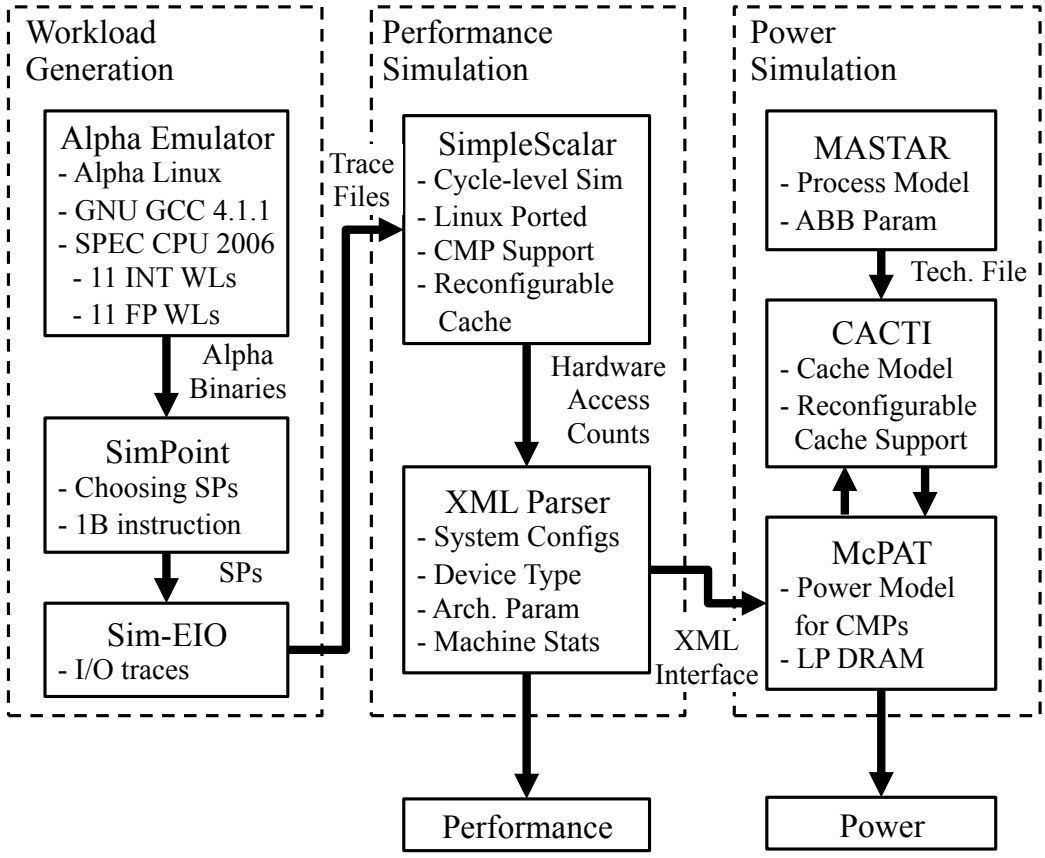


Figure 4.3: Simulation framework. We use AlphaVM, SimPoint, and Sim-EIO to generate our workloads. We use SimpleScalar for performance simulation. Lastly, we use MASTAR, CACTI, and McPAT to simulate power consumption of the target system.

(*Debian Lenny*) system on the emulator. We compile the benchmarks natively using an Alpha compiler, gcc-4.1.1 (provided along with *Debian*). We compile the benchmarks with the -O2 option and link glibc-2.5 statically. We were not able to compile 447.dealII. Moreover, one integer benchmark (403.gcc) and five floating point benchmarks (416.gamess, 433.milc, 450.soplex, 465.tonto, and 481.wrf) either could not be finished or did not match to the reference outputs. For these reasons, they have been omitted from our study. Using the reference inputs, all of the compiled benchmarks were run to completion on SimPoint [52]. We take the most representative *simpoint*, consisting of 1B instructions per benchmark. Each simulation point contains 1.1B instructions. In our experiments, we simulate the first 100M instructions to warmup the cache, and then we simulate the next 800M cycles after cache warmup to acquire detailed statistics. Because benchmarks' IPCs differ, we simulate a fixed cycle count instead of a fixed instruction count to ensure all benchmarks within each multiprogrammed workload are active simultaneously.

4.3.2 Performance Simulation

We use a modified Simplescalar simulator for the Alpha ISA [53] to conduct our study. Table 4.3 shows our baseline processor configuration. As mentioned in Chapter 2, we model state-of-the-art power-efficient cores. As such, we use a relatively narrow 2-way issue core to achieve high power efficiency. The cores are attached to a three-level cache hierarchy. In particular, the on-chip cache hierarchy has a split 8-way 32KB L1 private cache, a unified 256KB L2 private cache, and

Group	Benchmark	MPKI	Group	Benchmark	MPKI
(h_4)	Mcf	51			
	Libquantum	29	(l_9)	Astar	0.82
High	Lbm	22		Perlbench	0.69
	Omnetpp	15		Hmmer	0.66
(h_0)	GemsFDTD	14		H264ref	0.54
(m_6)	Leslie3d	9.5	Low	Sjeng	0.27
	Sphinx3	8.2		Gobmk	0.2
	Xalan	6.8		Calculix	0.2
Medium	Bwaves	4.8		Gromacs	0.13
	Zeusmp	4.1		Namd	0.07
	CactusADM	2.3	(l_0)	Povray	0.03
(m_0)	Bzip2	1.9			

Table 4.2: Benchmarks classification based on misses per kilo instructions (MPKI) of 2MB LLC.

a shared LLC. The LLC is 2MB for a single core, 4MB for two cores, 8MB for 4 cores, and 16MB for 8 cores. Its associativity increases by 4 ways for each additional core. The cache block size is 64 bytes for all caches. The baseline cache hierarchy maintains the noninclusive inclusion property for the L2 cache and the LLC. We model DDR-3 memory of 32GB with a 140ns access latency [54]. We assume a low power version of DRAM to align with our power efficient core model.

We extend our performance simulator to support reconfigurable caches and to generate hardware access counts for a power simulation. As shown in Figure 4.3, we define system configurations including number of cores, cache hierarchy and memory/ memory controller topology. Moreover, our XML parser also includes device type, architectural parameters, and machine statistics in the XML file that will be fed into our power simulator.

4.3.3 Power Simulation

We use McPAT [55] and CACTI 6.5 [56] for power modeling. Our baseline model uses the 32*nm* technology node and ITRS *high performance devices*.

Multi-level Caches We adopt a state-of-art circuit- and device-level static power reduction technique to model the static power of the shared LLC more realistically. Specifically, we assume high-V_t devices throughout [44], but apply reverse body bias (RBB) in standby mode to further reduce standby leakage [43]. When an access occurs, we apply a forward body bias (FBB) to restore the threshold voltage for low access delay. We assume that applying FBB does not impact the access

Cores	2.0 GHz 2-way out-of-order 64-entry ROB, 24-entry LSQ Gshare/bimodal hybrid branch predictor 2048-entry meta table 512-entry BTB
L1 I-Cache	32 KB, 2-way, 64-byte blocks, 1 cycle
L1 D-Cache	32 KB, 2-ports, 8-way, 64-byte blocks, 4 cycles
L2 Unified Cache	256 KB, 8-way, 64-byte blocks, 7 cycles
L3 Shared Cache	up to 16 MB, 32-way, 64-byte blocks, 19 cycles Noninclusive, Bus-type interconnect
Memory	32 GB Low Voltage Quad-rank RDIMM DDR3-800, 140ns loaded latency

Table 4.3: Architectural configuration.

delay for the cache [43]. We utilize stack effect in conjunction with ABB to model way selection [57, 43]. We use the Model for Assessment of cmoS Technologies And Roadmaps (MASTAR 2011) from ITRS [58] to derive parameters required for CACTI according to our assumptions.

We model up to 8 out-of-order cores attached to thee-level cache hierarchies consisting of private L1 and L2 caches, and a shared L3 cache. The baseline cache hierarchy maintains noninclusive inclusion properties for L2 and L3 caches. Before resizing caches, we apply existing techniques to ensure the baseline cache hierarchy is reasonably efficient. In particular, we assume the L3 cache serializes tag and data accesses such that only a single data way is ever accessed regardless of the number of configured cache ways. Due to greater latency sensitivity, we perform

parallel tags and data access in the L2 cache, though we serialize broadcasting the accessed data block in data array h-tree. Figure 2.3 from Chapter 2 shows this cache-hierarchy power breakdown of the baseline multi-level caches for each SPEC CPU2006 benchmark.

4.4 Limits of Multi-Level Cache Resizing

The goal of offline analysis is to provide the global view of the performance and power curves. To achieve this goal, we search the whole solution space of the performance and power functions for all (x, y, z) in Eq. 5.7 by simulating all data points in the solution space.

Exhaustive Search We conduct an off-line exhaustive search to study the limits on performance improvement and power savings that our technique can potentially provide. This provides a reference by which to compare other schemes.

To facilitate the study, we run all possible combinations of configurable caches. The extensive simulations enable the exhaustive search over the entire solution space to find the best static solution. Table 4.5 shows the cache configurations for this study. Each workload is simulated 3,072 times to search the entire solution space, and 67,584 simulations are conducted to run all of the workloads.

Benefits of Multi-Level Cache Resizing with Performance Constraint Multi-level cache resizing is crucial to eliminate wasteful power consumption in a cache hierarchy because deeper cache hierarchies distribute power consumption

Table 4.4: Cache parameters for the baseline multi-level caches.

L1 Cache	
Access (tag array / data array / data array h-tree)	Parallel / Parallel / Parallel
Read energy per access	0.396 nJ
Write energy per access	0.483 nJ
Subthreshold leakage	41.5 mW
Gate leakage	22.3 mW
L2 Cache	
Access (tag array / data array / data array h-tree)	Parallel / Parallel / Serial
Read energy per access	0.258 nJ
Write energy per access	0.279 nJ
Subthreshold leakage	154.5 mW
Gate leakage	78.5 mW
L3 Cache	
Access (tag array / data array / data array h-tree)	Serial / Serial / Serial
Read energy per access	4.38 nJ
Write energy per access	6.6 nJ
(Standby) Subthreshold leakage	237.5 mW
(Standby) Gate leakage	1666 mW

Config #	1	2	3	4	5	6	7	8	9	10	11	12	...	32
L1	16:2	4	6	8	32:5	6	7	8	64:5	6	7	8	...	N/A
L2	512:1	2	3	4	5	6	7	8	N/A	N/A	N/A	N/A	...	N/A
LLC	8192:1	2	3	4	5	6	7	8	9	10	11	12	...	32

Table 4.5: Reconfigurable cache configurations (set:ways). For example, #7 configuration of L2 is 512:7 (set number is omitted if it is same to the set number of a previous configuration). In total, 67,584 simulations are conducted.

across more levels. As we pointed out earlier, we wish to consider *Pareto*-optimal cache hierarchy configuration because a cache hierarchy configuration determines the overall system performance as well. To do so, we enforce a performance degradation level within 1% compared to the performance level of the baseline cache hierarchy configuration. Note that such *Pareto*-optimal cache hierarchy configurations may increase the total energy consumption in some cases, but we focus on reducing the total system power consumption and our approach generally reduces energy as well due to the negligible performance degradation while reducing power consumption significantly. Figure 4.4 summarizes the limits of power savings from different approaches; comparing power savings from L1-only, L2-only, L3-only and multi-level exhaustive searches. These results demonstrate multi-level cache resizing can provide significant power savings compared to single level cache resizing at the same performance degradation level. Multi-level cache resizing can save total system power by as much as 27%, and 15.4% on average, while L1-only, L2-only, and L3-only resizing provide, on average, 8.6%, 5.1%, and 3.3% respectively.

Workload	L1	L2	L3	Power	Workload	L1	L2	L3	Power
perlbench	2	2	3	0.785	bwaves	1	1	1	0.730
bzip2	1	2	4	0.826	zeusmp	4	4	4	0.888
mcf	2	3	3	0.935	gromacs	3	2	4	0.866
gobmk	4	6	3	0.835	cactusADM	12	6	1	0.910
hmmer	4	3	3	0.828	leslie3d	8	3	2	0.871
sjeng	7	4	1	0.814	namd	11	4	1	0.861
libquantum	1	1	1	0.840	povray	8	8	1	0.855
h264ref	1	4	4	0.766	calculix	2	1	1	0.760
omnetpp	2	2	4	0.894	GemsFDTD	8	1	2	0.860
astar	2	3	4	0.811	lbm	1	5	3	0.926
xalan	1	2	2	0.865	sphinx3	2	1	4	0.874
INT AVG	2.5	2.9	2.9	0.836	FP AVG	5.5	3.3	2.2	0.855
WL AVG	4.0	3.1	2.5	0.846					

Table 4.6: Statically optimal multi-level configurations.

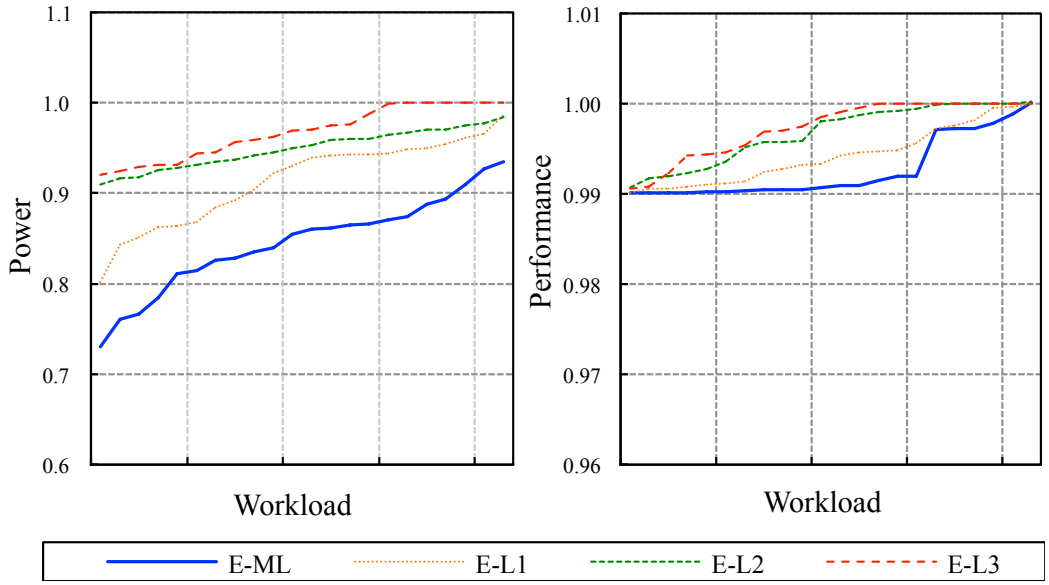


Figure 4.4: Power and performance comparison between exhaustive and per-level exhaustive searches. (For example, E-L1 only searches by resizing L1 cache capacity.)

Figures 4.5, 4.6, 4.7, and 4.8 show power consumption breakdown in the cache hierarchy per technique. Generally, workloads with higher MPKI consume more power in the cache hierarchy. In the first group with low MPKI workloads, the workloads consume 0.8W in the cache hierarchy given the baseline configurations. In the second group, workloads with low, medium, and high MPKI consume 1.2W, 0.9W, and 1.7W, respectively.

L1 cache resizing saves significant power consumption for workloads that dissipate a large portion of their power in the L1 cache. For example, in *h264ref*, Figure 4.6, the dynamic power consumption at the L1 caching level alone exhibits 51.6% of the total power consumption in this workload. As such, L1 cache resizing can reduce the power consumption in the cache hierarchy by more than 50%.

Likewise, L2 cache resizing saves significant power consumption for workloads that dissipate a large portion of their power in the L2 cache. *namd* and *calculix* in Figure 4.5, and *cactusADM* and *xalan* in Figure 4.7 exhibit more than 20% power reduction in the cache hierarchy with L2 cache resizing. Unlike L1 cache resizing, static power consumption in the L2 caching level makes up a significant portion of the total power consumption in the cache hierarchy: *namd*, *calculix*, *cactusADM*, and *xalan* show 32%, 36%, 27%, and 34%, respectively, of their total power consumption in the cache hierarchy, at the L2 caching level. As a result, L2 cache resizing effectively reduces the total power consumption of the cache hierarchy by down sizing the L2 cache.

Similarly, L3 cache resizing is effective when a workload consumes significant power at the L3 caching level. For example, *namd* saves around 24% of the total power consumption by down sizing the L3 cache. In this case, the static power consumption in the L3 cache takes 32% of the total power consumption.

On the other hand, multi-level cache resizing achieves best power reduction at a similar performance level compared to other techniques by orchestrating resizing at all three caching levels. Note that multi-level cache resizing never down sizes a caching level smaller than the size chosen by the other techniques. For example, the L1 cache size of multi-level cache resizing is always equal to or larger than the L1 cache size under L1 resizing. In particular, the L1 cache size of multi-level cache resizing for *grimaces* is three times larger than the L1 cache size under L1 cache resizing in Figure 4.5 and in Table 4.6. In this case, it is more power efficient to stop downsizing the L1 capacity at three and to reduce the size of the L2 cache down to

two rather than downsizing the L1 capacity all the way to one.

Moreover, multi-level cache resizing introduces bigger power dissipation at certain caching levels compared to the power consumption in the baseline cache configuration. *xalan* in Figure 4.7 exhibits significant power dissipation at the L3 cache in its dynamic power, which is 25% of the total power consumption or 93mW, in multi-level cache resizing technique. The L3 cache, in the baseline configuration, consumes only 36mW and it is 5% of the total power consumption in the cache hierarchy. However, even with such increase in the power consumption of the L3 cache, multi-level cache resizing reduces the total power consumption as much as 46% for *xalan*.

The other important consideraten is performance. We cannot simply combine the cache resizing decisions from each caching level to achieve the best power reduction because cache resizing at each level in isolation introduces more traffic to the next caching level. Theses traffic increases taken together result in worse performance than the aggregate performance degradation. However, multi-level cache resizing achieves best power reduction while maintaining a high performance level.

Therefore, we conclude that we need multi-level cache resizing for two reasons. First, no single-level approach captures all of the wasteful power consumption because a multi-level cache hierarchy distributes power consumption across different caching levels. And second, multi-level cache resizing can balance multiple caching levels to achieve acceptable performance degradation while at the same time significantly reducing the power consumption.

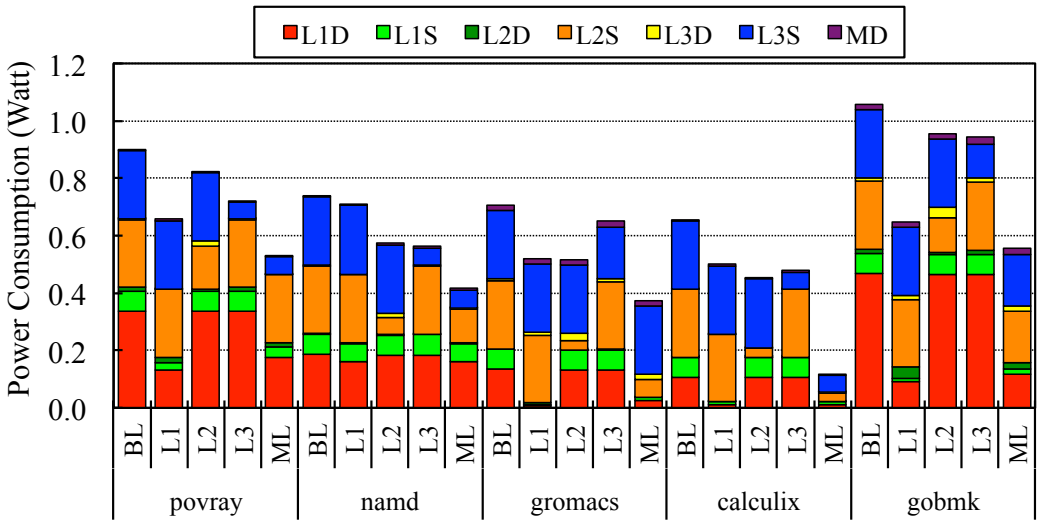


Figure 4.5: Power breakdown of workloads with low MPKI (first group).

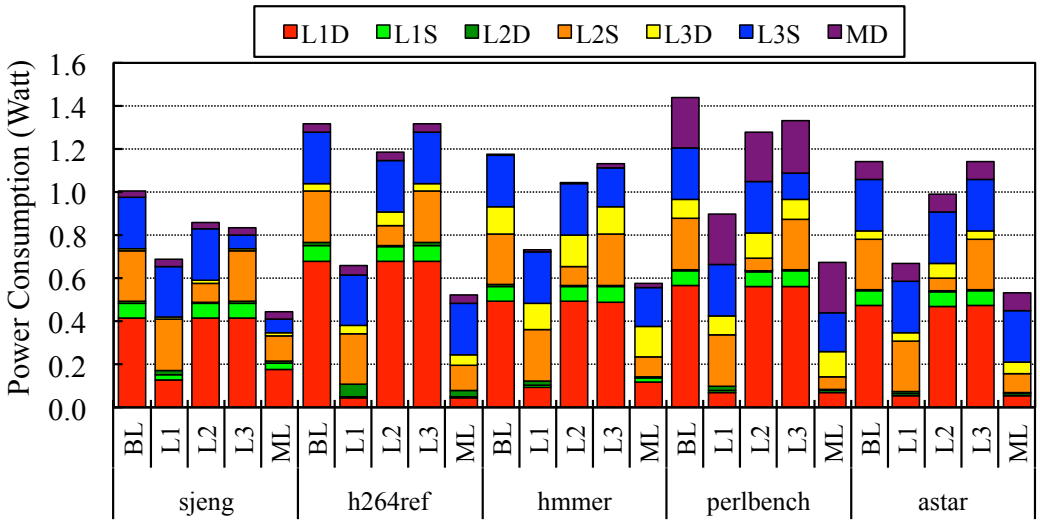


Figure 4.6: Power breakdown of workloads with low MPKI (second group).

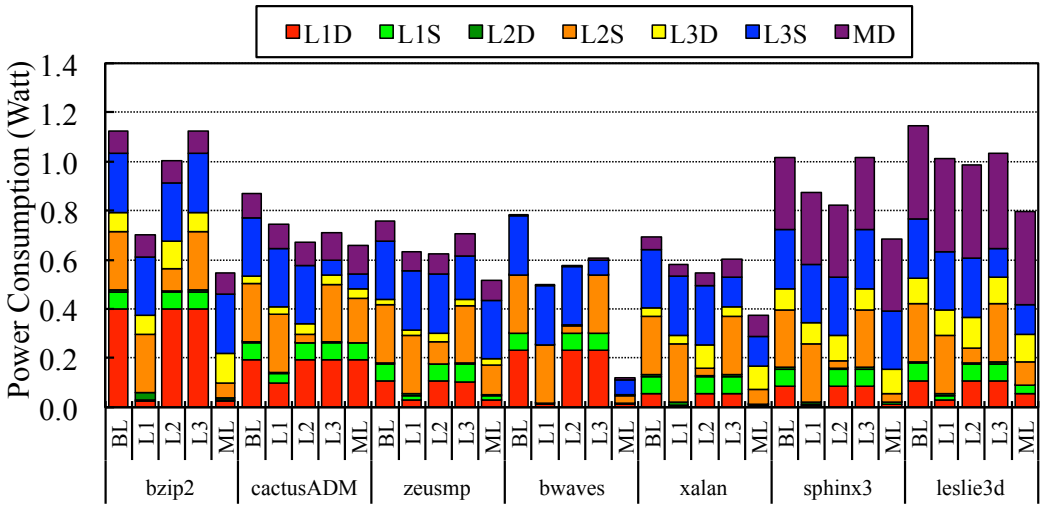


Figure 4.7: Power breakdown of workloads with medium MPKI.

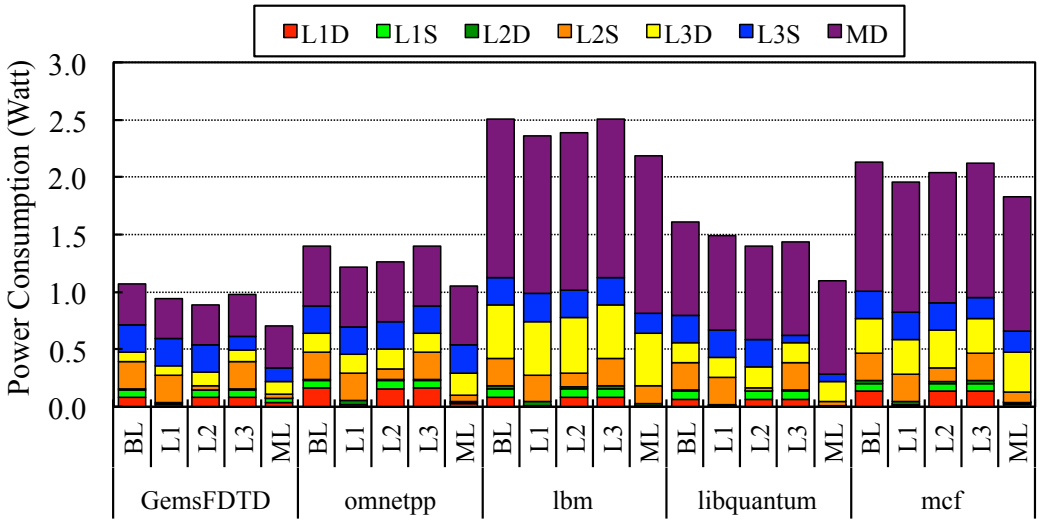


Figure 4.8: Power breakdown of workloads with high MPKI.

Chapter 5

Greedy Coordinate Descent Method

Given a deep cache hierarchy, Chapter 4 uses exhaustive search to identify the best cache resizing configurations. In this chapter, we develop a feasible algorithm for finding the best resizing configurations that can eventually be used in an online fashion. We propose to solve the multi-level cache resizing problem using the coordinate descent method [23]. Since there is no generalized solution for a constrained coordinate descent method, we take a greedy approach to iterate between coordinate directions and to select subsequent points in the search space to visit. In the following sections, we first discuss our prediction-based approach that does not require direct evaluations. And then, we define a greedy way in which our coordinate-descent method to iterate to find the optimal cache sizes.

5.1 Analytical Model for MCR

Consider solving a problem of finding optimal multi-level cache sizes for a uniprocessor. Let x, y, and z be the cache sizes in a three-level cache hierarchy. Let $f(x, y, z)$ be the power consumption of the system consisting of a core with a three levels of cache. Let $g(x, y, z)$ be the performance of the system.

$$Power = f(x, y, z), \;\; Performance = g(x, y, z)$$

Let μ be the normalized value of a degraded performance level (*i.e.* between 0 and 1) relative to the cache hierarchy with baseline sizes of x_B, y_B and z_B.

The goal of studying this analytical model is to provide a mathematical background to solve the constrained optimization problem. To achieve this goal, we present a generalized form of the coordinate descent method and explain our greedy-based heuristic iterations.

A simple form of the problem we want to solve is

$$\text{minimize } f(x,y,z), \text{ subject to } g(x,y,z) \geq c = \mu g(x_B, y_B, z_B) \tag{5.1}$$

The Lagrangian for this is

$$\mathcal{L}(x,y,z,\lambda) = -f(x,y,z) + \lambda(g(x,y,z) - c) \tag{5.2}$$

So,

$$\frac{\partial \mathcal{L}}{\partial x} = -\frac{\partial f(x,y,z)}{\partial x} + \lambda \frac{\partial g(x,y,z)}{\partial x} = 0 \tag{5.3}$$

$$\frac{\partial \mathcal{L}}{\partial y} = -\frac{\partial f(x,y,z)}{\partial y} + \lambda \frac{\partial g(x,y,z)}{\partial y} = 0 \tag{5.4}$$

$$\frac{\partial \mathcal{L}}{\partial z} = -\frac{\partial f(x,y,z)}{\partial z} + \lambda \frac{\partial g(x,y,z)}{\partial z} = 0 \tag{5.5}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = g(x,y,z) - c \geq 0 \tag{5.6}$$

From Eq. 5.3, 5.4 and 5.6 we obtain the equations,

$$\nabla f(x,y,z) = \lambda \nabla g(x,y,z) \text{ and } g(x,y,z) \geq c, \tag{5.7}$$

which are satisfied if and only if (x,y,z) is a minimum of $f(x,y,z)$ and satisfies $g(x,y,z) \geq c$ [59].

5.2 Performance Approximation

Solving Eq. 5.7 is very challenging. One of the most challenging parts is performance characterization, *i.e.* $g(x, y, z)$, because even differences between two configurations, $\Delta g(x, y, z)$, require performance evaluations and evaluating performance is unstable in a time-varying system. For this reason, we approximate the solution by replacing the performance constraint such that

$$g\prime(x, y, z) \propto g(x, y, z) \text{ and } g\prime(x, y, z) \geq c = \mu g\prime(x, y, z) \tag{5.8}$$

Rather than performance (*e.g.* IPC), we use a proxy, $g\prime(x, y, z)$, that mimics performance for the purposes of optimization, but which is much easier to predict. In particular, we use AMAT (Average Memory Access Time) for $g\prime(x, y, z)$. The main benefit of using AMAT is that we can predict it with performance events, *e.g.* cache-miss counts, so that we are not dependent on direct performance evaluations.

5.3 Stack Distance Measurement

AMAT can be computed from cache miss and memory reference counts. In this section, we review techniques for predicting cache misses given cache of different sizes. These techniques are based on stack distance profiling [60].

5.3.1 Stack Distance

Assume a memory-reference stream, $A_0, B_0, C_0, A_1, B_1, D_0, C_1, B_2, A_2, A_3, B_3....$ The stack distance of a given memory reference is defined by the number of unique memory references between the reference and its previous use. For example, the

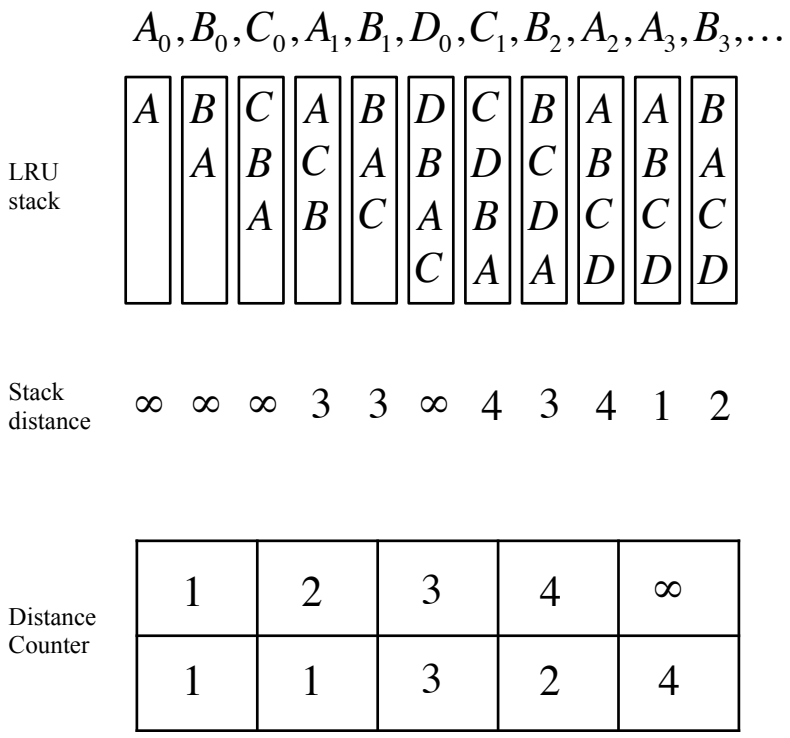


Figure 5.1: LRU stack, stack distance, and distance counter.

stack distance of the first reference to a memory block, such as A_0, B_0, C_0, and D_0, is ∞. The stack distances of A_1, B_1, C_1, B_2, and A_3 are 3, 3, 4, 3, and 1, respectively. Again, stack distance counts only unique memory references. Stack distance of A_2 and B_3 are 4 and 2, respectively. Figure 5.1 illustrates LRU stack and corresponding stack distance and distance counters.

Stack distance counters provide cache miss information for caches that maintain an LRU policy. Figure 5.1 shows a histogram, providing the counts of stack distance value from the example address trace. As shown in Figure 5.1, the number of references with *infinite* stack distance signify the cold misses. In Figure **??**, there are 4 such occurrences. If we have a cache size of 4, there are no more cache misses besides these cold misses. The distance counters provide complete informa-

tion about cache misses corresponding to any cache size. For example, if we have a cache size of 3, there will be two additional misses, *i.e.* capacity misses, which would not have occurred if the cache size were 4. In general, stack distance counters provide cache hit and miss information for caches of any capacity.

5.3.2 Way Counters

Stack distance counters provide cache miss-rates as a function of cache size. Despite the benefit of stack distance counters, there are two problems. First, cache miss-rates based on stack distance counters are only strictly valid for a fully-associative cache because the LRU stack does not consider set-associative caches. Second, implementing stack distance counters is nontrivial. Stack distance counters require memory reference traces and LRU stack profiling. The overhead of implementing an LRU stack is significant because it requires nontrivial amount of memory space to store LRU information and efficient algorithm to maintain the LRU property.

To solve these problems, Suh *et. al.* proposed a low overhead, on-line memory monitoring scheme: *way counters* [19]. Way counters utilize existing hardware in a cache to capture LRU information and require negligible extra hardware to store distance counter values. Although way counters approximate cache misses, it has two main strengths. First, way counters account for real cache hits and misses hence it results in better characterization of cache behavior. Second, way counters work at runtime, and require minimal extra hardware.

For these reasons, we use way counters to approximate stack distances and employ them to drive cache resizing. For simplicity, we only use way counters and exclude set counters because we have sufficient cache associativity, and assume uniform memory reference distribution across sets. In other words, we use aggregate way counters throughout all sets in the cache rather than maintaining separate way counters per set. Figure 5.2 shows a working example for way counters. As shown, way counter values approximate the stack distance counters in Figure 5.1. In this example, we assume two cache sets. Notice, we can approximate the stack distance at distances 2 and 5 by adding stack distance counters 1 and 2, and counters 3 and 4, for new counter 0 and 1, respectively. However, the final way counter values are 3 and 4. This is because stack distance for a fully associative cache can not account for the set mappings of individual memory references. On the other hand, way counters have the advantage of reflecting the actual set mappings in an existing cache organization.

Figure 5.3 illustrates the implementation of way counters. As shown, same way LRU from two different sets will increase the same way counters by aggregating way LRU using a MUX and the way LRU will increase corresponding way counters through DEMUX.

5.4 Power Efficiency Gain

We predict performance and power savings based on way counter values. Utilizing this prediction, we search an optimal cache hierarchy configuration. To enable

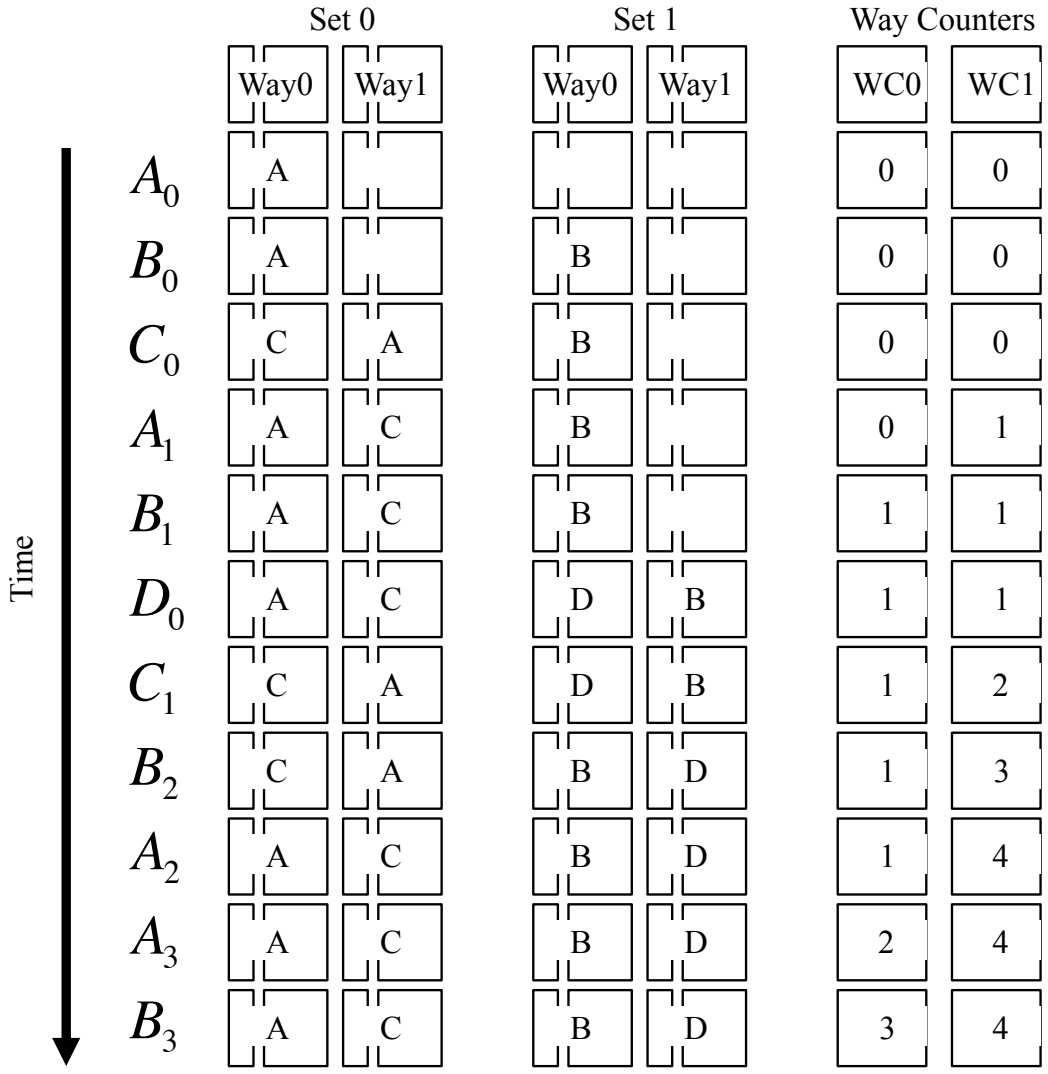


Figure 5.2: Way counter workflow.

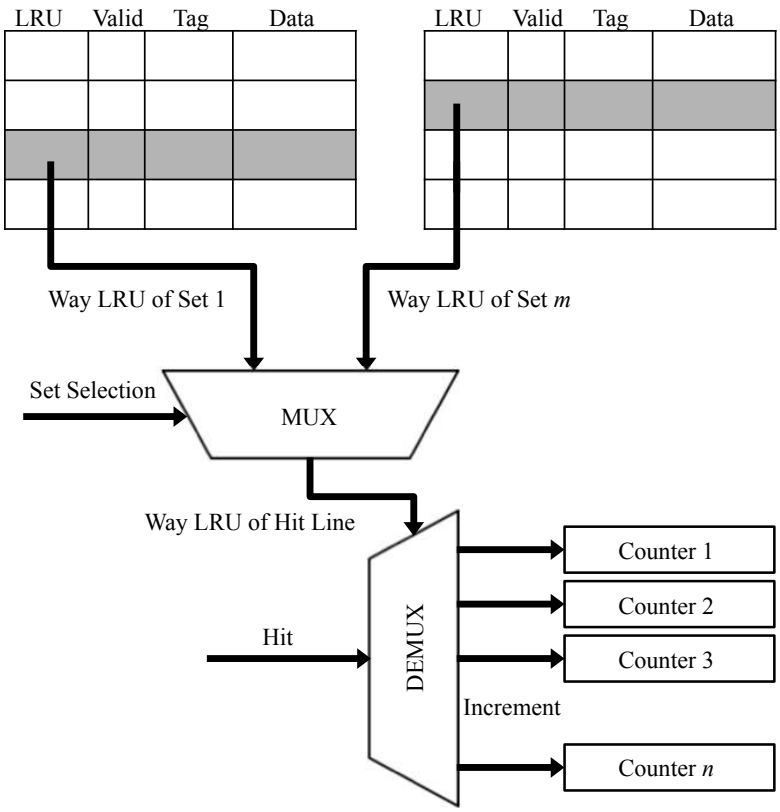


Figure 5.3: Way counter implementation.

such search iterations following the coordinate descent method, we define new metric, *power efficiency gain* (PEG), to select a coordinate and a subsequent point that will maximize power savings. We define PEG as the power savings per unit performance degradation. If $power_a$ and $power_b$ are two power consumption levels that two configurations of a system dissipate when the system receives cache capacities of a and b respectively, and $performance_a$ and $performance_b$ are their corresponding performance levels, then the power efficiency gain, PEG_b^a of decreasing the capacity from a to b is defined as,

$$PEG_b^a = (power_a - power_b)/(performance_a - performance_b) \tag{5.9}$$

Fundamentally, PEG approximates Equation 5.15. Our greedy iteration reduces the number of iterations by searching the maximum PEG value per caching level. In other words, we solve Equation 5.7 by disabling the way of a caching level with maximum power efficiency gain. The *pseudo* code for the our GCD MCR algorithm is shown in Algorithm 1.

5.5 Greedy Coordinate Descent Method

The problem, as defined in Equation. 5.1, is a nonlinear equation and solving it is very challenging mainly due to the infeasibility of characterizing the performance and power functions. Thus, we resort to an iterative method. The iterative methods are commonly used to solve problems of nonlinear programming, which evaluate gradients or function values. First, we exclude heuristics because it requires performance evaluations of intermediate solutions which may degrade the performance

Technique	Complexity	Quality	Feasibility	Example
Direct methods	High	High	Extremely low	Lagrangian multiplier
Iterative methods	Medium	High	Medium	Newton's method, Conjugate gradient methods, Gradient decent
Heuristics	Low	Medium	High	Hill climbing, Genetic algorithms, Nelder-Mead method
Coordinate descent methods	Low	High	High	MCR

Table 5.1: Optimization technique classification.

severely, but we compare MCR to one popular heuristic in Section 6.3.1, the Nelder-Mead simplex method.

Iterative methods commonly evaluate Hessians or gradients to solve problems. Among these, we can generalize the sequence of such steps to solve a problem using gradients,

$$\mathbf{x}_{n+1} = h(\mathbf{x}_n, \nabla f(\mathbf{x}_n)), n \geq 0. \tag{5.10}$$

Solving the problem by following a sequence in Equation 5.10 is more feasible than directly solving Equation 5.7. For example, the gradient descent method is a well-known [61] optimization method to find a local optimum. The number of steps it takes is proportional to the negative of the gradient function. However, we do not usually have gradient functions in cache resizing, so we are better of considering

non-derivative optimization. In particular, coordinate descent [23] is very popular to solve non-differentiable functions. Compared to other techniques such as gradient descent and Nelder-Mead, coordinate descent has strength in its capability to solve huge problems efficiently [62]. The iterative steps of coordinate descent, Nelder-Mead, and greedy iterative, method are illustrated in Figure 5.4.

Despite its efficiency, the coordinate descent can still potentially require a large number of steps because each step cyclically iterates through each direction, *i.e.* the coordinate direction of each variable, minimizing objective functions. The other problem is that coordinate descent works well with unconstrained functions. However, we have a constraint: the bounded performance degradation. For this reason, we propose a greedy approach which utilizes the observation from Equation 5.7: parallel gradients of performance and power functions.

Greedy algorithms are well known in the literature for solving computer science/engineering problems due to their low complexity and their ability to obtain locally optimal, and in some cases, globally optimal solutions [61]. Note, our use of the greedy approach simply refers to the sequence, on each iteration, of taking the decision that provides the best immediate solution which is close to the state of two parallel gradients above. Here, we clarify that we do not use the greedy approach in a heuristic way to solve the problem. Instead, we approximate the Lagrangian as in Equation 5.7 by using the coordinate descent method and by generating its descent via a greedy-style approach.

In other words, our approach uses a combination of features from the common approaches to derive an alternative method to approximate the solution of

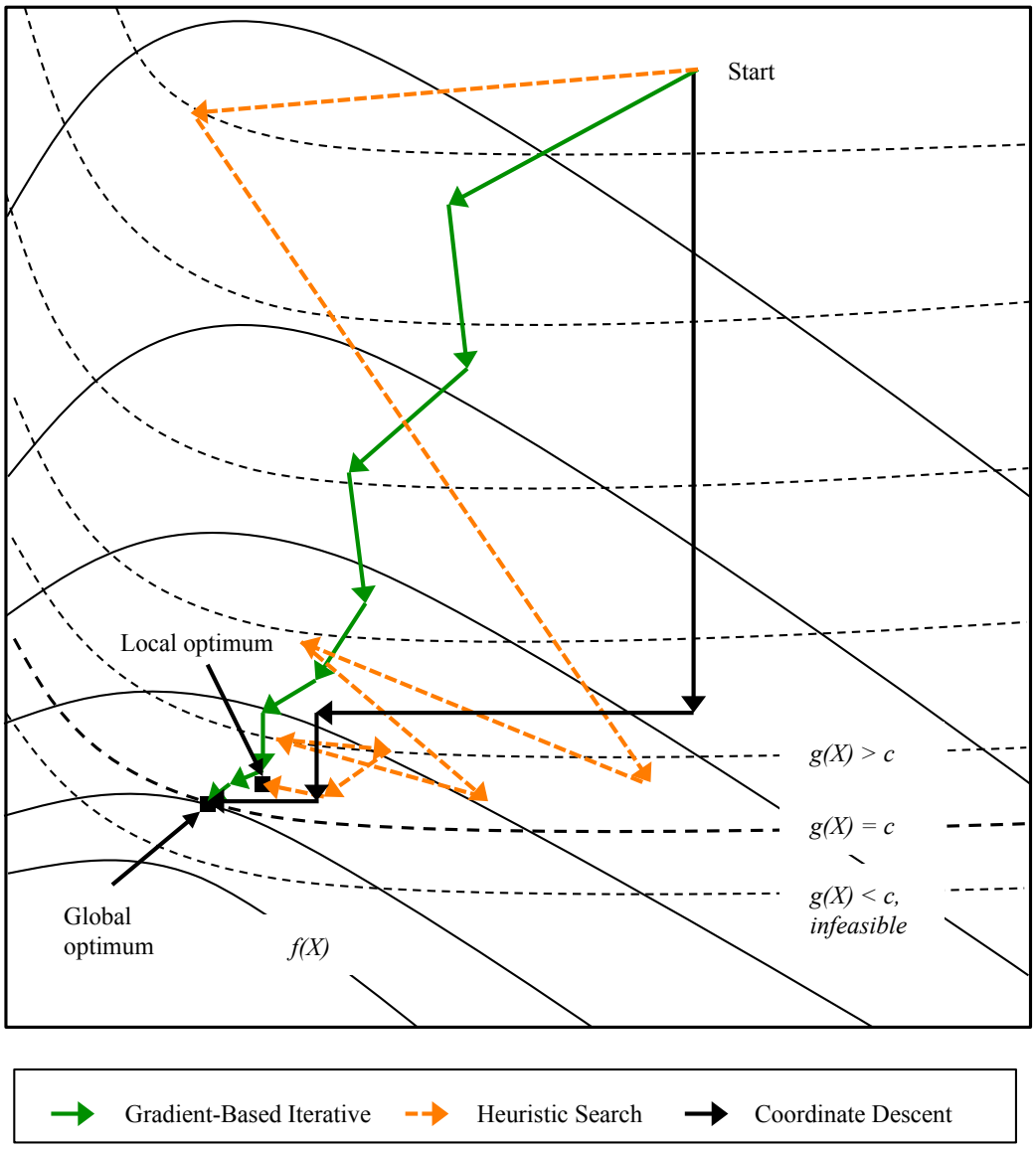


Figure 5.4: Comparison between *Coordinate Descent*, *Nelder-Mead*, and *Gradient Descent* method. Contour lines of objective function $f(x)$ and constraint function $g(x)$ are plotted in solid lines and dotted lines, respectively.

Equation. 5.7. Again, our goal is to design a method to achieve high performance with low complexity. To achieve this, we propose an iterative greedy-style algorithm following coordinate descent method in the way that makes two gradients parallel

$$\nabla f(x, y, z) = \lambda \nabla g(x, y, z). \tag{5.11}$$

We define sequence function, $h(\mathbf{x}_n, \nabla f(\mathbf{x}_n))$ as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \vec{v} \tag{5.12}$$

such that

$$\vec{v} = k\hat{\mathbf{e}}_i \text{ that maximize } \frac{f(\mathbf{x}_n) - f(\mathbf{x}_n + \vec{v})}{g(\mathbf{x}_n) - g(\mathbf{x}_n + \vec{v})}, \tag{5.13}$$

where

$$\hat{\mathbf{e}}_i \in \{\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, ...\hat{\mathbf{e}}_{\text{num of variable}}\}. \tag{5.14}$$

In our greedy approach, we eliminate outlier, *i.e.* maximum value of

$$\frac{\partial f(\mathbf{x})}{\partial \hat{\mathbf{e}}_i} / \frac{\partial g(\mathbf{x})}{\partial \hat{\mathbf{e}}_i}, \tag{5.15}$$

to converge quickly to the parallel state of Equation 5.11 as shown in Equation 5.13. The *pseudo* code for our method for the problem is shown in Algorithm 1.

Algorithm 1: Greedy Coordinate Descent Method

```
1.1  greedy_coordinate_descent_method():
1.2  begin
1.3      while g(x) ≥ c do
1.4          foreach level i do
1.5              candidate x_i = get_outlier(x, i)
1.6          end
1.7          next x = maximum of candidate x_i
1.8      end
1.9  end
1.10 get_outlier(x, i):
1.11 begin
1.12     foreach available k do
1.13         candidiate x_k = x + k ê_i
1.14         evaluate (f(x) − f(x_k))/(g(x) − g(x_k))
1.15     end
1.16     return maximum of candidiate x_k
1.17 end
```

Chapter 6

GCD Uniprocessor Multi-Level Cache Resizing

In this chapter, we design *greedy coordinate descent multi-level cache resizing algorithm* that searches for the optimal cache configuration for a uniprocessor. We implement the PEG-based sequence function that we defined in Chapter 5. As we described, *PEG* approximates the partial derivative of the gradient in our constrained multi-variable optimization which consists of the objective function, total power consumption of a given cache hierarchy, and the constraint function, bounded performance degradation. As a result, we realize an evaluation-less search utilizing our *PEG*-based greedy coordinate descent method.

6.1 GCD Uniprocessor MCR Algorithm Design

6.1.1 GCD MCR Algorithm Description

GCD MCR Framework We optimize multi-level cache configurations per cache level at each iteration cyclically. Each cache size is mapped to a separate coordinate within our greedy coordinate descent method and a new configuration is determined per iteration according to the allocation that has maximum PEG. Figure 6.1 illustrates the framework for MCR. First, each caching level emits way counter values every corresponding epoch and these values are stored in the way-counter registers. Second, MCR logic concurrently compares PEGs from each caching level and re-

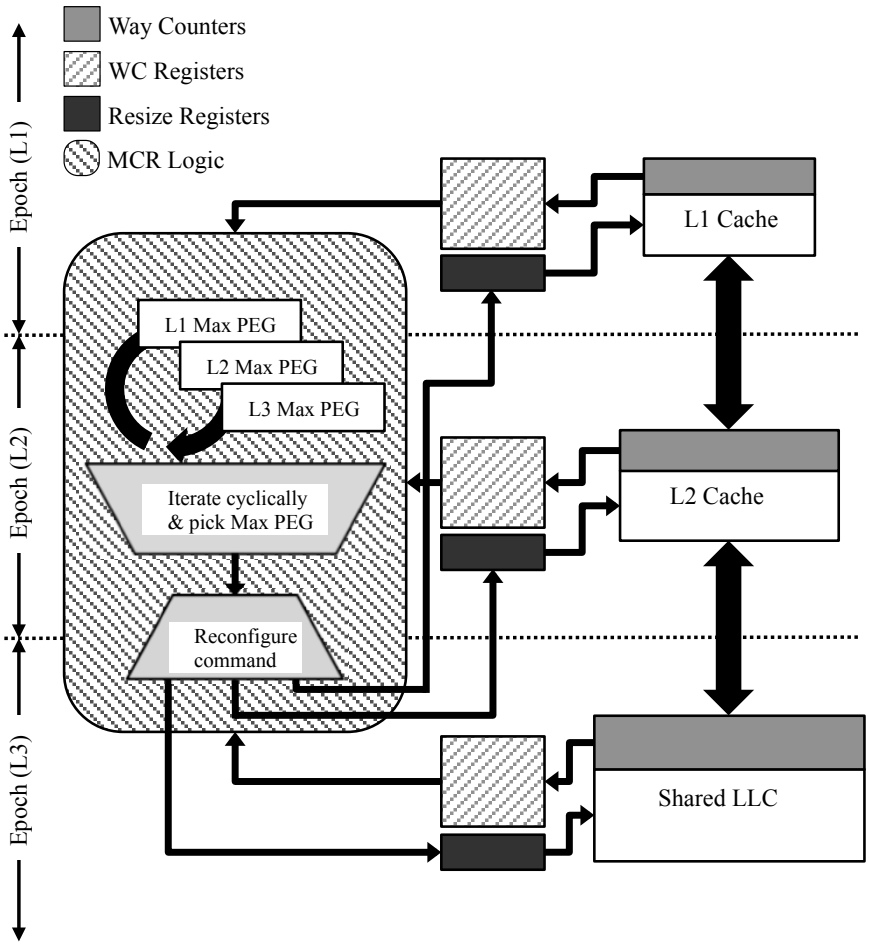


Figure 6.1: MCR framework.

sizes the caching level, for each corresponding epoch, which the has maximum PEG value. Finally, MCR stops if no PEG value greater than zero presents.

Power Efficiency Gain (PEG) We implement Algorithm 1. To do so, we first design a method to generate a search sequence (Equation 5.12) by defining the search vectors (Equation 5.13). Algorithm 3 shows the pseudo code for computing PEG in our study. Line 3.12 computes Δ power by resizing the cache from a to b.

In particular, Δ power at a down sizing event is the sum of the power reduction of the dynamic power consumption and the static power consumption of the cache,

and the increased power consumption of the next level of cache. For the static power reduction, all three caching levels exhibit static power consumption roughly in proportion to their sizes. On the other hand, only the L1 and L2 caches show near-proportional dynamic power consumption to their sizes because the L3 cache exploits serialized data array accesses after a tag hit to optimize the dynamic power consumption in the organization. The increased power consumption of the next level of cache is determined by the additional cache misses caused by the down sizing and the dynamic access energy of the next-level cache.

Δ AMAT can be estimated by calling the *get_amat*() procedure in Algorithm 4. The difference between computing *balance* in Line 2.7 and Δ AMAT in 3.13 is that Δ AMAT considers additional cache misses by adding the way counts associated with each disabled way.

Because of the Δ power computation, our GCD MCR algorithm does not shrink a cache beyond the balance point that saves more power consumption from the dynamic and static power consumption of the cache due the current cache's resizing than the increased power consumption from the dynamic power consumption of the next-level cache. Moreover, PEG enables comparisons across caching levels because it provides the metric of power saving per unit AMAT increase to which different caching levels contribute separately. As such, the maximum PEG value from the caching levels determines the search direction in the coordinate system.

Uni-directional Searches Algorithm 2 describes the outermost loop of our GCD MCR algorithm. The *gcd_mcr*() procedure is invoked every Epoch within the corre-

sponding interrupt handler and it continues to down size the cache hierarchy within the performance degradation range. To prevent oscillation, our GCD MCR search direction is uni-directional towards smaller caches. However, to make our GCD MCR algorithm responsive to dynamically changing behavior, we start a new search every *reset period*, as shown in Line 2.13. Moreover, we update each set of way counters at different frequencies. As shown in Lines 2.10, 2.11, and 2.12, we update each set of way counters at certain epoch boundaries only. Also, reconfiguration can only happen at those corresponding epoch boundaries.

The *while* loop in Line 5.16 iterates until there is no more candidate for further down sizing in the corresponding cache level according to the Epoch count. As shown in Line 2.17, the *while* loop terminates if the winner does not belong to the current resizable cache level.

6.1.2 Scalability of GCD MCR Algorithm

We measure the time complexity of our GCD MCR algorithm using two metrics: search complexity and computation complexity. We define search complexity as the number of iterations (epochs) needed to find the optimal cache configurations. For the Nelder-Mead simplex method, search complexity is the number of steps until it converges.

On the other hand, computation complexity is measured in the number cycles to execute the algorithm at each epoch. For example, the computation complexity of the Nelder-Mead simplex method is $\boldsymbol{O(nm)}$, where n is the number of cores and

Algorithm 2: GCD MCR

```
2.1  gcd_mcr(perfLimit):
2.2  begin
2.3      /* main epoch loop */
2.4      while true do
2.5          totalAccesses = sum of caching level zero's all way counts and misses
2.6          /* Maximum AMAT increase to guarantee performance */
2.7          balance = get_delta_amat(wcL, totalAccesses, perfLimit)
2.8          balance -= increased deltaAMAT because of current cache allocations
2.9          /* update PEG at different frequencies */
2.10         if not (epoch % L1_FREQ) then maxPEG[0] = update_max_peg(L1)
2.11         else if not (epoch % L2_FREQ) then maxPEG[1] = update_max_peg(L2)
2.12         else if not (epoch % L3_FREQ) then maxPEG[2] = update_max_peg(L3)
2.13         else if not ( epoch % RESET_FREQ) then start new search
2.14         /* computation loop*/
2.15         while balance do
2.16             winner = caching level with maximum value of maxPEG
2.17             if winner is from the caching level can be reconfigured then
2.18                 balance = balance – winner.deltaAMAT
2.19                 allocL[winner.level] = winner.alloc
2.20                 update_max_peg(current_level)
2.21             else break
2.22         end
2.23         /* Send reconfiguration command */
2.24         if allocL is changed then reconfigure()
2.25     end
2.26 end
```

Algorithm 3: GCD MCR (continued)

```
3.1  update_max_peg(current_level):
3.2  begin
3.3      alloc = current allocation of caching level of current_level
3.4      balance = current remained balance
3.5      foreach available way-off i do
3.6          peg[i] = get_peg_value(wc, alloc, alloc – i , balance)
3.7      end
3.8      winner = allocation with maximum peg value return winner
3.9  end
3.10 get_peg_value(wc, a, b, balance):
3.11 begin
3.12     deltaPower = sum of dynamic and static power changes caused by change in misses
         when the assigned way decreases from a to b
3.13     deltaAMAT = increased AMAT caused by change in misses when the assigned way
         decreases from a to b
3.14     peg = 0
3.15     if deltaAMAT is smaller than balance then
3.16         peg = deltaPower / delatAMAT
3.17     end
3.18     return peg
3.19 end
```

Algorithm 4: GCD MCR (continued)

```
4.1  get_delta_amat(wcL, totalAccesses, perfLimit):
4.2  begin
4.3      AMAT = get_amat(wcL, totalAccesses) /* predict AMAT with given way counts */
4.4      /* convert perf limit to AMAT limit */
4.5      deltaAMAT = AMAT * (1.0 – perfLimit) / perfLimit
4.6      return deltaAMAT
4.7  end
4.8  get_amat(wcL, totalAccesses):
4.9  begin
4.10     AMAT = cacheLat[0] /* all data reference goes to level-1 cache */
4.11     foreach level i do
4.12         AMAT = AMAT + cacheLat[i+1] * misses of caching level i / totalAccesses
4.13     end
4.14     return AMAT
4.15 end
```

m is the number of caching levels. For this uniprocessor study, it is $\boldsymbol{O(m)}$ because the Nelder-Mead simplex method requires comparisons between $m+1$ vertices per iteration.

From the perspective of search complexity, finding the optimal cache sizes in a multi-level cache hierarchy is an *NP-hard* problem as the depth increases. This is because it has a search complexity of $\boldsymbol{O(k^m)}$, where the cache hierarchy has m-level caches and each cache has up to k configurable sizes. We reduce the search complexity down to $\boldsymbol{O(km)}$ in worst case, but $\boldsymbol{O(k)}$ in average case due to our greedy approach which reduces the number of iterations of the main *while* loop. Asymptotic analysis is shown in Algorithm 5.

The worst case arise when all km PEG values are sorted and interleaved across different caching levels. In this case, every epoch iteration in the main loop in Line 5.7 will traverse between different caching levels in the PEG order. Moreover, because of the sorted PEG order in each caching level, the cache reconfiguration step in Line 5.20 will be fixed at one. Therefore, in such a case, our GCD MCR algorithm exhibits $\boldsymbol{O(km)}$ search complexity.

However, in the average case, the PEG values are not in sorted order and our GCD MCR algorithm does not suffer from alternating resizing between caching levels nor from the fixed down stepping of each one. In the average case, it can be done within $\boldsymbol{O(k)}$ epochs because moderate numbers of alternations between caching levels are bounded by the maximum configurations. We will present search time results to show the search complexity in the average case in Section 6.3.1 of Chapter **??**.

From the perspective of computation complexity, our GCD MCR algorithm exhibits $\boldsymbol{O(k^2)}$. We can reduce the complexity down to $\boldsymbol{O(km)}$ by comparing the LRU way's PEG value. However, it might cause a suboptimal search for not considering the best PEG value per level before comparing these PEGs across different caching levels. Besides, this will increase the number of alternating PEG comparisons across caching levels of the main Epoch loop in Line 2.4 of Algorithm 2. Moreover, search complexity is more important than computation complexity in our study because each search takes more cycles compared to that of the computation orders of magnitude. As such, we realize the computation complexity of $\boldsymbol{O(k^2)}$ both to reduce the search complexity and to enhance the chances of finding the global optimum.

6.1.3 Hardware Overhead

To predict power consumption and AMAT of a given cache hierarchy, we use shadow tag arrays [63, 11]. A shadow tag is similar to a regular cache structure, but has no data array. The major source of hardware overhead is the shadow tags. Shadow tags require $s * w * t$ bits, where s is the number of sampled sets, w is the number of ways, and t is the tag entry bits. Way counters require $4 * w$ bits. In addition to the storage bits, we need an adder for incrementing the way counters.

Our shadow tags contain one sampled set per 32 regular tags and we assume a 42-bit physical address. The shadow tags array requires 4034 Bytes of overhead which is an increase of 0.15% in the storage requirement compared to the base line

Algorithm 5: GCD MCR Complexity Asymptotic Analysis

```
5.1  gcd_mcr():
5.2  begin
5.3      ...
5.4      /* Search Complexity */
5.5      /* loop will be executed O(km) in the worst case, however */
5.6      /* our greedy approach enhances convergence rate, hence O(k) in the average case */
5.7      while true do
5.8          /* Computation Complexity */
5.9          /* Initialization requires O(km) */
5.10         ...
5.11         /* update PEG at different frequencies */
5.12         /* update_max_peg requires O(k) */
5.13         ...
5.14         /* computation loop*/
5.15         /* this computation loop requires O(k^2) */
5.16         while balance do
5.17             ...
5.18         end
5.19         /* Send reconfiguration command */
5.20         if allocL is changed then reconfigure()
5.21     end
5.22 end
```

Level	L1	L2	LLC
Sets	64	512	8192
Ways	8	8	4
Size of tag entry (bits) (valid bit + tag bits + LRU bits)	33 (1 + 29 + 3)	30 (1 + 26 + 3)	26 (1 + 23 + 2)
Tags (Byte)	2112	15360	106496
Data (KB)	32	512	2048
Area of baseline cache (KB)	34	527	2152
Shadow tag sets	2	16	256
Shadow tags (Byte)	66	480	3328
Way counters (Byte)	64	64	32
Area overhead of shadow tags (Byte)	130	544	3360
Total area overhead (%)	0.37	0.1	0.15

Table 6.1: Storage overhead of shadow tags.

caches. Table 6.1 shows area overhead details for the shadow tags.

6.1.4 Implementation

We implement our GCD MCR algorithm in the simulator from Section 4.3.2 of Chapter 4. In particular, we modify our simulator to emit an interrupt every epoch (epoch size will be discussed in Section 6.3.3), and execute an interrupt handler. The interrupt handler, which runs on our modified SimpleScalar simulator, gathers way

counts and sends them to the MCR server which runs as another process. We also start to run new GCD MCR search after a certain number of epochs has elapsed. This *reset interval* will also be discussed in Section 6.3.3. We also modify the simulator to allow software to reconfigure the caches within the interrupt handler.

Performance and power estimations are calculated within the MCR server. In particular, the MCR server requires per-access energies and leakage energies from CACTI to predict power consumption per epoch based on the way counts, so we implement configurable registers to store these per-access energy values. These energy registers can be implemented either in hardware or in software.

Cache Reconfiguration To enable cache reconfiguration, we modified Simplescalar's cache module to model selective ways [2] for the L2 cache and LLC, and selective sets and ways [8] for the L1 cache. We assume all caches in the hierarchy, except for the L1 I-cache, are reconfigurable and can change their size in increments of a cache way from 1 to the associativity number of ways in the L2 cache and LLC, and of a corresponding number of sets and ways in the L1 cache. (Our work does not consider I-cache resizing, and assumes the I-cache is always fixed). Hence, for our hierarchy, there are 12, 8, and 32 different configurations for the L1, L2, and L3 caches, respectively. In the static-optimal version of MCR from Chaper 4, we try all possible permutations of the per-cache configurations to identify the one that is most power-efficient. While each cache's access delay also changes across different configurations, we assume a constant number of CPU cycles to access each cache chosen to handle that cache's worst-case access delay (*i.e.* with all ways enabled).

Our simulator accounts for the overheads associated with resizing each cache. When up-sizing the caches, we assume 2/4/7 cycles to power up and to flash invalidate each way for L1/L2/LLC respectively. When down-sizing, we walk the down-sized way(s) to flush their contents. Clean cache blocks are discarded after checking upstream caches to maintain inclusion. Dirty cache blocks check upstream caches and are also written back to the next-lower level. We assume these operations are pipelined such that flushing takes 1 cycle per walked cache block. Down-sized ways are selected in reverse way ID order. Because we do not physically move cache blocks once they are filled, the flushed cache blocks have an equal probability of being at any position in the LRU stack. Moreover, we do not attempt to reconstruct the per-set LRU stacks after flushing.

Low-Pass Filter in Dynamic Cache Reconfiguration Online cache hierarchy reconfiguration introduces many issues, most of which are caused by transient occurrences in our GCD MCR algorithm. Moreover, such occurrences may form a positive feedback loop resulting in exacerbated performance degradation. As such, we employ two mechanisms in our technique: search-restart and low-pass filter. First, search-restart prevents forming a positive feedback loop along with the unidirectional search in Section 6.1.1. By resetting cache configurations to the baseline configuration, GCD MCR is not only able to avoid forming a feedback loop, but also can adapt to the dynamically changing phases of workloads. Second, we employ a low-pass filter in our GCD MCR. In our study, this is implemented by averaging previous cache configuration and the new configuration from the GCD MCR algo-

rithm. There are two reasons for having such a low-pass filter. Online GCD MCR utilizes way counts of the past epoch because our GCD MCR does not require any a priori information. As such, the solution of GCD MCR is speculative and brings the possibility of miss prediction. Moreover, there could be transient behavior due to the relatively fine granularity in the epoch size of our GCD MCR compared to other possible evaluation-based search techniques. As a result, we employ a low-pass filter to reduce the penalty from a miss prediction and the excessive numbers of reconfigurations due to transient behaviors.

6.2 Experimental Methodology

Static MCR We conduct an offline static MCR to examine its potential power savings and performance improvement compared to other schemes. This static MCR study has two goals: first, identify potential performance gains and power savings compared to the other techniques, and second, determine the limit on the maximum performance and the power savings from perfect information, *e.g.* way counts as a priori information. The latter will allow us to assess how well our dynamic MCR technique performs.

To facilitate the study, we take way counts of the baseline cache configuration, and use theses as input to Algorithm 2. Since the way counts of lower caching levels can be changed with a cache resizing at the upper level cache, we update way counts as the algorithm proceeds and needs way counts from different cache configurations as we already have exhaustive simulation results.

Random We conduct an offline evaluation of random search to compare static MCR to randomly generated solutions. We generate cache configurations randomly and evaluate their performance and power savings against the exhaustive simulation results. Note that we randomly select LLC sizes up to 16MB.

Nelder-Mead Nelder-Mead simplex method is a simple minimization algorithm which was first introduced in 1965 [64]. Since then, NM simplex method was commonly used in various fields of science, especially in multi-variable optimization problems, and this work has been cited more than 18,000 times so far. We conduct an offline evaluation of the NM method. We use a classical simplex method by Nelder and Mead, which consists of *Sort, Reflection, Expansion, Contraction, and Reduction* [64]. *Pseudo* code for this study is shown in Algorithm 6.

Epoch Size The choice of the epoch size affects the performance of our greedy iterative method. Reducing the epoch size generally provides a better chance to capture dynamic phase changes during a workload run, but at the same time encounters more reconfiguration overheads due to frequent reconfigurations. On the other hand, Increasing the epoch size generally reduces the reconfiguration overheads, but loses adaptability for its limited number of chances to reconfigure. We tried 40K, 200K, 400K, 500K, and 1M, cycle epoch sizes and pick a 200K cycle epoch size for this study because it exhibits adaptiveness with low overhead. Moreover, different caching levels employ different reconfiguration frequencies due to different access frequencies. For our study, the L1 cache reconfigures every epoch and the L2

Algorithm 6: Nelder-Mead Method

```
6.1  nelder_mead_method():
6.2  begin
6.3      while true do
6.4          sort vertices such that f(x_1) ≤ f(x_2) ≤ ... ≤ f(x_{n+1})
6.5          if all vertices are same or differences are within threshold then goto end
6.6          calculate center of gravity x_0 except x_0
6.7          reflection
6.8              compute reflected point x_r
6.9              if f(x_1) ≤ f(x_r) ≤ f(x_n) then replace x_{n+1} with x_r goto sort
6.10         expansion
6.11             if f(x_r) < f(x_1) then
6.12                 compute expanded point x_e
6.13                 if f(x_e) < f(x_r) then replace x_{n+1} with x_e goto sort
6.14                 else replace x_{n+1} with x_r goto sort
6.15         contraction
6.16             compute contracted point x_c
6.17             if f(x_c) < f(x_{n+1}) then replace x_{n+1} with x_c
6.18         reduction
6.19             for all but the best point, replace the point with reduced point
6.20     end
6.21 end
```

cache reconfigures every 5 epochs. Lastly, the LLC reconfigures every 25 epochs.

6.3 Evaluation of the GCD Uniprocessor MCR

In this section, we evaluate the power savings and the performance degradation of our GCD multi-level cache resizing implementation. We will first show the offline analysis and then, we will present the power savings and performance degradation of our online GCD multi-level cache resizing implementation.

6.3.1 Offline Analysis

The goal of this offline analysis is to understand limits of our GCD multi-level cache resizing implementation.

Total System Power Consumption and Performance We compare the power and performance of static MCR to three schemes: random, Nelder-Mead, and exhaustive. Random scheme shows wide variance both in power and performance. While its average shows only 0.7% performance degradation and 1.7% power increase, it degrades performance by as much as 7.1% and increases power by as much as 37.9%. Static MCR exhibits significant power savings, 8.8%, while Nelder-Mead achieves 9.1%. These results show that the static MCR's solution quality is slightly worse than that of the Nelder-Mead method. In terms of the performance degradation, both MCR and the Nelder-Mead method exhibit similar performance levels.

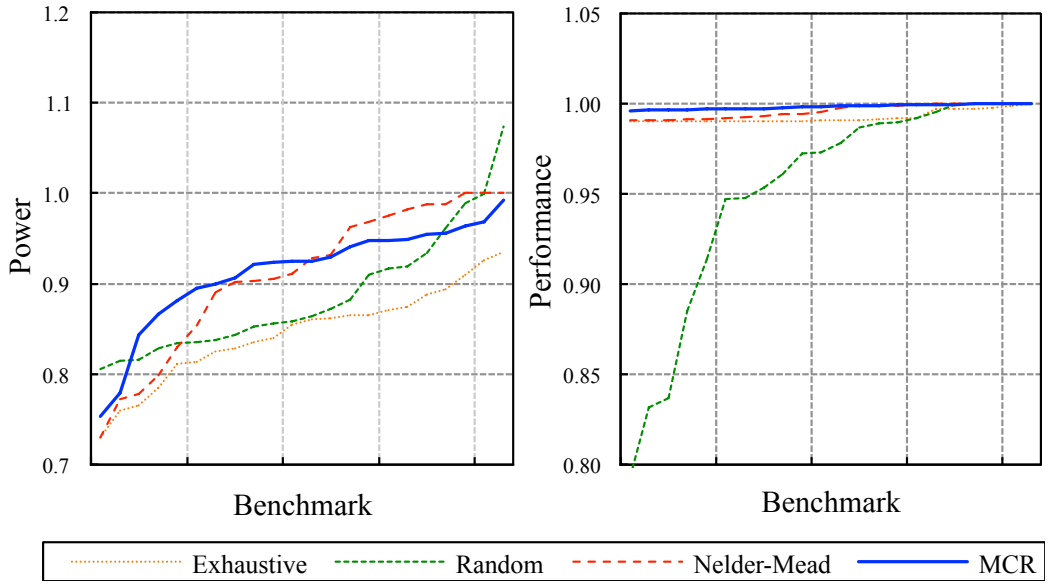


Figure 6.2: Power and performance comparison between random, Nelder-Mead, and MCR.)

Search Complexity Although Nelder-Mead exhibits slightly better off-line performance, significant benefit of MCR is its fast convergence rate. In other words, our GCD MCR algorithm requires a very small number of cache reconfigurations to find an optimal configuration. Figure 6.3 shows the number of iterations of the MCR and Nelder-Mead methods. The Nelder-Mead method requires almost 6 times the number of iterations at the MCR method requires, 3.2 compared to 18.9. As we discussed in Section 6.1.2, our GCD MCR exhibits fewer number of search iterations than the number of the worst case analysis.

Search Movement and Search Iterations Figure 6.4 illustrates the epoch taken by our GCD MCR method search iterations and cache reconfigurations for several representative workloads. Figure 6.4-(a), (b), and(c) show the cases that

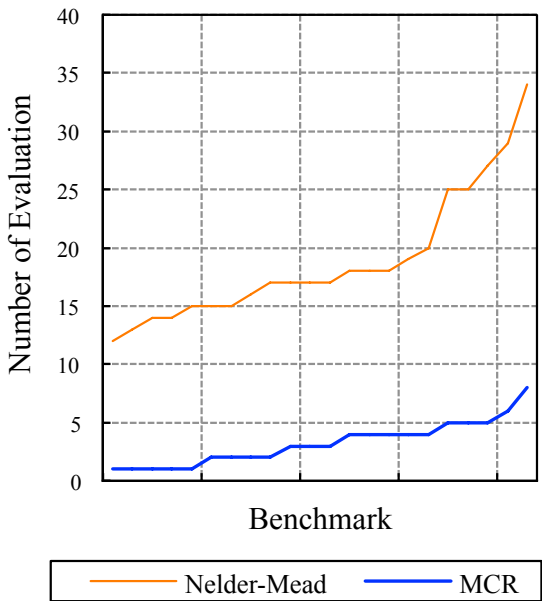


Figure 6.3: Number of evaluations comparison between Nelder-Mead and MCR.

invoke only one movement or reconfiguration per coordinate. In particular, *bzip2* requires only one reconfiguration to find the optimal configuration by downsizing the L1 cache all at once in Figure 6.4-(a). Likewise, *mcf* and *perlbench* require two and three search iterations to find the optimal configurations.

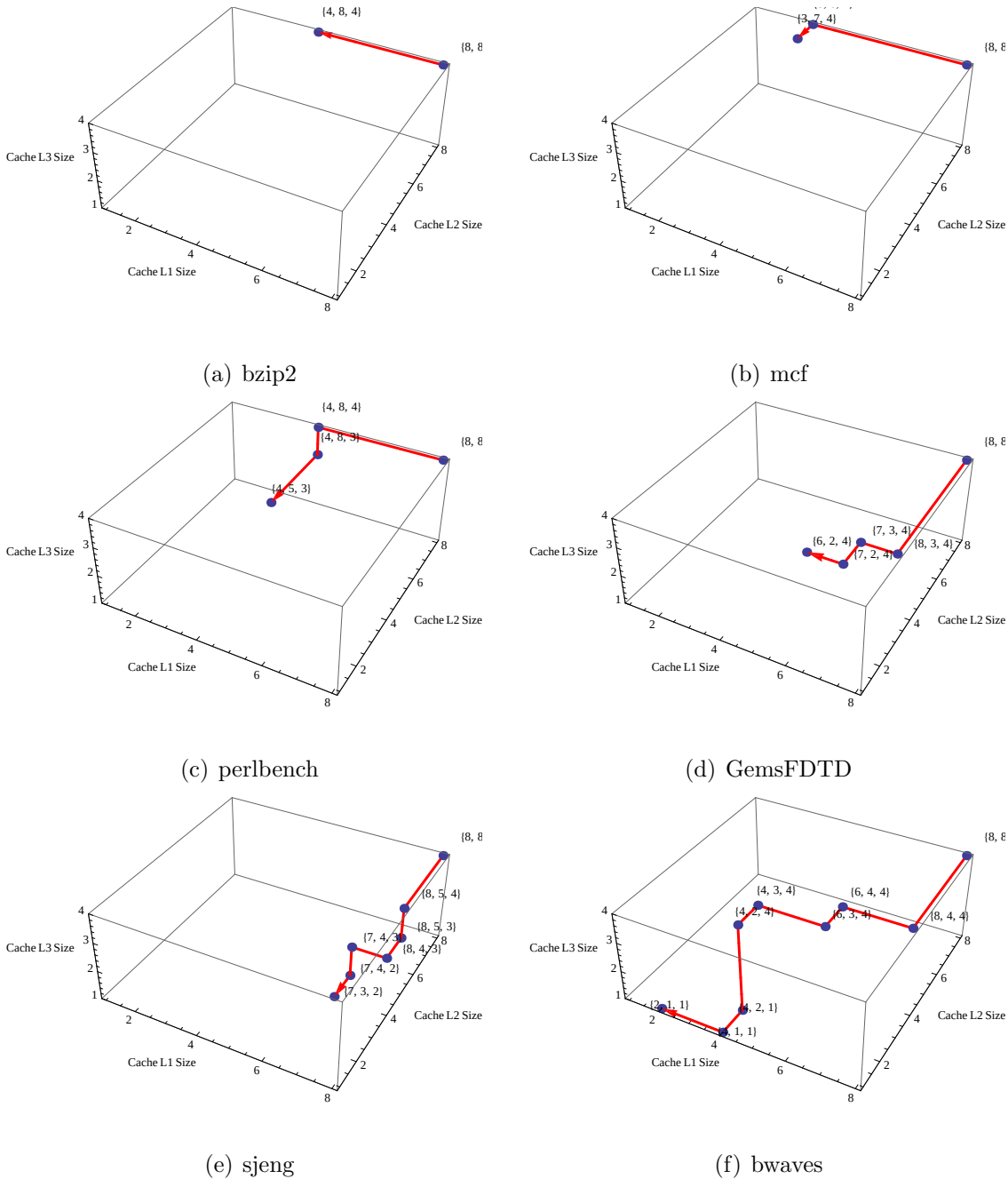
GemsFDTD exhibits PEG-ordered cache reconfiguration between the L1 and L2 caches in Figure 6.4-(d). First, GCD MCR shrinks the L2 size to 3 and continues to downsize the L1 cache to 7 because the L1's PEG value from 8 to 7 is bigger than the PEG value of L2 from 3 to 2. After that, downsizing alters again between the L1 and L2, and finally, reaches the optimal configuration of (6,2) for the L1 and L2 configurations.

sjeng and *bwaves* demonstrate a greater number of alternating downsizing steps between caching levels. In particular, *sjeng* exhibits the order of L2, L3, L2, L1, L3, and L2, in its downsizing decisions, resulting in a total of 6 iterations in

Figure 6.4-(e). Likewise, *bwaves* shows the downsizing order of L2, L1, L2, L1, L2, L3, L2, and L1, in Figure 6.4-(f).

GCD MCR Power Savings Compared to Static Optimal In the bars labeled "MCR", Figure 6.5 shows the relative power savings of the total power consumption of the cache hierarchy normalized to the power savings of the static optimal cache configurations (the other bars in Figure 6.5 will be explained later). Three workloads, *braves, calculix, libquantum*, exhibit more than 90% of the power savings of the static optimal. On the other hand, two workloads, *xalan, h264ref*, demonstrate less than 20% of the power savings of the static optimal. The rest of the workloads exhibit the relative power savings between 39% and 73% (on average 54%).

Cache Power Consumption Breakdown of INT and FP Workloads Table 6.3 shows the cache configurations of the static optimal and our GCD MCR. In the integer (INT) workloads, the optimal cache configurations of our GCD MCR is larger than the cache configuration of the static optimal. In particular, differences in the L1 and L2 cache configurations in the integer workloads are larger than that in floating point (FP) workloads: 6.1 (SO: 2.5, MCR: 8.6) and 2.2 (SO: 2.9, MCR: 5.1) for INT workload, respectively, compared to 3.8 (SO: 5.5, MCR: 9.3) and 0.4 (SO: 3.3, MCR: 3.7). In other words, GCD MCR shows better prediction in FP workloads. As a result, our GCD MCR achieves 65% of the power savings of the static optimal for FP workloads, while achieving only 48% for INT workloads, as shown in Figures 6.7 and 6.8. For the L3 configurations, GCD MCR predicts better



(a) bzip2 (b) mcf

(c) perlbench (d) GemsFDTD

(e) sjeng (f) bwaves

Figure 6.4: GCD multi-level cache resizing solution-search steps.

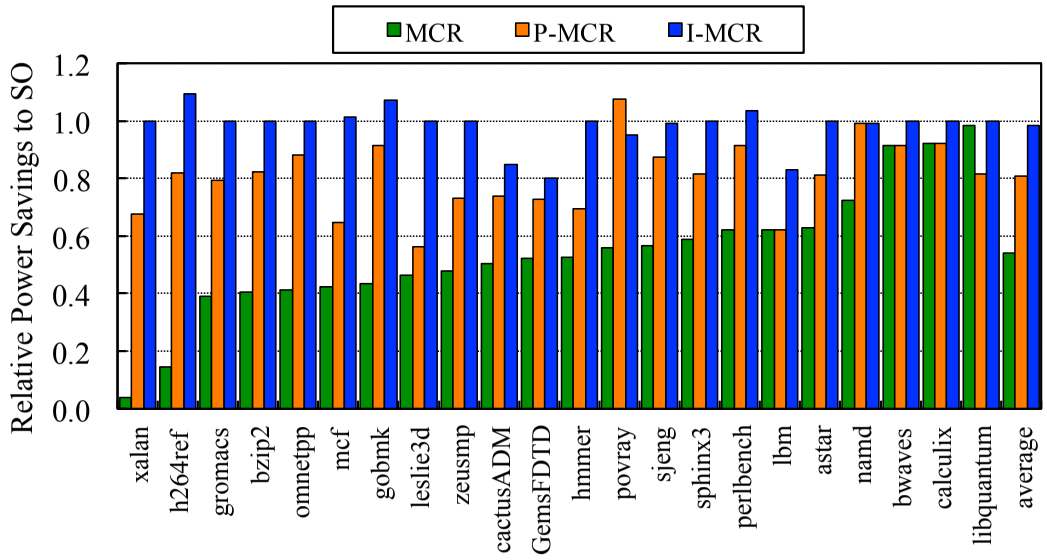


Figure 6.5: Relative power savings of GCD MCR (MCR), GCD MCR with priori information (P-MCR), and ideal GCD MCR (I-MCR) compared to the power savings of static optimal (SO).

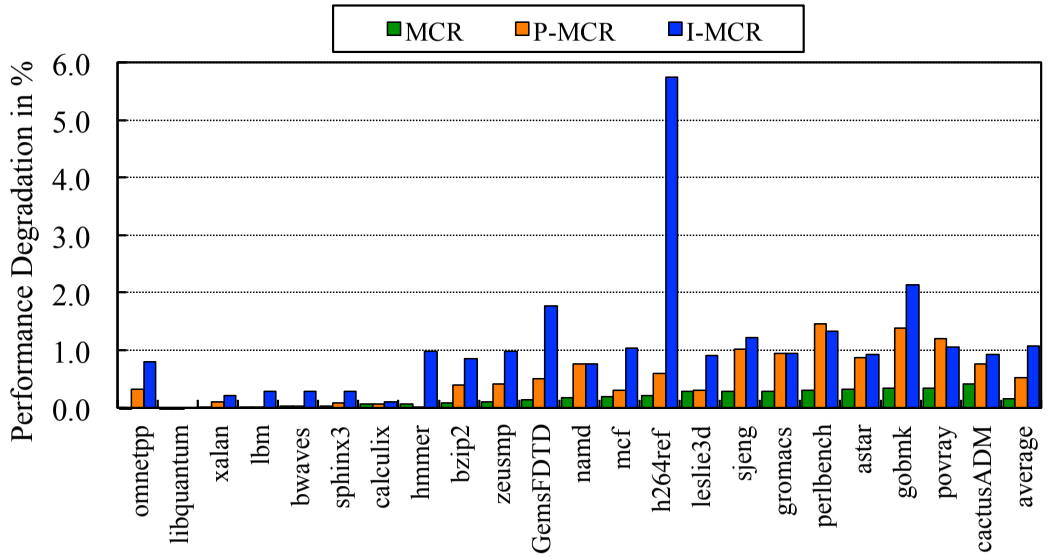


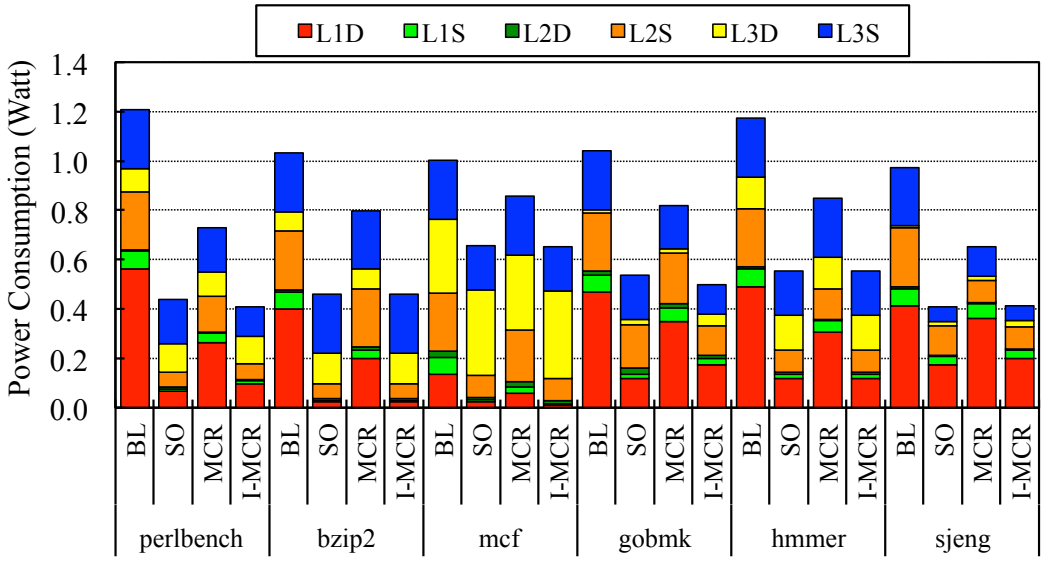
Figure 6.6: Relative performance degradation level in percentile of GCD MCR (MCR), GCD MCR with priori information (P-MCR), and ideal GCD MCR (I-MCR).

in INT workloads than in FP workloads. As shown in Table 6.3, GCD MCR estimates the L3 configuration of 3.2 on average, compared to the L3 configuration of 2.5 of the static optimal. On the other hand, GCD MCR predicts the L3 configuration of 3.0 while the static optimal is 2.2. There errors are directly translated in to higher power consumption in the cache hierarchy. Figures 6.7 and 6.8 show the cache power consumption breakdowns.

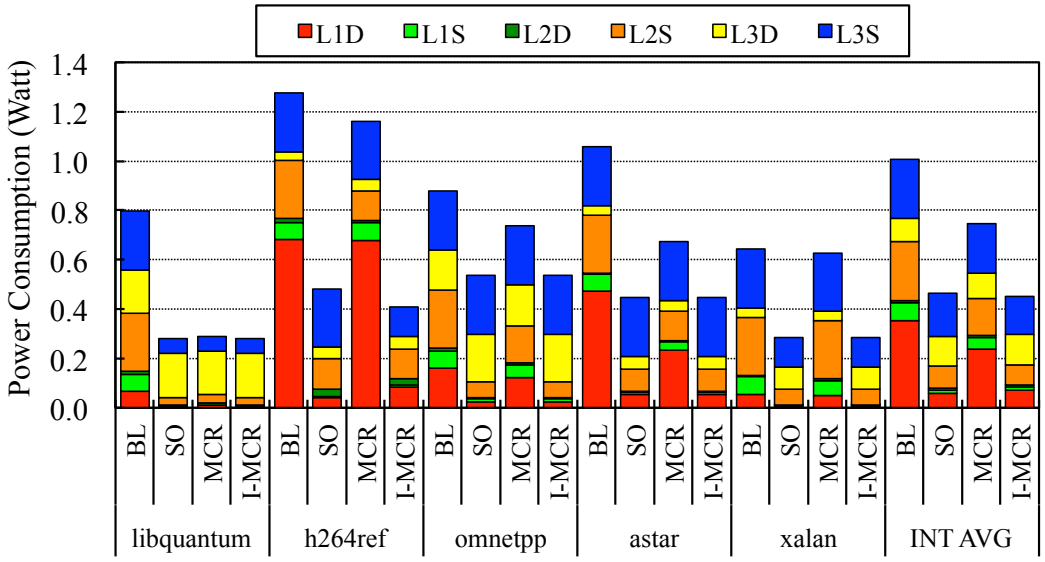
6.3.2 Discussion

Our GCD MCR algorithm is based on prediction to avoid errors and overheads in the actual evaluations. Moreover, we implement our greedy coordinate descent heuristically in the decision and the PEG value comparisons to approximate the global optimum in the solution space. Therefore, it is important to understand what are the sources of errors that cause the suboptimal cache configurations identified by our GCD MCR algorithm.

GCD MCR with a Priori Information There are two main parts in our GCD MCR algorithm in which the errors commence to be accumulated: performance approximation with AMAT and AMAT/power estimation with way counts. As we discussed in Section 5.2, we utilize AMAT to approximate the performance changes by reconfiguring a cache hierarchy. Moreover, we also employ way counts to estimate AMAT/power changes without actual evaluations. First, we present the power savings of a GCD MCR with AMAT of the static optimal as a priori information (P-MCR). We acquire $\Delta AMAT$ from the exhaustive searches and then, provide

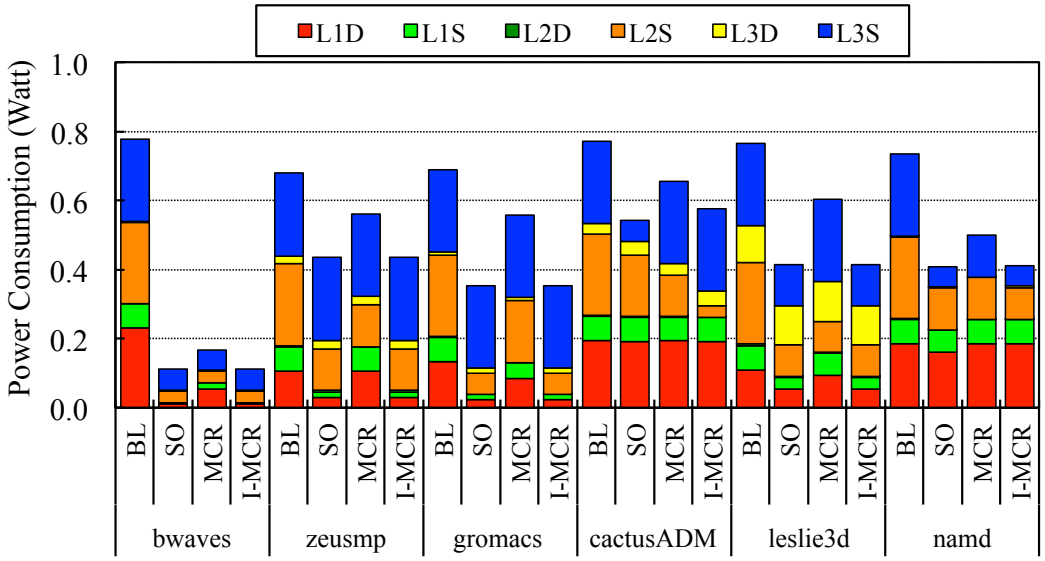


(a) INT-part I

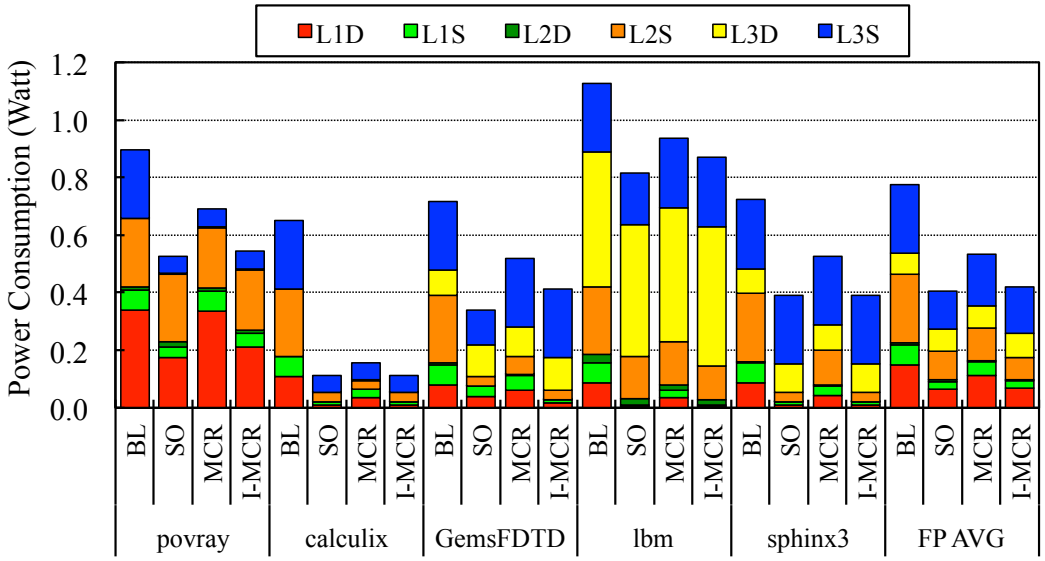


(b) INT-part II

Figure 6.7: Power consumption breakdown comparisons of static optimal (SO) and MCR for INT benchmarks.



(a) FP-part I



(b) FP-part II

Figure 6.8: Power consumption breakdown comparisons of static optimal (SO) and MCR for FP benchmarks.

$\Delta AMAT$ to our GCD MCR algorithm as *balance* in Line 2.7 of Algorithm 2 in Section 6. Figure 6.5 summarizes the power savings of an ideal GCD MCR having AMAT as a priori information. Eliminating errors caused by the part of performance approximation with AMAT improves the power savings to 81% compared to the power savings of 54% using our GCD MCR implementation.

Errors in AMAT and Power Consumption Estimation with Way Counts Second, the AMAT estimation based on way counts is another source of the errors. We conduct a study to understand the errors introduced by way counts based estimations by substituting the approximation logic to the actual power consumption and AMAT information from the exhaustive searches. We present the power savings and performance degradation level of an ideal GCD MCR in Figure 6.5 and 6.6 in the bars labeled "I-MCR". As shown in Figure 6.5, the ideal GCD MCR achieves around 98% of the power savings of the static optimal cache configurations. We can conclude that among the 46% gap comparing the power savings of our GCD MCR implementation and the power savings of static optimum, 27% of the power savings of static optimum is not achievable in our implementation due to the performance approximate with AMAT estimation. Likewise, 17% of the power savings of static optimum is not achievable due to the error in AMAT/power estimation with way counts. Moreover, unachievable 1.7% of the power savings of static optimum is caused by the suboptimization of our greedy coordinate descent.

Non-monotonicity in the Performance Function of AMAT The other notable observation is that *GemsFDTD*, *sjeng*, *h264ref*, *perlbench*, and *gobmk* exhibit worse performance degradation level than 1%. For these workloads, AMAT of ideal GCD MCR is lower than the AMAT of static optimum. However, the IPC of ideal GCD MCR is worse than that of static optimal. The non-monotonicity in the performance function of AMAT occurs for these workloads. In particular, the ideal GCD MCR finds an optimal solution of (2, 4, 2) compared to the optimal cache configuration of (1, 4, 4) for *h264ref*. Although these two configurations result in the similar AMAT, but their impacts on the IPC are different. As shown in Table 6.2, the difference exhibits 5.7% in the IPC. The solution from the ideal GCD MCR has a larger L1 cache and a smaller cache comparing the cache configuration of the static optimum. We observe that cache misses at different caching levels have different impacts on the IPC although they result in the same AMAT. We suspect that there are more chance to tolerate additional memory latency of the L2 cache, 7 cycles, in our out-of-order processor, than the dram latency cached by L3 cache misses, around 300 cycles. Likewise, both cache configurations of the ideal GCD MCR for *perlbench* and *gobmk* have smaller L3 caches and larger L1 caches compared to the cache configurations of static optimum. Similarly, for *sjeng*, the cache configurations of the ideal GCD MCR has smaller L2 cache and larger L1 cache and thus, the additional L3 cache latency degrades the performance despite of the lower AMAT. On the other hands, *GemsFDTD* shows the opposite case: the smaller L1 and the larger L3 result in lower AMAT, but degrades the performance more severely. In this case, both L3 caches suffer from high miss rates of 0.883 and

	AMAT		IPC	
	SO	I-MCR	SO	I-MCR
GemsFDTD	22.172	21.836	0.228	0.226
sjeng	4.416	4.396	0.939	0.935
h264ref	4.714	4.712	1.217	1.159
perlbench	4.454	4.289	1.115	1.114
gobmk	4.35	4.283	1.005	0.995

Table 6.2: AMAT and IPC of static optimal (SO) and ideal GCD MCR (I-MCR).

0.737, for SO and I-MCR, respectively. For the raw miss counts, I-MCR exhibits lower counts of 2,314,094 compared to 2,437,957 of SO. So, we conclude that I-MCR causes more L3 cache hits with the larger L3, but the more misses in the L1 harms the IPC more severely because of the relatively high AMAT of 22.

6.3.3 Online Analysis

Online GCD MCR Power Savings and Performance Degradation Level We compare the power and performance of online GCD MCR to two schemes: static MCR, and the static optimal from the exhaustive search study in Section 6.3.1. Figure 6.9 summaries the power savings and performance degradation level of our online implementation. Online GCD MCR has the benefit of being able to adapt to dynamically changing memory reference characteristics at the cost of reconfiguration overhead. Our online GCD MCR exhibits both benefits and costs. Online GCD MCR saves the total power consumption 13.4% on average while maintaining per-

	L1			L2			L3		
	SO	MCR	I-MCR	SO	MCR	I-MCR	SO	MCR	I-MCR
perlbench	2	8	3	2	5	2	3	3	2
bzip2	1	8	1	2	8	2	4	4	4
mcf	2	6	1	3	7	3	3	4	3
gobmk	4	10	6	6	7	4	3	3	2
hmmer	4	9	4	3	4	3	3	4	3
sjeng	7	11	8	4	3	3	1	2	1
libquantum	1	2	1	1	1	1	1	1	1
h264ref	1	12	2	4	4	4	4	4	2
omnetpp	2	10	2	2	5	2	4	4	4
astar	2	8	2	3	4	3	4	4	4
xalan	1	11	1	2	8	2	2	4	2
AVG	2.5	8.6	2.8	2.9	5.1	2.6	2.9	3.4	2.5
bwaves	1	4	1	1	1	1	1	1	1
zeusmp	4	12	4	4	4	4	4	4	4
gromacs	3	9	3	2	6	2	4	4	4
cactusADM	12	12	12	6	4	1	1	4	4
leslie3d	8	11	8	3	3	3	2	4	2
namd	11	12	12	4	4	3	1	2	1
povray	8	12	9	8	7	7	1	1	1
calculix	2	6	2	1	1	1	1	1	1
GemsFDTD	8	10	3	1	2	1	2	4	4
lbm	1	6	1	5	5	4	3	4	4
sphinx3	2	8	2	1	4	1	4	4	4
AVG	5.5	9.3	5.2	3.3	3.7	2.5	2.2	3.0	2.7
AVG	4.0	9.0	4.0	3.1	4.4	2.6	2.5	3.2	2.6

Table 6.3: Cache configurations of static optimal (SO), GCD MCR (MCR), and ideal GCD MCR (I-MCR).

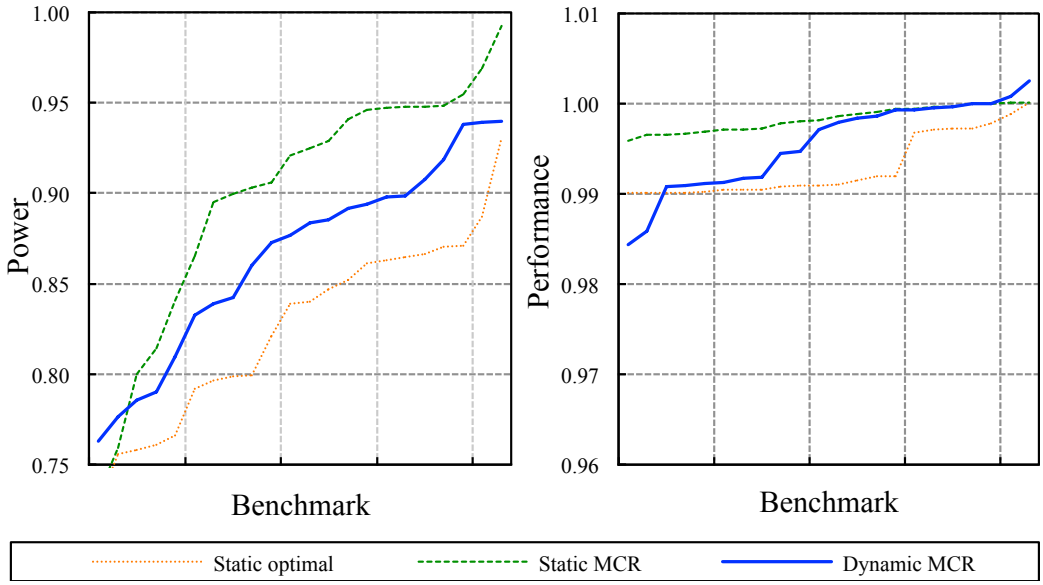


Figure 6.9: Power and performance comparison between dynamic MCR and Nelder-Mead (with static MCR and static optimal).

formance degradation of 0.45% on average. It shows more power savings compared to 10% of static MCR, but two workloads of online GCD MCR shows 1.4% and 1.6% performance degradation level (*gromacs* and *namd*).

Comparison to Static GCD MCR Figure 6.10 shows power savings comparisons between static and online GCD MCR, which are normalized to the power savings of static optimum. The power savings of our online GCD MCR is generally improved compared to the power savings of our static GCD MCR, resulting in 77% of the power savings of static optimum. It is 42% improvement from the power savings of our static GCD MCR. It is also notable that such improvements of *xalan* and *h264ref* are as high as 10X and 5X, respectively. In term of the performance degradation level, results from our online are generally worse than the performance

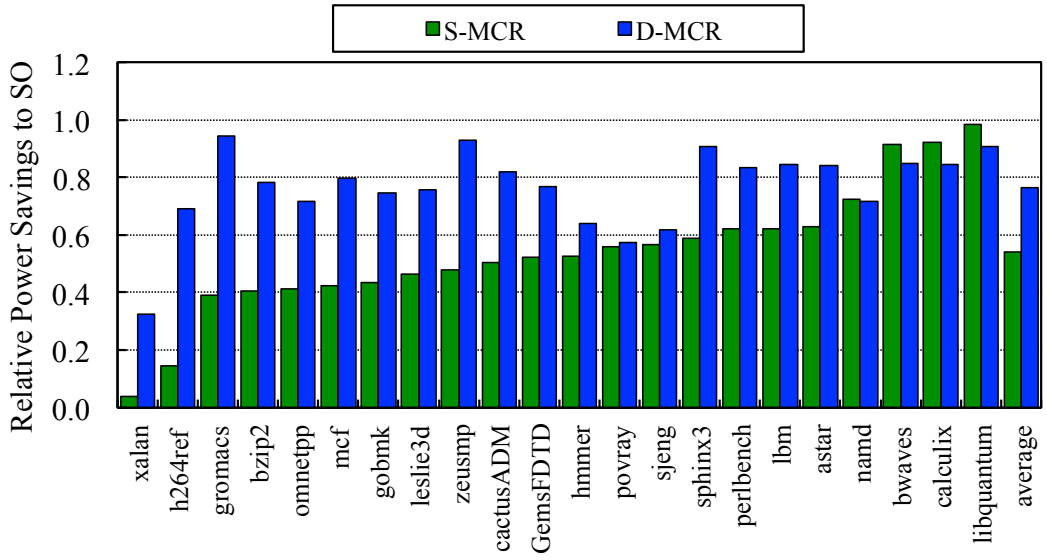


Figure 6.10: Relative power savings of online GCD MCR (D-MCR) and static GCD MCR (S-MCR) compared to the power savings of static optimal (SO).

degradation level of static MCR, but its average is still bound within the performance degradation level of 1%. It is also notable that our online GCD MCR exhibits performance improvement for *h264ref* and *omnetpp*. On the other hands, *grimaces* and *name* exhibit worse performance degradation level than the bound of 1%.

Cache Power Breakdown of Online GCD MCR Figures 6.12 and 6.13 show the cache power consumption breakdown of our online GCD MCR. Throughout the entire workloads, our online implementation achieves more power savings from the dynamic and static power consumptions of L1 cache. On average, online Gcd MCR reduces the power consumption of L1 by 32% compared to the power consumption of static MCR. The bigger power savings from integer workloads are achieved by reducing the power consumptions of 38% and 33% in L1 dynamic and static power consumption, respectively, as shown in Figure 6.12. On the other hands, the L1

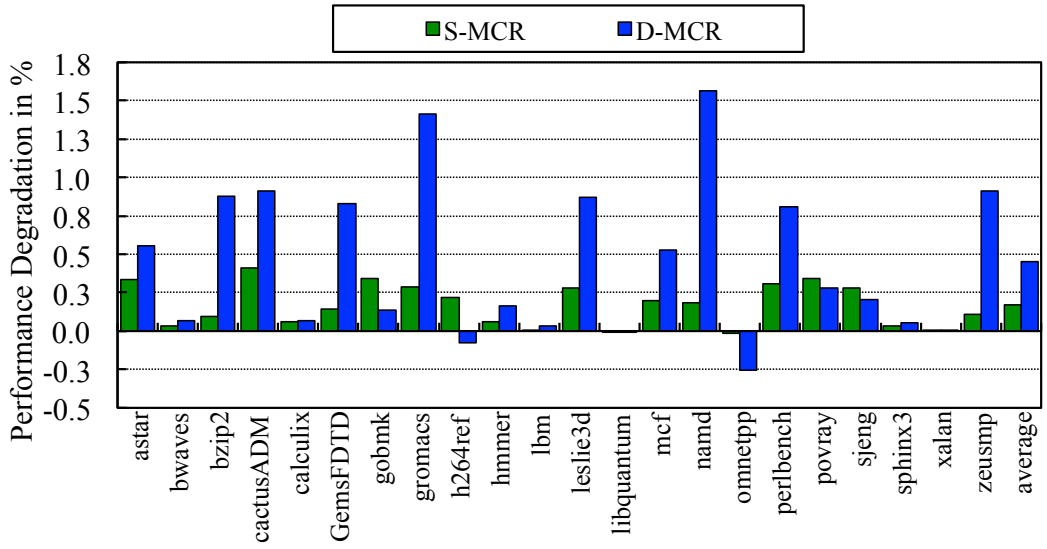
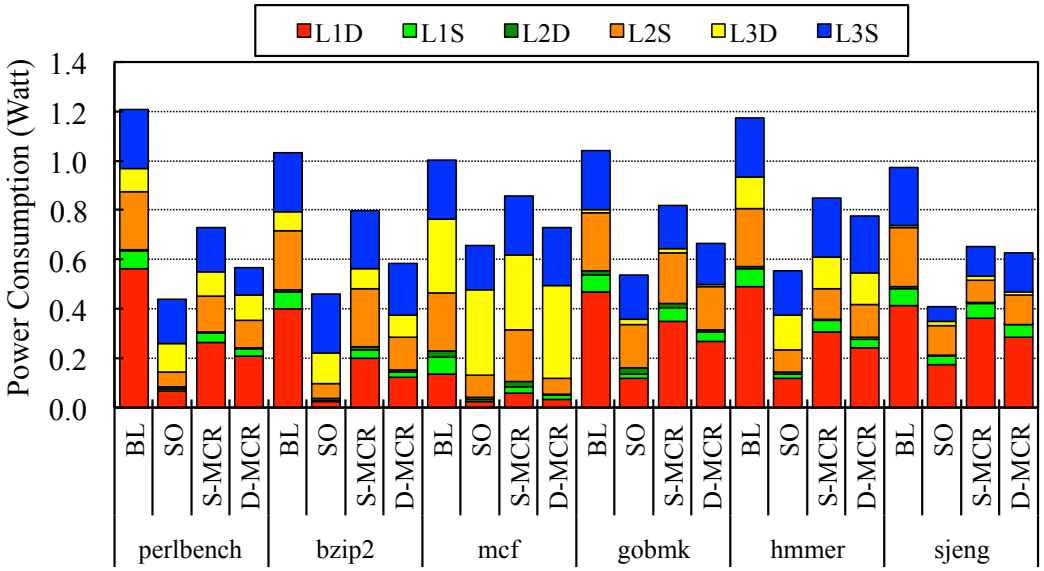
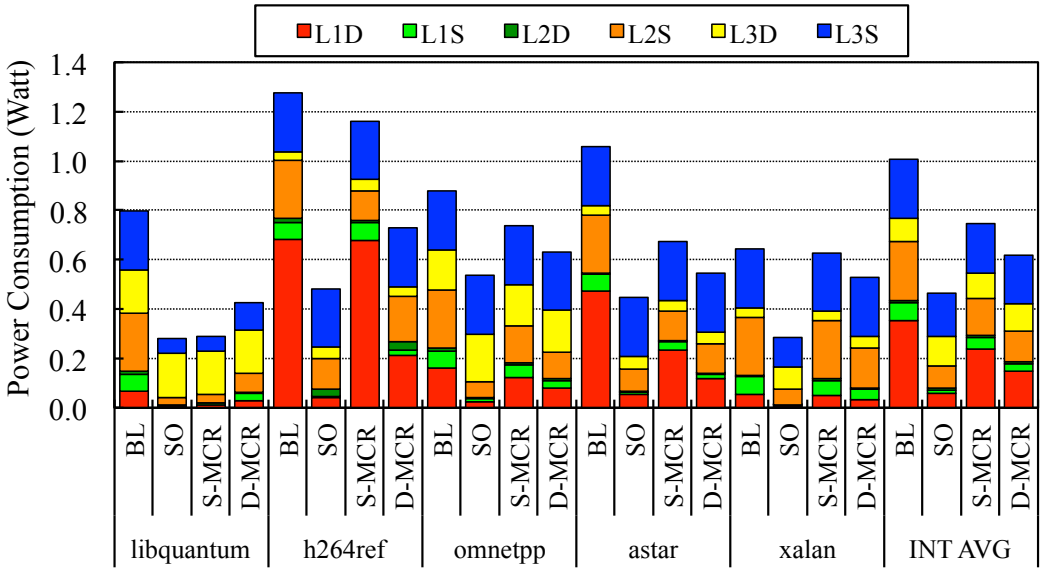


Figure 6.11: Relative performance degradation level in percentile of online GCD MCR (D-MCR) and static GCD MCR (S-MCR) compared to the power savings of static optimal (SO).

power consumption of our online GCD MCR is around 74% and 68% for dynamic and static power, respectively, as shown in Figure 6.13. The power savings of L1 cache is as high as 78% and 81% for *h264ref*, and 67% and 66% for *lbm*, in its dynamic and static power consumption, respectively. Online GCD MCR also further reduces the power consumption of L2 cache by reducing 8% and 19% in dynamic and static power, respectively, on average, compared to the power consumption of L2 of static GCD MCR. However, online GCD MCR shows worse power savings for L3 cache by exhibiting 6% and 1% increase in its dynamic and static power consumption, respectively. For *name*, our online implementation consumes 2.8X and 1.5X compared to the L3 dynamic and static power consumption, respectively.

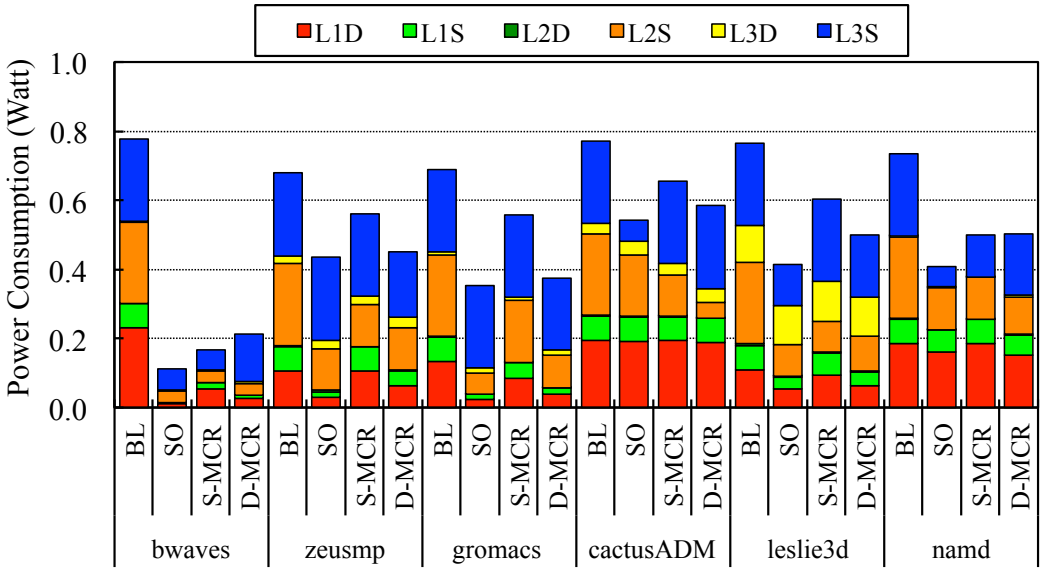


(a) INT-part I

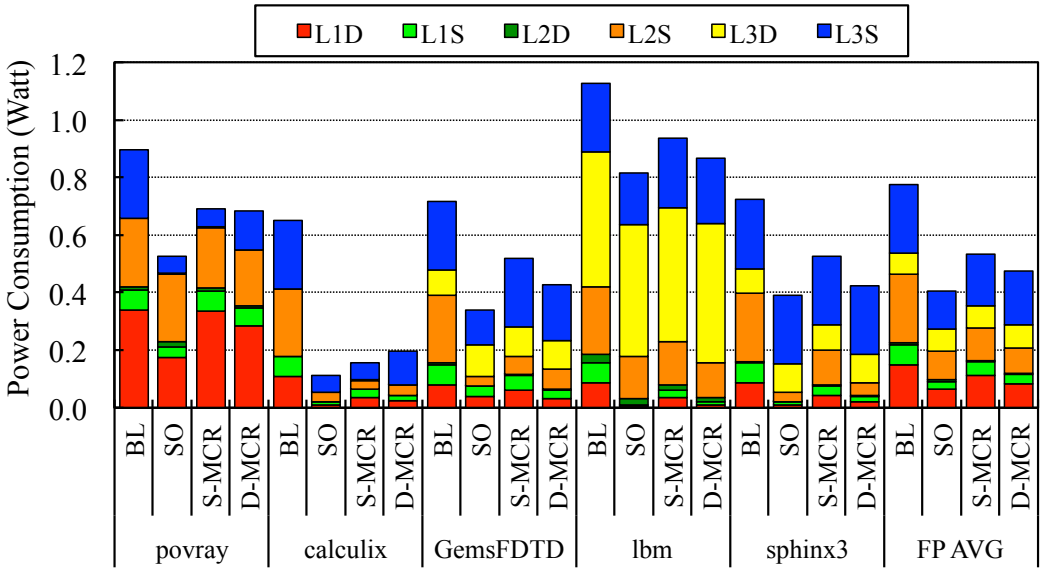


(b) INT-part II

Figure 6.12: Power consumption breakdown comparisons of online GCD MCR for INT benchmarks.



(a) FP-part I



(b) FP-part II

Figure 6.13: Power consumption breakdown comparisons of online GCD MCR for FP benchmarks.

Search Movement and Search Iterations Table 6.4 summarizes the reconfiguration numbers per workload. On average, there were 2.1, 1.8, and 0.6 reconfigurations in L1, L2, and L3 cache, respectively (there are 32 new searches according to the reset frequency and these number are divided by this number of new searches from the numbers shown in Table 6.4), for having 32 search windows per simulation (4,000 epochs per run and reset frequency is every 125 epochs). The iteration numbers are 4.4 on average, sum of reconfiguration number of each caching level, is comparable to the number of 3.2 of static MCR which was shown in Figure 6.2 of Section 6.3.1. Despite of transient behaviors and dynamically changing way counts, our GCD MCR shows its fast convergence rate by showing little difference in the iteration numbers of static and online GCD MCR.

6.3.4 Discussion

Correlation Study We conduct a statistical analysis to understand the relationship between the improvement of power savings and other parameters, which in our online implementation compare to the power savings in the static study. First, we calculate the improvement ratio of power savings of online scheme to power savings of static scheme. Then, we collect parameters as candidates to test the correlations. We use IPC, AMAT, and L1/L2/L3/Total cache reconfigurations numbers in this study. Table 6.5 shows parameters used in this test. Likewise, we conduct another statistical analysis to understand the relationship between the performance degradation levels and these parameters. Table 6.6 shows parameters used in this test as

WL	Cache Reconfig.#				WL	Cache Reconfig. #			
	L1	L2	L3	Total		L1	L2	L3	Total
perlbench	52	61	31	144	bwaves	15	8	58	81
bzip2	90	70	13	173	zeusmp	99	68	25	192
mcf	91	82	4	177	gromacs	48	99	26	173
gobmk	96	52	36	184	cactusADM	6	58	0	64
hmmer	79	47	11	137	leslie3d	63	55	15	133
sjeng	72	60	36	168	namd	70	73	34	177
libquantum	24	21	21	66	povray	47	44	26	117
h264ref	120	41	0	161	calculix	37	33	40	110
omnetpp	97	75	0	172	GemsFDTD	67	70	31	168
astar	72	67	0	139	lbm	67	5	16	88
xalan	91	83	2	176	sphinx3	45	70	0	115
INT AVG	80	60	14	154	FP AVG	51	53	25	129
WL AVG	66	56	19	142					

Table 6.4: Number of cache reconfigurations of GCD MCR

well.

Table 6.7 shows the correlation coefficients and their statistical significances. The L1-cache reconfiguration numbers are significantly correlated with the power savings improvement in our online implementation compared to the static study. It is notable that the L3-cache reconfiguration numbers are significantly correlated with the power savings improvement, reversely, having a negative correlation coefficient of -0.4. Their p-values are 0.063 and 0.0653, respectively. As such, we conclude that the frequent L1 cache resizing improved the power savings in the L1 cache by capturing dynamically changing memory-access characteristics in our workloads. However, online GCD MCR exhibits increased power consumptions because of the frequent L3 cache resizing being enacted adversely.

Epoch Sizes and Glitches from Transient Errors and Search Resets There are three workloads that underperform compared to the results in the static study: *libquantum*, *calculix*, and *bwaves*. Figure 6.14 shows the traces of dynamic cache reconfigurations in *libquantum* workload. It seems like that there are program phase changes during around the epoch number of 1,000 and 3,000. However, it turned out that the way counts during the period do not provide enough resolution to search an optimal for being low counts due to short epoch cycles. *libquantum* exhibits the longest AMAT among the entire workloads, 70.7 cycles. Due to the longest AMAT, this workload requires a longer epoch size in cycles to accumulate reasonable amount of cache hits, and thus way counts. For being failing in such accumulating way counts, our online GCD MCR fails to achieve the power savings as high as of the

WL	Static	Online	Ratio	IPC	AMAT	Number of Reconfigurations			
						L1	L2	L3	Total
xalan	0.040	0.403	9.983	0.124	8.014	91	83	2	176
h264ref	0.144	0.695	4.819	1.229	4.248	120	41	0	161
gromacs	0.391	0.943	2.414	0.858	4.498	48	99	26	173
bzip2	0.406	0.814	2.003	0.982	5.152	90	70	13	173
mcf	0.423	0.823	1.943	0.195	43.500	91	82	4	177
zeusmp	0.478	0.893	1.865	0.338	7.569	99	68	25	192
cactusADM	0.506	0.907	1.793	0.296	5.287	6	58	0	64
omnetpp	0.414	0.735	1.777	0.293	17.262	97	75	0	172
gobmk	0.435	0.735	1.691	1.015	4.181	96	52	36	184
leslie3d	0.464	0.768	1.655	0.315	13.591	63	55	15	133
GemsFDTD	0.524	0.817	1.561	0.230	21.454	67	70	31	168
sphinx3	0.590	0.907	1.538	0.376	15.597	45	70	0	115
perlbench	0.620	0.840	1.356	1.126	4.187	52	61	31	144
astar	0.628	0.844	1.344	1.002	4.462	72	67	0	139
lbm	0.623	0.811	1.302	0.319	17.575	67	5	16	88
hmmer	0.526	0.656	1.246	1.261	4.025	79	47	11	137
sjeng	0.568	0.617	1.087	0.947	4.282	72	60	36	168
povray	0.559	0.568	1.017	0.643	4.264	47	44	26	117
namd	0.725	0.719	0.992	0.581	4.240	70	73	34	177
bwaves	0.915	0.847	0.925	0.687	4.014	15	8	58	81
calculix	0.921	0.844	0.917	0.528	4.015	37	33	40	110
libquantum	0.982	0.717	0.730	0.227	70.748	24	21	21	66
average	0.540	0.768	1.998	0.617	12.371	66	56	19	142

Table 6.5: Parameters to test the correlation of the improvement of the power savings in online GCD MCR.

WL	Static	Online	Ratio	IPC	AMAT	Number of Reconfigurations			
						L1	L2	L3	Total
h264ref	0.0022	-0.0008	-0.365	1.229	4.248	120	41	0	161
gobmk	0.0034	0.0014	0.403	1.015	4.181	96	52	36	184
libquantum	-0.0001	0.0000	0.688	0.227	70.748	24	21	21	66
sjeng	0.0028	0.0021	0.728	0.947	4.282	72	60	36	168
povray	0.0034	0.0028	0.824	0.643	4.264	47	44	26	117
calculix	0.0006	0.0007	1.100	0.528	4.015	37	33	40	110
sphinx3	0.0003	0.0005	1.546	0.376	15.597	45	70	0	115
astar	0.0033	0.0055	1.659	1.002	4.462	72	67	0	139
bwaves	0.0003	0.0007	2.077	0.687	4.014	15	8	58	81
cactusADM	0.0041	0.0092	2.227	0.296	5.287	6	58	0	64
perlbench	0.0031	0.0081	2.648	1.126	4.187	52	61	31	144
mcf	0.0020	0.0053	2.671	0.195	43.500	91	82	4	177
hmmer	0.0006	0.0017	2.764	1.261	4.025	79	47	11	137
leslie3d	0.0028	0.0087	3.131	0.315	13.591	63	55	15	133
gromacs	0.0029	0.0141	4.950	0.858	4.498	48	99	26	173
GemsFDTD	0.0014	0.0083	5.849	0.230	21.454	67	70	31	168
xalan	0.0000	0.0000	6.610	0.124	8.014	91	83	2	176
zeusmp	0.0011	0.0091	8.265	0.338	7.569	99	68	25	192
namd	0.0019	0.0157	8.420	0.581	4.240	70	73	34	177
bzip2	0.0009	0.0088	9.368	0.982	5.152	90	70	13	173
omnetpp	-0.0001	-0.0026	20.777	0.293	17.262	97	75	0	172
lbm	0.0000	0.0003	35.296	0.319	17.575	67	5	16	88
average	0.0017	0.0045	5.529	0.617	12.371	66	56	19	142

Table 6.6: Parameters to test the correlation of the performance degradation levels in online GCD MCR.

	IPC	AMAT	Number of Reconfiguration			
			L1	L2	L3	Total
Power	-0.148	-0.128	0.403*	0.321	-0.400*	0.324
Performance	-0.318	0.0417	0.206	-0.164	-0.156	-0.0109

Table 6.7: Correlation coefficient of each metric. * indicates statistical significance at 10% level.

power savings in the static study.

Figure 6.15 shows the traces in *bwaves* workloads. Unlike the *libquantum*, *bwaves* has a relatively short AMAT of 4 cycles. However, this workload exhibits the oscillating patterns in the L3 cache reconfigurations. In this case, the L3 cache's way counts appear in marginal counts so that it's PEG value can be translated into the different sizes according to slight changes in the way counts. As a result, *bwaves* exhibits the largest number of the L3 cache reconfigurations. Such transient errors can be enacted online and results in inferior power savings compared to the static study.

Likewise, *calculix* shows the second largest number in the L3 cache reconfigurations. Moreover, *calculix* also exhibits several glitches in the L2 cache reconfigurations. In particular, these glitches in the L2 cache reconfigurations follow the reset frequency in our online GCD MCR. After the reset frequency, marginal way counts are prone to result in the different result than the result from the static study or the average cache size of adjacent epochs. As a result–having both transient errors

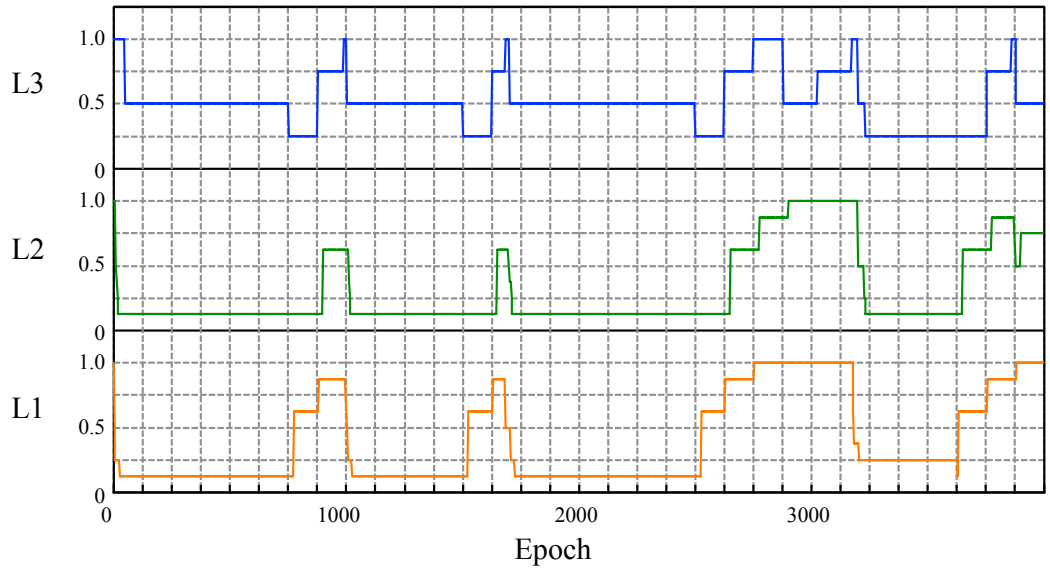


Figure 6.14: Dynamic cache reconfigurations of online GCD MCR in *libquantum* workload.

in the L3 caches and glitches in the L2 caches–, *calculix* exhibits the second worst power-savings improvement compared to the power savings of the static study.

Dynamic Adaptation in Online GCD MCR On the other hands, our online implementation generally increase the power savings by capturing dynamically changing memory-access characteristics. In particular, *xalan* exhibits 10X improvement in the power savings by achieving 40% of the power savings of the static optimal, from 4% in the static study. Figure 6.17 shows the traces of the dynamic cache reconfigurations in *xalan* workload. The capacities of the L1 and the L2 caches are dynamically changing in alternative patterns: smaller L1 and larger L2 or *vice versa*. As a result, the online GCD MCR achieves significant power savings while maintaining the performance degradation level.

Similarly, *h264ref* achieves a notable improvement of 5X in the power savings

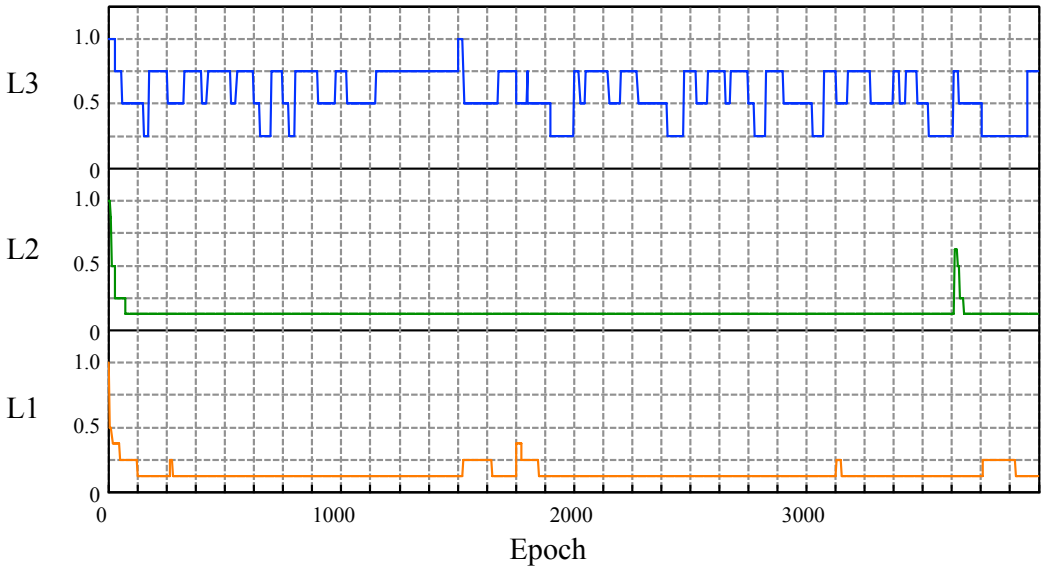


Figure 6.15: Dynamic cache reconfigurations of online GCD MCR in *bwaves* workload.

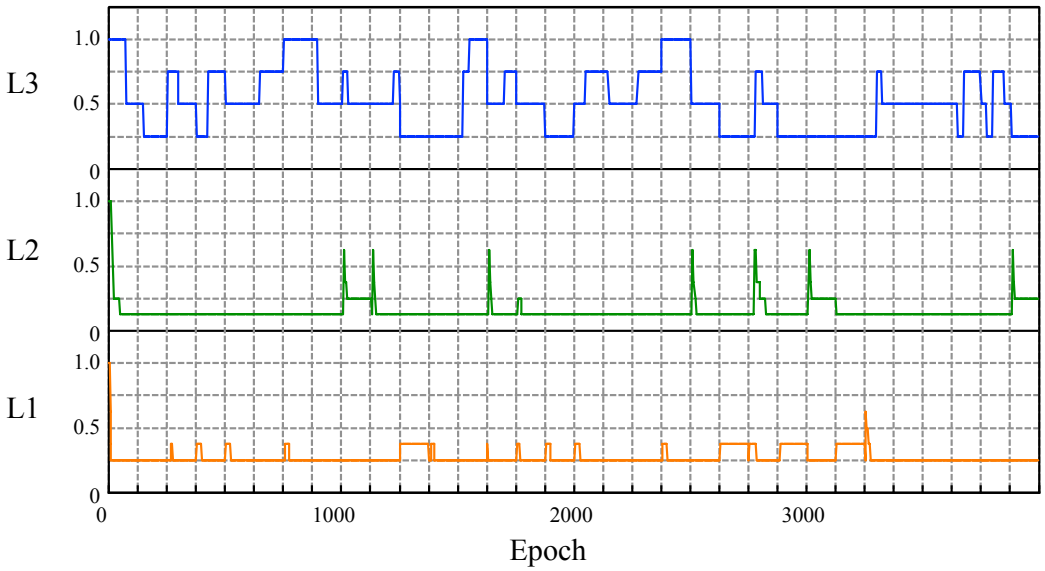


Figure 6.16: Dynamic cache reconfigurations of online GCD MCR in *calculix* workload.

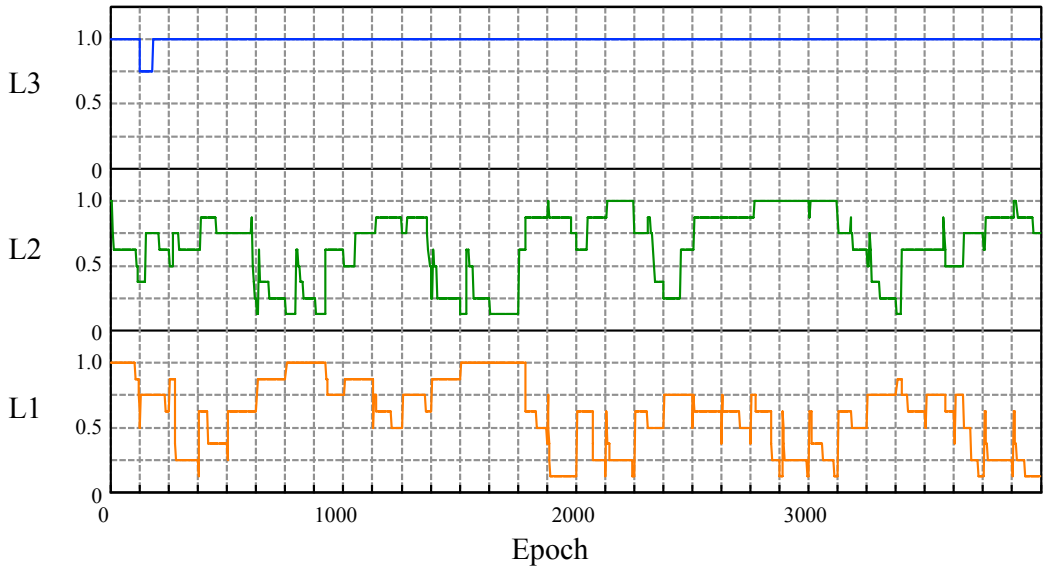


Figure 6.17: Dynamic cache reconfigurations of online GCD MCR in *xalan* workload.

compared to the power savings in the static study. As shown in Figure 6.18, *h264ref* exhibits the largest number in the L1 cache reconfigurations, 120 times. Despite of the significant number of glitches are visible in the L1 cache reconfigurations, the power-savings improvement shows that *h264ref*'s dynamic adaptation results in better power savings even with a slight performance improvement as we discussed earlier. In particular, *h264ref* exhibits dynamic adaptation in the L2 cache reconfiguration without noticeable glitches.

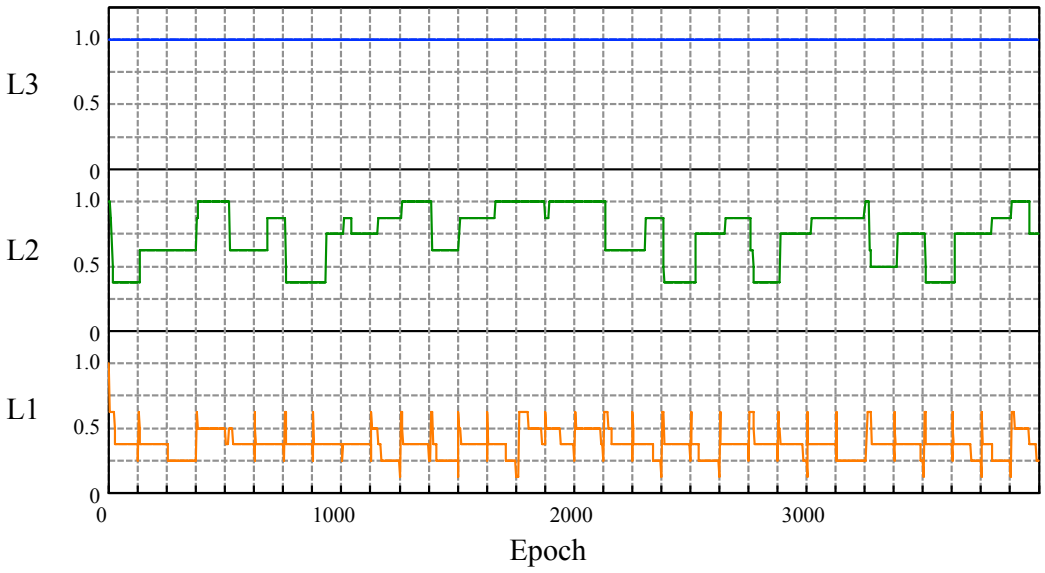


Figure 6.18: Dynamic cache reconfigurations of online GCD MCR in *h264ref* workload.

Chapter 7

PEG-based Last Level Cache Partitioning

In Chapter 6, we have studied the greedy coordinate descent multi-level cache resizing method for a uniprocessor, and evaluated its power savings and performance degradation. In this chapter, we first discuss the cache partitioning problem for shared last level caches (LLC) in CMPs. And then, we will conduct a limit study to understand the peak performance and power savings with a performance degradation constraint that we can expect from cache partitioning. After that, we discuss our PEG-based cache partitioning algorithm and present power savings and performance degradation results.

7.1 Last Level Cache Partitioning Problem

Modern CMPs are commonly equipped with a shared LLC to efficiently utilize cache capacity and share data between cores. For multiprogrammed workloads, partitioning a shared LLC amongst the cores is an effective technique to avoid cache interference between cores so that one can efficiently utilize the shared capacity. Ideally, finding optimal private-cache sizes and partition sizes in the shared LLC leads to saving most of the wasteful power consumption in the CMP on-chip cache hierarchy without noticeable performance degradation. However, we can not apply our previous uniprocessor GCD multi-level cache resizing technique directly to

the CMP multi-level cache resizing problem to achieve power savings because the cache partitioning problem is somewhat different from the cache resizing problem. Specifically, there are two major differences.

First, the cache partitioning problem is constrained by the total capacity in the shared LLC. In uniprocessor GCD multi-level cache resizing, cache capacities at different caching levels are independent of each other. The only way different caches are coupled is in how they affect performance. However, in the cache partitioning problem, the sum of all partitions is strictly limited to the total size of the LLC. In other words, all variables in a cache partitioning problem are coupled due to the maximum capacity of the LLC.

Second, cache partitions not only change the power consumption in an LLC, but also determine the CMP performance. More importantly, the optimal cache partitioning can improve the CMP performance significantly. There has been a large body of research in cache partitioning, and most of these efforts improve the overall performance compared to some baseline (usually even partitioning, or EP). As we discussed earlier, the uniprocessor GCD MCR mostly decreases the performance by down sizing caches, and we limit the maximum performance degradation. As such, to apply the GCD approach similarly in a cache partitioning problem, we need to know the maximum performance level of a given CMP with an LLC partitioning technique to define a performance degradation level.

In addition to these differences, the cache partitioning problem also has a strong motivation in the perspective of power efficiency. Although there have been many performance-oriented online LLC partitioning methods, only a few exist to

improve power efficiency. A shared LLC can be considered as a single cache, so we can apply a smiler approach in the GCD multi-level cache resizing by considering PEG. A recent study [10] (COOP) applies a cache resizing technique to a shared LLC in CMPs on top of the utility-based cache partitioning scheme [11]. However, this approach still does not consider the power efficiency gain as we do in the uniprocessor multi-level cache resizing.

For these reasons, we begin by discussing our PEG-based LLC partitioning method in this chapter as the first step towards solving the multi-level cache resizing problem for CMPs.

7.2 Experimental Methodology

7.2.1 Simulator

We modify our SimpleScalar simulator, described in Section 4.3.2, to support multiprogrammed workloads. Since we implement way-aligned partitioning, we develop a multi-process simulator consisting of concurrent processes. Each process simulates one of the cores in CMP, and there is one additional master process controlling and synchronizing the concurrent simulation processes. The master process also orchestrates each simulation process so that each core can receive the designated LLC partition size. It is assigned according to an LLC partitioning policy. We simulate communication channels between cores by utilizing interprocess communication and implement it via the Linux socket library.

The master process simulates our GCD CMP multi-level cache resizing al-

gorithm and the master process and other concurrent processes communicate with each other to send/receive way counts and reconfiguration commands at every epoch. The total computation/execution time in the master process is either added to the cycle counts of slave processes, or considered as concurrent computation in the extra hardware according to the implementation methods of our algorithms. Note that we account for the computational overheads in the total execution time by adding measured cycle counts obtained via running the algorithm in the simulated core for the case of software implementation.

7.2.2 Metrics

Measuring performance in multi-program workloads is a non-trivial process. There are several metrics to quantify the performance of a system which runs multiple programs concurrently. We discuss performance improvement with respect to three metrics which are popular in systems research: weighted speedup, harmonic mean of weighted IPCs, and total system power consumption. Weighted speedup will be used to quantify system throughput and harmonic mean of normalized IPC will be considered as a system fairness metric as it measures average turn-around-time increases [65].

$$\textit{Weighted Speedup} = \sum_{i=1}^{N} \frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{7.1}$$

$$\textit{Harmonic Mean of Weighted IPCs} = \frac{N}{\sum\limits_{i=1}^{N} \frac{IPC_i^{SP}}{IPC_i^{MP}}} \tag{7.2}$$

$$Total\ System\ Power = \sum_{i=1}^{N} Pwr_i^{MP} \tag{7.3}$$

7.2.3 Multiprogrammed Workloads

We generate multi-program workloads randomly to mix all three categories in Table 4.2. We created 20 workloads each for 2-, 4- and 8-core CMPs, resulting in a total of 60 workloads. Table 7.1 shows all combinations of our multiprogrammed workloads.

7.3 Limits of Last Level Cache Partitioning

The performance limit of last level cache partitioning will show the ideal peak performance. Finding the optimal cache partitions is an *NP-hard* problem [22] and thus it requires an exhaustive search over the solution space. Because of this complexity of the cache partitioning problem, we need an online algorithm to approximate an optimal partitioning. As such, we first need to understand the gap between the ideal peak performance and the online performance level that we will take as local optimal at run time. In particular, we use the utility-based cache partitioning (UCP) scheme [11] to implement an online algorithm to approximate the global optimal at runtime. One of the main reasons of picking UCP as an online algorithm to approximate an ideal partitioning is that UCP does not require additional extra hardware on top of the hardware we need for GCD multi-level cache resizing, and it is considered one of the state-of-the-art cache partitioning techniques.

Workload	Benchmarks (2, 4 and 8 Cores)		
G2/4/8-A	$m_{\{0,2\}}$	$l_{\{0,1\}}, m_{\{2,3\}}$	$l_{\{3,6\}}, m_{\{4,5,6\}}, h_{\{1,3,4\}}$
G2/4/8-B	$m_{\{0,6\}}$	$l_{\{5,6,9\}}, m_6$	$l_{\{1,2,5,7\}}, m_{\{1,4,4\}}, h_2$
G2/4/8-C	l_0, m_0	$l_2, m_{\{1,4\}}, h_2$	$l_{\{0,1,3,3,9\}}, m_{\{0,2,6\}}$
G2/4/8-D	m_0, h_0	$l_9, m_5, h_{\{0,1\}}$	$l_{\{1,3,6\}}, h_{\{0,0,1,2,2\}}$
G2/4/8-E	$l_{\{4,6\}}$	$l_1, m_{\{4,6\}}, h_0$	$l_{\{0,3,4,6\}}, m_{\{1,4,6\}}, h_3$
G2/4/8-F	l_5, m_6	$l_{\{5,9\}}, m_6, h_1$	$l_{\{1,5\}}, m_{\{0,2,4,4\}}, h_{\{3,4\}}$
G2/4/8-G	l_5, h_0	$l_{\{1,8,9\}}, m_2$	$l_{\{1,3,6,7,7,9\}}, m_1, h_2$
G2/4/8-H	l_5, m_5	$l_3, m_6, h_{\{1,2\}}$	$l_3, m_{\{1,3,5,5,6\}}, h_{\{0,2\}}$
G2/4/8-I	$l_{\{2,6\}}$	$l_{\{2,3,4\}}, h_0$	$l_{\{4,4\}}, m_{\{1,4,5\}}, h_{\{0,1,3\}}$
G2/4/8-J	l_6, m_1	$l_{\{0,3,7\}}, h_0$	$l_{\{3,7\}}, m_{\{3,4,6,6\}}, h_{\{1,4\}}$
G2/4/8-K	$l_{\{3,6\}}$	$l_{\{0,6\}}, m_{\{1,6\}}$	$l_{\{0,4,5,6,7,9\}}, m_{\{3,4\}}$
G2/4/8-L	m_2, h_1	$l_{\{3,4\}}, m_{\{2,5\}}$	$l_4, m_{\{1,1,3,4,4,5,6\}}$
G2/4/8-M	l_2, h_1	$l_{\{0,5\}}, m_0, h_0$	$l_{\{0,0,1,2\}}, m_{\{3,5,5\}}, h_3$
G2/4/8-N	m_1, h_1	$l_{\{1,6,8\}}, m_2$	$l_{\{2,4,5\}}, m_{\{2,3,4\}}, h_{\{0,2\}}$
G2/4/8-O	m_5, h_1	$l_{\{1,3,6\}}, m_2$	$l_{\{1,1\}}, m_{\{1,4,4\}}, h_{\{1,1,2\}}$
G2/4/8-P	l_9, m_2	$l_{\{4,6\}}, m_4, h_4$	$l_{\{5,7,7\}}, m_{\{4,4\}}, h_{\{0,3,4\}}$
G2/4/8-Q	$l_{\{0,9\}}$	$m_{\{0,3\}}, h_{\{1,2\}}$	$l_{\{2,4,6\}}, m_6, h_{\{0,0,1,3\}}$
G2/4/8-R	$m_{\{3,4\}}$	$l_{\{5,6,9\}}, h_2$	$l_{\{1,5\}}, m_{\{1,2,2,2,4\}}, h_2$
G2/4/8-S	$m_{\{3,6\}}$	$l_{\{0,1,5,7\}}$	$l_{\{2,7\}}, m_{\{2,5,6\}}, h_{\{2,3,4\}}$
G2/4/8-T	$l_{\{1,2\}}$	$l_{\{6,8\}}, m_5, h_6$	$l_{\{1,4,5\}}, m_{\{2,3,5\}}, h_{\{0,1\}}$

Table 7.1: Multi-program workloads. Each benchmark is shown in abbreviations according to the benchmark classification based on MPKI in Chapter 4. Benchmark classifications are shown in Table 4.2. For example, G2-A workload consists of m_0 and m_2 and these are *bzip2* and *zeusmp* as shown in Table 4.2, respectively.

Exhaustive Search and Runtime Maximum Power Savings with 1% Performance Degradation To conduct this limit study, we utilize our extensive simulation results from Section 4.4 to find the limit in performance of cache partitioning by searching those results exhaustively (we label this search as E). Moreover, we also search the limit in last level cache resizing as we did in the uniprocessor MCR limit study in Section 4.4: we set a maximum performance degradation level of 1% based on the performance level that UCP achieves (we label this search as U+PD). This limit study in a last level cache resizing will show the power savings goal that our PEG-based cache partitioning algorithm may potentially achieve.

Performance Limit of Cache Partitioning Figure 7.1 summarizes the result of these limit studies (Figure 7.1 also shows results for our technique which we will discuss later). Note that all values are normalized to the result of even partitioning (EP). On average, the exhausitive-search-based performance-oriented optimal partitioning (E) improves the system throughput by 2.8% and also increases the total system power consumption by 0.6%. On the other hand, UCP (U), an online scheme to find a performance-oriented optimal partitioning, improves the system throughput by 2.2% by achieving almost 81% of the throughput improvement of the optimal partitioning, and increases the total system power consumption by 0.3%. The limit of power savings with 1% performance degradation, if we take the LLC partition of UCP as the reference, is around 3% as the U+PD line in Figure 7.1 shows. Moreover, U+PD achieves a system throughput improvement of 1.8%, so this is within the 1% performance degradation compared to UCP.

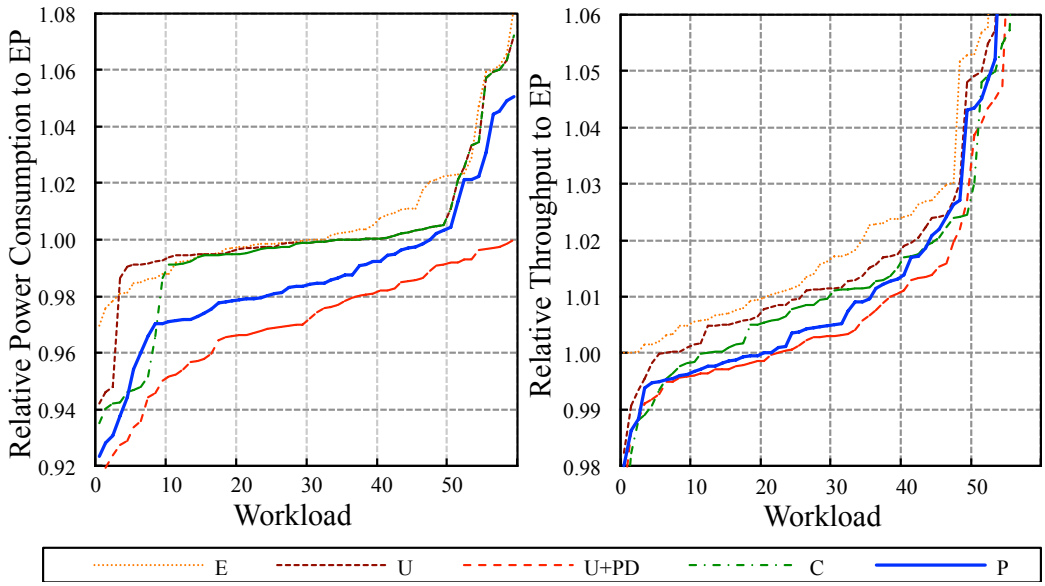


Figure 7.1: Power and system throughput comparison between exhaustive search to find the optimal partitioning for the best throughput (E), UCP (U), exhaustive LLC resizing search to find the optimal partitioning for the best power savings based on UCP with the performance degradation level of 1% (U+PD), COOP (C), and our PEG-based cache partitioning (P). Power and system throughput are normalized to the results of even partitioning (EP).

Power Savings Limit in Cache Partitioning We compare results for the exhaustive search (EXH) and best power savings with the performance degradation level of 1% (U+PD) against the result for UCP because we employ cache partitions of UCP as the reference to define performance degradation at runtime. This comparison will show us the power savings goals for our on-line techniques that we will present in Section 7.5. In particular, Figure 7.2 shows power consumption breakdown per core for a few representative examples in our limit study. For example, G2-C/G/R/T shows the relative total power consumption of 97%, 89%, 82%, and 80%, respectively, compared to the total system power consumption of UCP. This limit study is only limited to the LLC partitioning and hence, most of the power savings comes from the power consumption of LLC dynamic and static power consumption, and dynamic power consumption of DRAM. Comparing these power consumption only, for example, G2-C/G/R/T exhibit 92%, 76%, 54%, and 44% savings compared to the power consumption of UCP, respectively. Figure 7.3 summarizes such power consumption across the entire 60 workloads for this study. The limit study in the LLC resizing with the performance degradation level of 1% shows that up to 63% power savings can be achieved, compared to the LLC and DRAM power consumptions of UCP. On average, the power savings across all the workloads is 16%.

Global Optimum vs. Local Optimum As we pointed out earlier in Section 7.1, COOP finds power-efficient cache partitions based on utility, and follows the same sequence that the UCP algorithm generates. This approach is prone to end up with

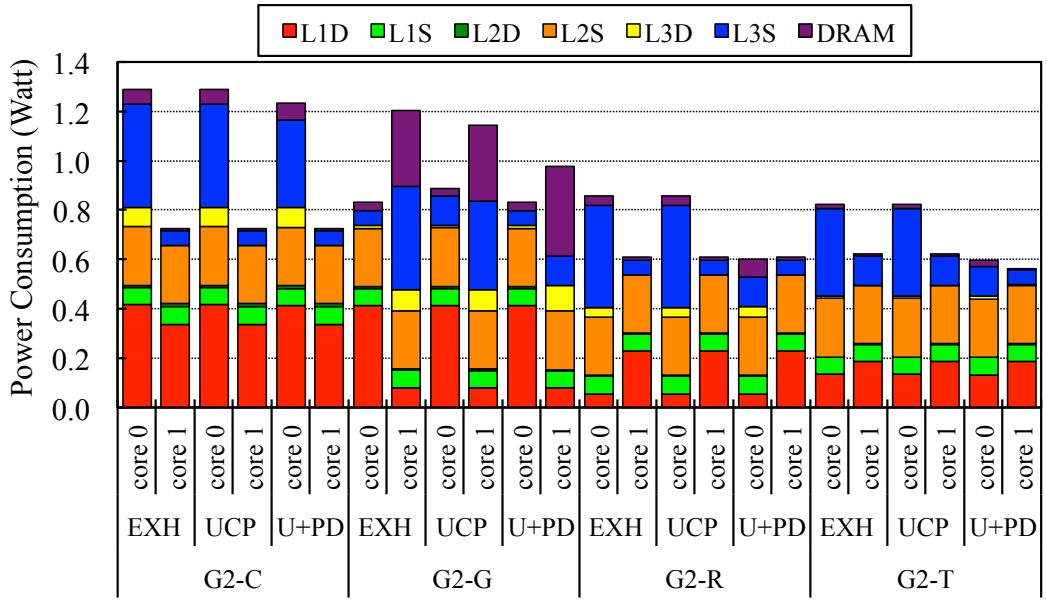


Figure 7.2: Power consumption breakdown for performance-oriented optimal (EXH), UCP, and exhaustive LLC resizing search to find the optimal partitioning for the best power savings with the performance degradation level of 1%(U+PD).

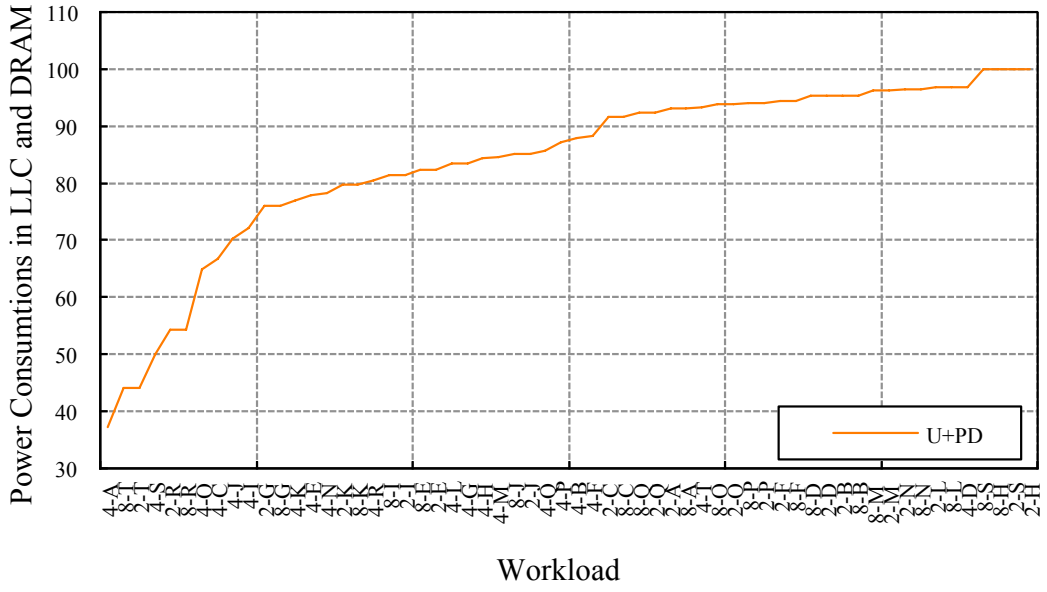


Figure 7.3: Power savings in LLC dynamic and static power consumption and dynamic power consumption of DRAM (compared to UCP).

sub optimal partitions because UCP allocates partitions according to the utility of each thread, which is performance oriented. As a result, the COOP approach will miss opportunities to find global optimal and we consider COOP as a local search for this reason. Table 7.2 shows cache partitions from an online algorithm, UCP, and the exhaustive search with 1% performance degradation, U+PD, for G2 workloads. We observe that a local search may result in suboptimal cache partitions: *e.g.* G2-E shows cache partitions of 1-7 and 2-4 for UCP and U+PD, respectively. A local search, downsizing each partition individually, comes up with 1-4; however, the global optimum is 2-4. As such, we need an algorithm that can search beyond the scope of a local search.

7.4 PEG-based Cache Partitioning Algorithm Design

To address the local minima problem, we design our heuristic by taking inspiration from simulated annealing [66]. Simulated annealing is a probabilistic meta-heuristic for a global optimization. It often finds the global optimal by mutating search directions so that local minima do not dominate the search direction. Such mutation is designed to find *neighbours*, and this enables it to escape local minima. We design a GCD approach for LLC partitioning and add *upsizing* step to define *neighbours* in the simulated annealing.

Upsizing We add a mutation step when searching for power-efficient cache partitions. This mutation step should change direction during a search to escape from a local optimal. Also it must eventually converge. To achieve these goals, we imple-

ment an *upsizing* step to escape from the down-sized partitions from the partitions given by UCP so that we can find the global optimal.

Our GCD approach normally searches for a solution by downsizing the partition with the biggest PEG value. This approach does not search different combinations of partitions from the UCP partitions. For example, partitions of 5/2 are not discoverable with this approach if the given UCP partitions are 3/4 because the first partition starts from 3 and it will continue the search with downsizing steps; however, the optimal partition is 5. To overcome this limited search scope, we consider most power efficient upsizing cases across all cores. To guarantee convergence, we limit upsizing to partitions which increase power consumption less than the power savings achieved by the current downsizing. In contrast to downsizing in which we used the metric, $\Delta Power/\Delta AMAT$ to pick the most profitable cache to downsize, for upsizing, we use the metric, $\Delta AMAT/\Delta Power$ to pick the partition which improves performance most within power constraints.

In terms of GCD, the coordinate system consists of each partition of the CMP and the constraint is the performance degradation compared to the performance of UCP. We start from the cache partitions of UCP and reduce the capacity while upsizing partitions when the power consumption increase caused by the upsizing is smaller than the power savings from the downsizing. We pick maximum PEG value for downsizing and minimum reciprocal PEG value for upsizing.

PCP Algorithm Description Our greedy coordinate descent LLC partitioning algorithm is shown in Algorithm 7. First, we relax the AMAT constraints by mul-

tiplying the number of cores in Line 7.5 so that our PCP can explore beyond local minima. Second, we use AMAT to approximate a weighted speedup. We used AMAT to approximate performance level for our uniprocessor algorithm in Chapter 6; however, we need to estimate a weighted speedup for CMPs. As such, we implement the get_amat_WS procedure in Line 7.8 to estimate a weighted speedup according to the current LLC partitions. As we did in our uniprocessor MCR, we estimate each AMAT per core, and then we estimate a weighted speedup as in Equation 7.1. We substitute IPC for $1/AMAT$ in this case.

Inside of the main loop below Line 7.8, we implement our greedy search which consists of downsizing and upsizing. From Line 7.9 to Line 7.14, we find the best candidate partition to resize by picking the maximum PEG. In Line 7.13, we terminate our greedy search if there is no more candidates with a positive PEG value. In Line 7.14, we set the number of ways in the downsizing to limit maximum ways in the following upsizing step.

As we mentioned earlier, in our greedy search we search neighbors to escape potential local minima. To define the neighbors, we search up-sizable partitions: upsizing should not increase the power consumption more than the power savings achieved by the previous downsizing, and the maximum ways in upsizing are limited to the previous number of ways in the downsizing. First, we implement *get_min_peg* in Line 7.16 to find a potential up-sizable partition which reduces AMAT most at the smallest power overhead. Its objective function is the reciprocal of PEG. This is a minimization problem. As such, the implementation of *get_min_peg* is straightforward.

Algorithm 7: PEG-based Cache Partitioning

```
7.1  begin
7.2      foreach cpu i do
7.3          prepare wcL1, wcL2, wcL3 and totalAccesses per cpu i
7.4          /* gives more room to find a global maximum */
7.5          balances[i] = get_delta_amat(wcL1[i], wcL2[i], wcL3[i], allocLLC[i], 1-(1-perfLimit)
             x ncpus)
7.6      end
7.7      /* main loop */
7.8      while get_amat_WS(wcL1, wcL2, wcL3, allocUCP, allocLLC ) >= perfLimit do
7.9          foreach cpu i do
7.10             peg[i] = get_max_peg(wcL3[i], totalAccesses[i], allocLLC[i], balances[i])
7.11         end
7.12         winner = core with maximum value of peg
7.13         if peg[winner] < 0 then break
7.14         plus_alloc = allocLLC[winner] - peg[winner].alloc
7.15         foreach cpu i and i is not equal to winner do
7.16             up_peg[i] = get_min_peg(wcL3[i], totalAccesses[i], allocLLC[i], balances[i],
                 plus_alloc)
7.17         end
7.18         up_winner = core with minimum value of 1/ peg
7.19         if peg[winner].deltaPower > 0 and peg[winner].alloc >= 1 then allocLLC[winner]
             = peg[winner].alloc
7.20         if up_peg[up_winner].deltaPower < peg[winner].deltaPower and get_amat_WC) >=
             perfLimit then allocLLC[up_winner] = up_peg[up_winner].alloc
7.21     end
7.22     return allocLLC
7.23 end
```

Lastly, we update the partitions when desired conditions are met. In Line 7.19, we downsize a partition only when the Δ power is positive (*i.e.* power savings). In Line 7.20, we upsize a partition only when the Δ power when upsizing is smaller than the Δ power that would occur for downsizing.

7.5 Evaluation of PEG-based Cache Partitioning

7.5.1 Offline Analysis

In this section, we compare our off-line version of PCP to U+PD and COOP. To realize an off-line version of PCP, we apply our PCP algorithm in Algorithm 7 on the exhaustive simulations acquired in Chapter 4.

Threshold Issue Unlike our PCP implementation, COOP requires a threshold to control the aggressiveness in resizing the LLC. For a fair comparison, we ran COOP repeatedly with different thresholds until it showed similar performance level to our PCP implementation. We were able to find thresholds for COOP, which results in similar performance levels, and the thresholds are 0.8039, 0.9, and 0.902 for G2, G4, and G8 workloads, respectively. So, our results represent a best-effort implementation for COOP.

Evaluation We first compare the power savings of PCP to COOP as shown in Figure 7.1. PCP exhibits power savings of 1.6% compared to the total power consumption of UCP. Meanwhile, COOP shows power savings of 0.5%. Not only is PCP's power savings greater than COOP's, it also achieves slightly better system

throughput. However, compared to U+PD, PCP performs worse, achieving only 57% of the power savings of U+PD. This shows off-line exhaustive search still does better than our off-line version of our PCP algorithm.

Table 7.2 shows the cache partition solutions by each techniques and their power consumption (includes both LLC and DRAM). Note that for theDRAM power, we only consider its dynamic power consumption. As can be seen, PCP is superior to COOP in two ways. First, the cache partitions of PCP are not limited to individual down sizing from the partition of UCP. As we pointed out earlier, COOP follows the same algorithm in UCP and it stops when the utility is below the given threshold. Besides the overhead of setting the threshold, COOP finds suboptimal partitions due to the limitation in exploring other possible (non-UCP) partitions. In contrast, PCP can search beyond the UCP partitions as shown in G2-A, G2-C, G2-E, G2-F, and G2-I due to the upsizing step discussed in Section 7.4. As a result, PCP achieves greater power savings compared to COOP.

Second, PCP shows better adaptiveness across the workloads. COOP exhibits LLC resizing only in three workloads, G2-F and G2-G, and G2-S. However, PCP exhibits LLC resizing in half of the workloads. As a result, PCP results in 7.0 active ways on average compared to 7.3 ways of COOP.

From the perspective of fairness, UCP improves fairness and U+PD achieves slightly better fairness compared to EP while showing worse fairness compared to UCP as shown in Figure 7.4. COOP achieves similar, but slightly worse fairness compared to UCP and our off-line version of PCP achieves similar, but slightly better fairness compared to U+PD.

	Partition				Power Consumption		
	UCP	U+PD	COOP	PCP	U+PD	COOP	PCP
G2-A	6-2	5-2	6-2	5-3	0.931	1.000	1.001
G2-B	6-2	5-2	6-2	5-1	0.954	1.000	0.970
G2-C	7-1	6-1	7-1	5-2	0.916	1.000	0.927
G2-D	6-2	5-2	6-2	5-1	0.953	1.000	0.930
G2-E	1-7	2-4	1-7	2-4	0.824	1.000	0.824
G2-F	2-6	1-6	1-2	1-7	0.945	0.729	0.991
G2-G	2-6	1-2	1-2	1-6	0.760	0.760	0.941
G2-H	1-7	1-7	1-7	1-7	1.000	1.000	1.000
G2-I	7-1	5-1	7-1	4-2	0.815	1.000	0.838
G2-J	6-2	5-1	6-2	4-1	0.851	1.000	0.797
G2-K	7-1	5-1	7-1	4-1	0.798	1.000	0.729
G2-L	6-2	5-2	6-2	6-2	0.967	1.000	1.000
G2-M	7-1	6-1	7-1	7-1	0.962	1.000	1.000
G2-N	7-1	6-1	7-1	7-1	0.965	1.000	1.000
G2-O	7-1	3-5	7-1	7-1	0.939	1.000	1.000
G2-P	6-2	6-1	6-2	6-2	0.940	1.000	1.000
G2-Q	7-1	6-1	7-1	6-1	0.924	1.000	0.924
G2-R	7-1	2-1	7-1	7-1	0.542	1.000	1.000
G2-S	1-7	1-7	1-2	1-7	1.000	0.727	1.000
G2-T	6-2	2-1	6-2	4-1	0.441	1.000	0.659
AVG	8.0	6.4	7.3	7.0	0.871	0.961	0.927

Table 7.2: Cache partition and power consumptions of LLC and DRAM.

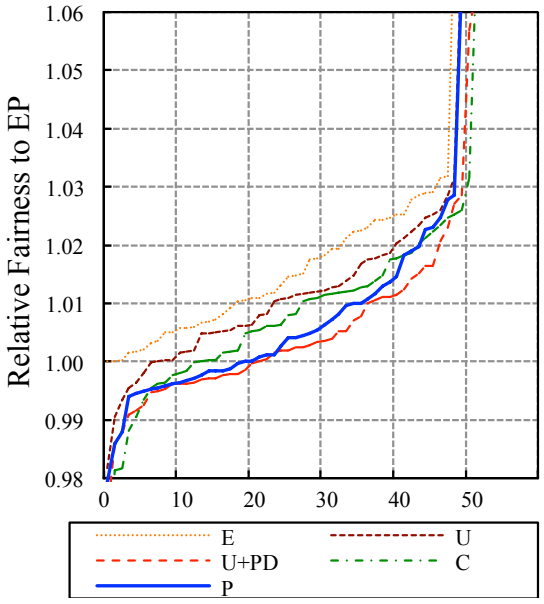


Figure 7.4: Fairness comparison between exhaustive search to find the optimal partitioning for the best throughput (E), UCP (U), exhaustive LLC resizing search to find the optimal partitioning for the best power savings based on UCP with the performance degradation level of 1% (U+PD), COOP (C), and our PEG-based cache partitioning (P). Fairness is normalized to the fairness of even partitioning (EP).

Discussion As we discussed in Section 6.3.2, our prediction-based approach introduces errors. To understand the gap between the power savings of PCP and the power savings from U+PD, we study two ideal cases which eliminate two main errors in PCP: weighted speedup approximation, and AMAT and power estimations. I-PCP-1 eliminates the error from our weighted speedup (WS) approximation. Instead of approximating the WS by means of AMAT, I-PCP-1 computes the actual WS based on our extensive simulation results in Section 4.4. Likewise, I-PCP-2 further removes errors from our way-counts based AMAT and power estimations. Figure 7.5 summarizes the results from this study. I-PCP-1 achieves power savings of 63% compared to the power savings of U+PD. I-PCP-2 achieves power savings of 84% compared to the power savings of U+PD. In comparison, PCP only achieves power savings of 57%. This shows the WS approximation and way-counts-based prediction introduce errors that limit limit the realizable power savings.

Although I-PCP-2 achieves more than 80% of the power savings of exhaustive search, U+PD, the gap in the power savings of 16% still persists. This is because our PCP algorithm does not always find the optimal solution and Table 7.3 shows a few representative cases. As we discussed earlier, the exhaustive search, U+PD, often exhibits cache partitions different from cache partitions consisting of each shrunken partition of the UCP solution, *e.g.* UCP cache partition of 1/7 vs. U+PD cache partition of 2/4 in G2-D. To cope with this, our PCP algorithm searches beyond local optima via the upsizing step in Algorithm 7. In particular, workloads G4-H and G4-Q show successful upsize-assisted searches. However, there are also counter examples. G2-D fails to find the solution of 2/4 by upsizing the partition of Core 0.

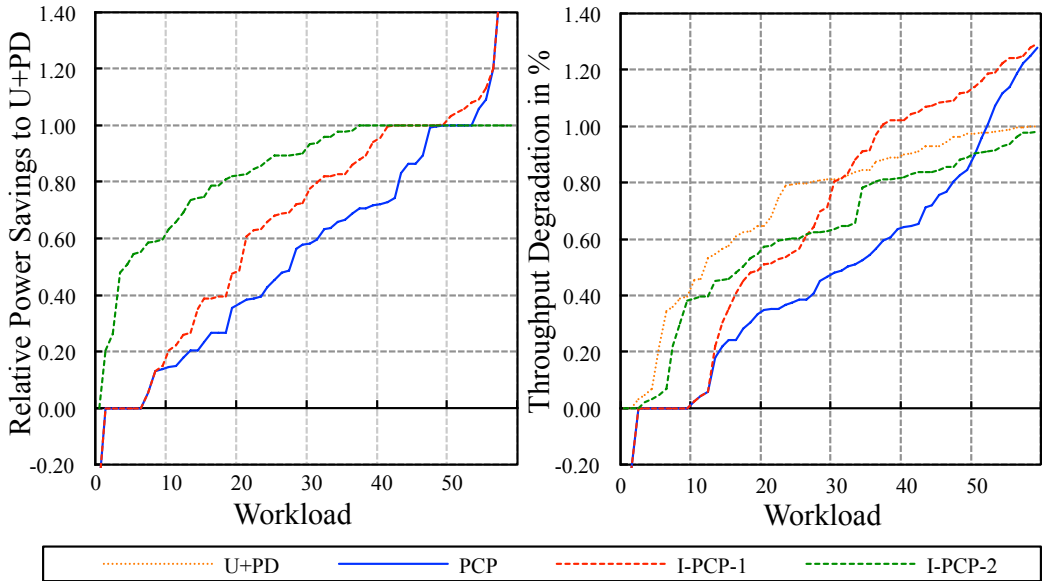


Figure 7.5: Power and system throughput comparison between PCP, I-PCP-1, and I-PCP-2. Power savings is normalized to the power savings of U+PD. Throughput is normalized to the throughput of UCP.

G2-O exhibits a large separation between the partitions of UCP and the partitions of U+PD (8-way differences: 4 ways per core). PCP is only able to find 6/2 instead of 3/5 of U+PD. In other words, PCP is stuck in the local optimum of 6/2 and fails to search the global optimum of 3/5. Likewise, G4-C shows another example of returning a local optimum: PCP was able to upsize the partition of Core 2 from 2 to 4, but failed to upsize to the partition of 13 in U+PD. G8-O exhibits another example of local optimum: PCP finds the optimal partition of Core4 by increasing its size up to 4, but fails to find the optimal partition of Core2 and Core6. So, we conclude that the heuristic global search in PCP results in the loss of power savings of 16% compared the power savings of U+PD.

WL	UCP	U+PD	I-PCP-2
G2-D	1/7	2/4	1/5
G2-O	7/1	3/5	6/2
G4-C	4/7/2/3	1/1/13/1	1/2/4/1
G4-H	2/2/11/1	2/4/7/1	2/4/7/1
G4-Q	6/1/7/2	4/1/7/4	4/1/7/4
G8-O	1/9/5/1/2/9/4/1	1/6/2/1/4/6/2/1	1/6/3/1/4/6/3/1

Table 7.3: Cache partition from UCP, U+PD, and I-PCP-2.

7.5.2 Online Analysis

Epoch Size As mentioned before, selecting an epoch size is crucial for our online techniques. To determine an appropriate epoch size for cache partitioning, we evaluate the performance of UCP using 3 different epoch sizes: 5M, 10M, and 20M cycles. Table 7.4 reports the throughput achieved for each epoch size normalized to the weighted speedup of even partitioning (EP). The last column in Table 7.4 shows the normalized throughput for the static (off-line) UCP technique. Online cache partitioning does not exceed the throughput of static UCP on average. There are two reasons for this. First, generally, a last level cache requires longer epochs compared to the L1 or L2 caches because it has a larger capacity and thus has a longer transient of the cache resizing event that occurs at the beginning of each epoch. Second, larger epochs sacrifices opportunities for adaptation. As a result, online throughput results are worse than the results for static UCP which has an

Epoch Size	5M	10M	20M	Static
Normalized WS	1.0125	1.0189	1.0186	1.0203

Table 7.4: Throughput of UCP according to the epoch size for G2 workloads.

oracle view over the way counts. So, we pick a 10M-cycle epoch size which shows the best throughput among these candidates.

Evaluation We first compare the power savings of PCP to COOP as shown in Figure 7.6. Note that power consumption levels are normalized to the power consumption of UCP. Throughput is also normalized to the weighted speedup of UCP. PCP exhibits 1.8% power savings while COOP shows 0.35% power savings. Hence PCP has better power savings compared to COOP while maintaining slightly higher performance. (The performance difference is not significant because both techniques are within the 1% performance degradation level on average.) Note that for PCP, there are 10 workloads with more than 1% performance degradation. This is due to errors in approximating the weighted speedup, already discussed in Section 7.5.1 for static PCP. Another important point is that PCP does not require threshold adjustment as COOP does. Note that we carefully tuned thresholds for COOP offline to achieve similar performance degradation level compared to the performance level of PCP.

Online PCP reduces the total system power consumption by almost the same amount as static PCP. However, online PCP exhibits worse performance degradation: performance degradation in online PCP is as high as 4% compared to 1.3%

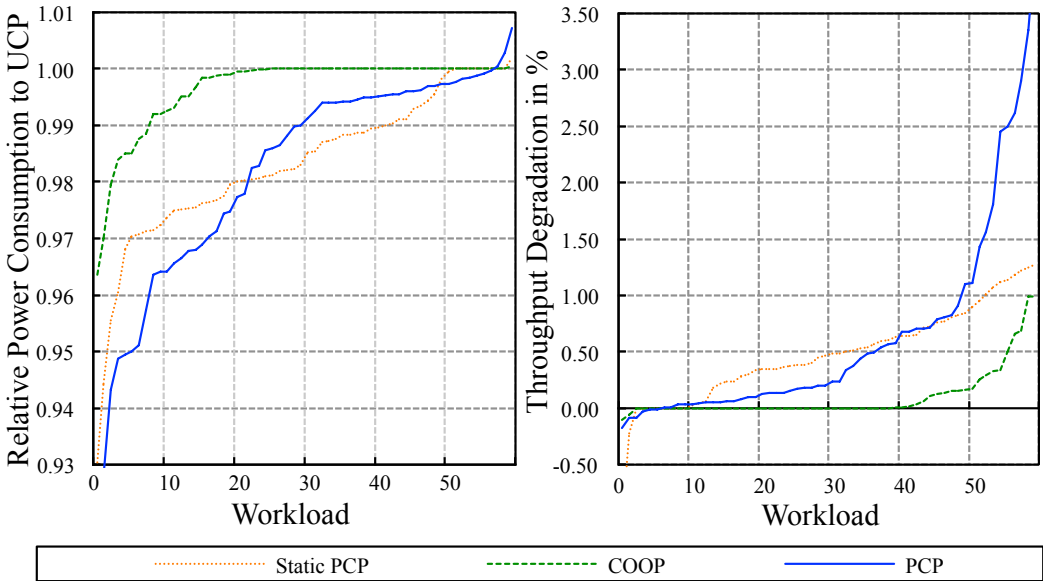


Figure 7.6: Power and system throughput comparison between online PCP, COOP, and PCP.

for static PCP.

Chapter 8

GCD CMP Multi-Level Cache Resizing

The search complexity in optimizing cache sizes in a CMP multi-level cache hierarchy is $O(k^{nm})$. We showed that the GCD method can reduce the search complexity of uniprocessor multi-level cache resizing from $O(k^m)$ to $O(km)$. With a similar approach, we can expect to reduce the search complexity of CMP multi-level cache resizing from $O(k^{nm})$ to $O((km)^n)$. But this still involves non-polynomial search complexity. Such high search complexity is infeasible for a run-time algorithm. As such, we need to further reduce the search complexity. In this chapter, we provide a limit study of CMP multi-level cache resizing to show the motivation of our approach, and then we present two techniques to reduce the complexity of CMP multi-level cache resizing.

8.1 Limits of CMP Multi-Level Cache Resizing

Quasi Optimal Search via NM Method We conduct an offline limit study for CMP multi-level cache resizing to understand the limit of power savings with a given performance degradation level. Although this limit study requires $O(k^{nm})$ simulations, we approximate the limits by combining the exhaustive simulations that we have discussed in Section 4.4. Combining the exhaustive simulation results does not consider contention at the shared bus between L3 caches and private caches.

Moreover, because the off-chip bandwidth is not accurately simulated, those combinations are one of approximations to predict total system power consumption and throughput.

Unfortunately, even this approximation requires exponential computation times and it is not feasible to get the approximation result beyond a two core system. Instead of performing an exhaustive search over the entire solution space, we run the *Nelder-Mead* simplex method to estimate the limit. While the Nelder-Mead (NM) simplex method does not always produce a global optimum, it often finds the global optimum, *e.g.* NM method finds the global optimum in 7 out of 20 G2 workloads (see below). As such, for G4 and G8 workloads, we repeat the method 100–1,000 times and pick the best one to estimate the global optimal.

Since we have the global optimal result for a two-core system based on our extensive simulation results, we first compare the result from exhaustive search and the NM result to assess the quality of our NM-based approximation. As shown in Table 8.1, the NM method's search results are similar to the solutions from the exhaustive search from the perspective of weighted speedup and power savings. Weighted speedup and power consumption are normalized to the result of EP. In particular, the solutions from exhaustive search and the NM method for G2-B, G2-C, G2-F, G2-G, G2-K, G2-L, G2-R, and G2-S show the exact same weighted speedup and power consumption. Moreover, the weighted speedup and power consumption of exhaustive search and NM method averaged across all workload, show only 0.05% and 0.16% differences, respectively. As a result, NM method shows 18.3% power savings compared to the power savings of 18.4% in the exhaustive search. Therefore,

we can conclude that the quality of the solutions from the two different approaches are essentially the same.

Comparing the CMP cache configurations, as shown in Table 8.2, in 7 out of 20 cases (G2-B, G2-F, G2-G, G2-K, G2-L, G2-R, and G2-S), the solutions from NM method are identical to the cache configurations from the exhaustive search. And in the cases where the NM method does not find the global optimal, one core's cache configuration is identical to the cache configuration of the global optimal in 7 out of 13 cases (G2-A, G2-C, G2-D, G2-E, G2-I, G2-N, and G2-Q) and at least 3 variables are matched for all 13 cases (there are 6 variables, L1/L2/L3 per core). In particular, among 13 local minima of NM method, 5/4/3 variables are matched in 1/7/5 cases, respectively. On average, the errors are 2.9 ways across these 13 local minima, *e.g.* the error in G2A is 2 ways because L1 and L2 cache configurations are not matched and the sum of the differences are 2.

As such, we estimate the limits of CMP multi-level cache resizing by using NM method instead of exhaustive search which is virtually impossible to finish. Due to the huge problem space for 8-core system, we repeat 1,000 times and each run consists of 500 iterations in the NM method. We repeat 100 times for 2- and 4-core systems. We refer to this NM-based approximation as a *quasi* optimal and treat it as an upper bound against which we we compare our CMP MCR techniques.

Maximum Power Savings with 1% Performance Degradation We examine the limits of power savings and performance of CMP multi-level cache resizing using the quasi optimal solution from the NM method. In this limit study, we take

	Exhaustive Search		Nelder-Mead Simplex Search	
WL	Weighted Speedup	Power	Weighted Speedup	Power
G2-A	0.99938	0.83831	1.00057	0.84058
G2-B	1.00955	0.84244	1.00955	0.84244
G2-C	1.01473	0.82938	1.01473	0.82938
G2-D	1.00843	0.84256	1.00927	0.84562
G2-E	0.99268	0.76075	0.99195	0.76105
G2-F	1.00602	0.82744	1.00602	0.82744
G2-G	0.99499	0.79515	0.99499	0.79515
G2-H	1.14518	0.80926	1.14549	0.81048
G2-I	0.99282	0.77028	0.99419	0.77495
G2-J	0.99702	0.76576	0.99649	0.76952
G2-K	1.00461	0.74897	1.00461	0.74897
G2-L	0.99955	0.85878	0.99955	0.85878
G2-M	1.00707	0.85408	1.00830	0.85697
G2-N	1.01450	0.84741	1.01468	0.84814
G2-O	1.05188	0.86533	1.05698	0.86801
G2-P	1.00258	0.81240	1.00158	0.81539
G2-Q	1.02004	0.80579	1.02019	0.80648
G2-R	0.99856	0.76866	0.99856	0.76866
G2-S	1.01744	0.82609	1.01744	0.82609
G2-T	0.99168	0.84874	0.99177	0.84918
AVG	1.01344	0.81588	1.01385	0.81716

Table 8.1: Performance and power of static optimal CMP cache configurations from exhaustive search and Nelder-Mead simplex search with 1% performance degradation. All values are normalized to the result of the even partitioning (EP).

	Exhaustive Search						Nelder-Mead Simplex Search					
	Core 0			Core 1			Core 0			Core 1		
WL	L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3
G2-A	3	3	6	4	3	2	3	3	6	3	4	2
G2-B	3	3	6	4	3	1	3	3	6	4	3	1
G2-C	3	3	7	4	7	1	2	3	7	4	7	1
G2-D	1	3	6	4	1	2	2	2	6	4	1	2
G2-E	2	4	2	1	4	5	4	6	1	1	4	5
G2-F	4	4	1	4	2	6	4	4	1	4	2	6
G2-G	4	4	1	4	2	3	4	4	1	4	2	3
G2-H	3	3	1	2	1	7	2	4	1	1	1	7
G2-I	1	4	5	2	1	2	1	4	5	3	2	2
G2-J	1	4	6	4	3	1	1	5	5	4	5	1
G2-K	1	4	5	1	1	1	1	4	5	1	1	1
G2-L	3	3	6	3	4	1	3	3	6	3	4	1
G2-M	3	3	6	2	1	2	4	3	6	4	1	2
G2-N	3	2	7	4	2	1	2	3	7	4	2	1
G2-O	1	2	2	1	1	6	4	2	2	2	1	6
G2-P	1	2	7	3	3	1	2	3	7	1	3	1
G2-Q	1	3	7	4	7	1	2	2	7	4	7	1
G2-R	1	2	2	1	1	1	1	2	2	1	1	1
G2-S	1	1	1	2	2	7	1	1	1	2	2	7
G2-T	2	2	4	8	3	1	4	3	3	8	4	1
AVG	2.1	2.95	4.4	3.1	2.6	2.6	2.5	3.2	4.25	3.1	2.85	2.6

Table 8.2: CMP cache configurations from exhaustive search and Nelder-Mead simplex search with 1% performance degradation.

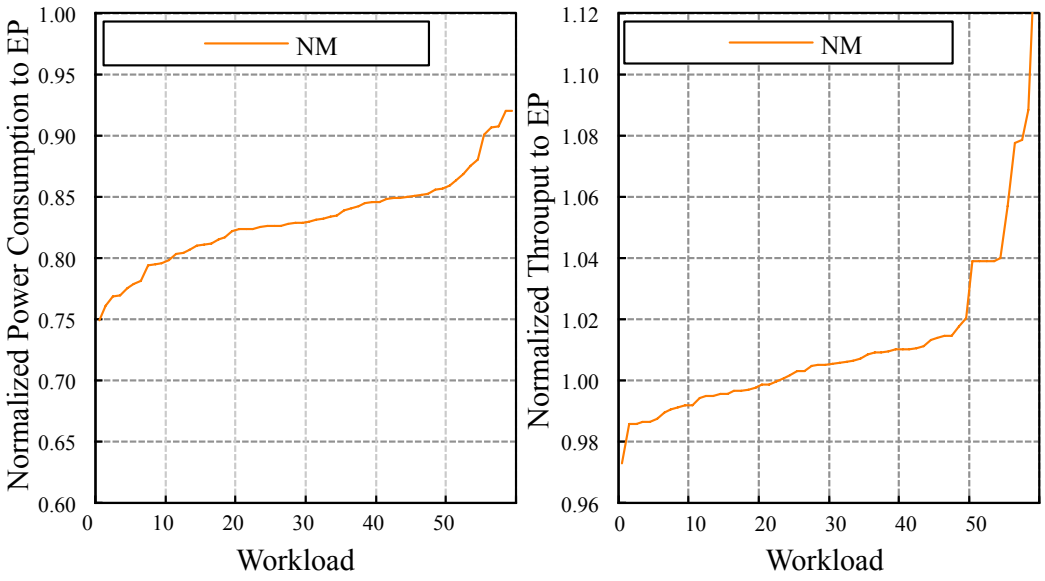


Figure 8.1: Power consumption and throughput of CMP multi-level cache resizing. Approximated by Nelder-Mead simplex method.

the performance of UCP as a reference and allow 1% performance degradation as we did in Chapter 7 for U+PD. Figure 8.1 summarizes the power consumption and performance of the quasi optimal configuration. On average, the static quasi optimal exhibits 83.2% of the power consumption compared to even partitioning (EP), demonstrating power savings of 16.8%. Note, unlike uniprocessor MCR, there are around 20 workloads (between workload 0 and workload 20 in the throughput chart in Figure 8.1) which underperform EP. UCP generally results in better performance than EP, but in some cases, UCP underperforms EP. As a result, our approximation of static optimal cache configurations also shows worse performance degradation than 1% in some cases. In other words, if we have a better-performing online cache-partitioning technique, our static optimal performance level will exhibit no worse than 99% of EP.

Power Savings in the Cache Hierarchy CMP multi-level cache resizing effectively reduces wasteful power consumption from the cache hierarchy. Table 8.3 summarizes the relative power consumption of our static quasi-optimal search using the NM method. And Figures 8.2, 8.3, and 8.4 show the power consumption breakdown for G2, G4, and G8 workloads, respectively. Relative power consumption level in Table 8.3 is normalized to the power consumption of UCP while Figures 8.2, 8.3, and 8.4 report actual power values. Each bar in Figures 8.2, 8.3, and 8.4 breaks down the power consumption across all cores, and hence, the values in G8 workloads are higher than the values in G2 or G4 workloads.

The static power consumption of each caching level reveals the relative cache size because we assume power gated sub-arrays of unused portions of cache do not consume any static power. On average, the quasi optimal reduces the L1 and L2 sizes by more than half, as seen by comparing the static power consumption of the L1 and L2 caches in Table 8.3. On the other hand, the total L3 cache size is reduced by around 18% and the dynamic power in the L3 cache is increased by 21%. This active L3 size on average indicates that the L1 caches have highest PEG, the second highest PEG for the L2 caches, and the L3-cache partitions have lowest PEG on average.

The power savings in the cache hierarchy not only vary across workload groups, but also show significant variance within a group. In particular, G2-R exhibits the biggest power savings, 70%, with the cache configuration of 1/1, 2/1, and 2/1 for L1, L2, and L3 caches, respectively (core 0 / core 1 configuration), while overall the G2 workload group shows a power savings of 45.3% on average, as shown in

Group	L1D	L1S	L2D	L2S	L3D	L3S	All
G2	0.320	0.353	0.833	0.387	1.250	0.857	0.547
G4	0.316	0.355	0.961	0.463	1.163	0.788	0.559
G8	0.444	0.447	0.777	0.493	1.226	0.821	0.645
AVG	0.360	0.385	0.857	0.448	1.213	0.822	0.584

Table 8.3: Average relative power consumption per caching level and per dynamic/static power. Values are normalized to the power consumption of UCP.

Figure 8.2. On the other hand, G2-C exhibits the smallest power savings of 32.8% among the G2 workload group. Among G4 workloads, G4-P shows the biggest power savings of 55.6% with the cache configuration of 1/1/1/1, 3/4/2/4, and 3/3/2/1. G4-M exhibits the smallest power savings of 30.2% while overall the G4 workload group shows power savings of 44.1%. Lastly, for G8 workloads, G8-N achieves power savings 53.1% with the cache configuration of 2/2/3/2/3/2/2/3, 3/2/4/4/4/4/5/4, and 2/1/1/1/1/2/1/3. On the other hand, G8-M exhibits power savings of 18.1% while overall the G8 workload group shows power savings of 35.5% on average, as shown in Figure 8.4.

Although only L3 dynamic power is increased on average, there are 14 workloads that exhibit increased L2 dynamic power. On average, however, L2 dynamic power is decreased by 14%. In particular, G2-J and G4-B show L2 dynamic power increase by 70% as shown in Figure 8.2 and 8.3, respectively. Moreover, L3 dynamic power is increased by as much as 160% for G2-R, as shown in Figure 8.2.

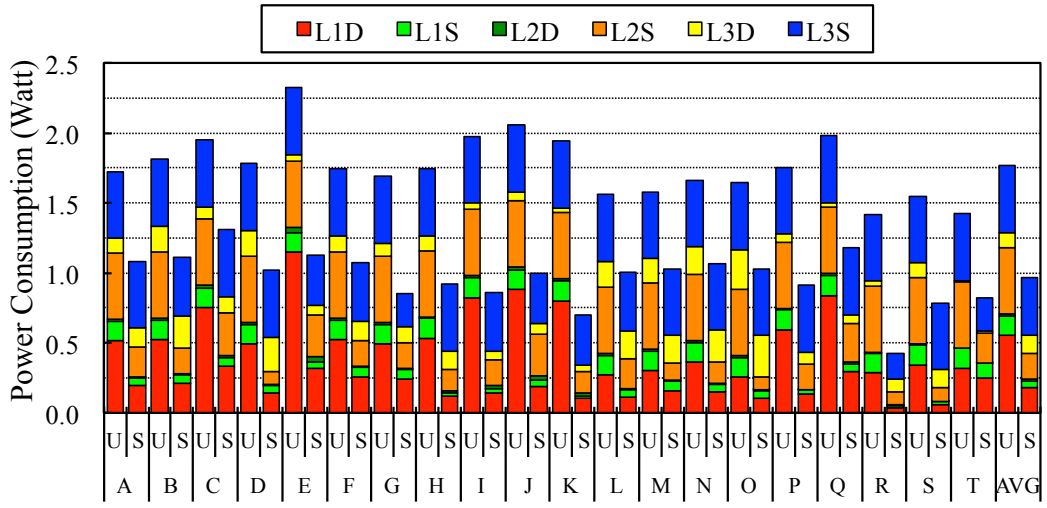


Figure 8.2: Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G2 workloads.

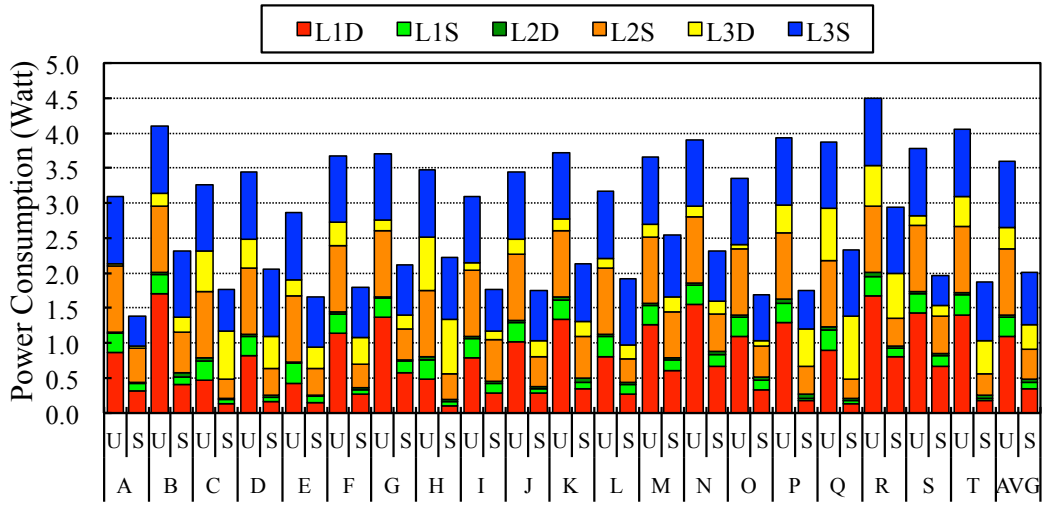


Figure 8.3: Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G4 workloads.

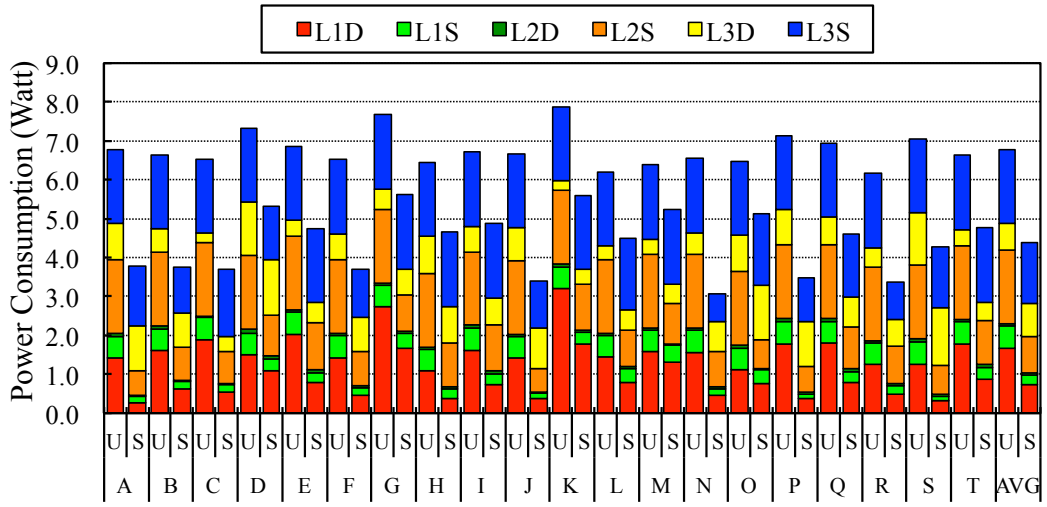


Figure 8.4: Power consumption breakdown of UCP (U) and Static Quasi-Optimal (S) for G8 workloads.

Comparing Uniprocessor MCR and CMP MCR Figure 8.5 shows cache power consumption breakdown for some representative G2 workloads. In particular, we compare the static optimal found via exhaustive search for the 2-core system against the optimal found individually for each of the two benchmarks for a uniprocessor, as discussed in Chapter 6. We can see that the optimal multi-level cache configurations are different from the combination of the corresponding uniprocessor MCR configurations.

First, an LLC partitioning determines the system throughput so, it is crucial to know the partitions that will achieve best performance online as we pointed out in Chapter 7. However, in some cases, cache partitions in uniprocessor static optimal (UM) and in static optimal in CMP (SO) are similar. In particular, G2-R and G2-T show that cache configuration of UM and SO are identical, as shown in Table 8.4, resulting in the same power breakdown. This is mainly due to the lack of contention

in the shared LLC so that each core's uniprocessor optimization can fit in the CMP cache without considering the interaction between the cores at the LLC level.

However, G2-C and G2-L show cache partitions that improve the system throughput also change private cache sizes too. In other words, due to the different size of the LLC, per core optimization also results in different balance points. In the case of G2-C, it achieves an improved weighted speedup: more L3 cache allocation and bigger private caches in core 0 boost the performance and its impact is big enough to cancel out the performance degradation from shrunken L2's capacity in core 1, as shown in Figure 8.5 and Table 8.4. Similarly, G2-L shows core 0's bigger caches across the entire cache hierarchy and core 1's L1 cache is also bigger at the cost of the shrunken L3.

Therefore, we can conclude that we can not apply our uniprocessor MCR technique on each core to solve the CMP multi-level cache resizing problem, except in some cases where there are no constraints in the LLC partitioning. In the next section, we will discuss other methods crucial to CMP multi-level cache resizing.

8.2 Dividing CMP Multi-level Cache Resizing into Subproblems

The CMP multi-level cache resizing problem scales exponentially as the number of cores grows, *e.g.* there are 24 variables for our 8-core system study and the solution space is larger than 8^{24} (the number of L3 configurations grows with the number of cores). As we pointed out earlier in Chapter 7, a cache partitioning problem is quite different from the cache resizing problem in private caches. As

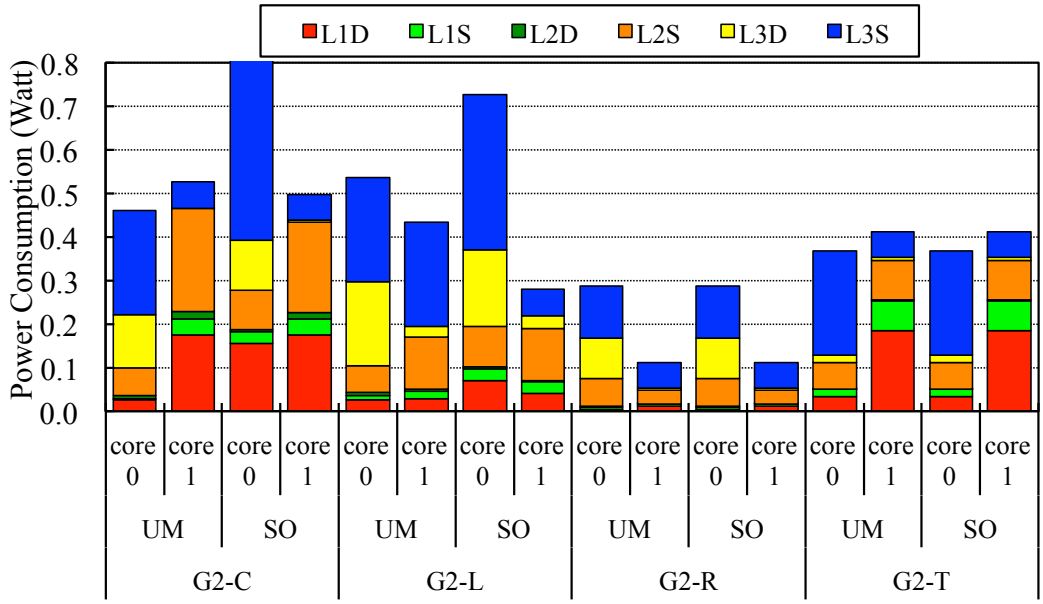


Figure 8.5: Cache power breakdown for baseline and static optimal (BL:baseline, UM:uniprocessor MCR, SO: static optimal with exhaustive search in CMP)

WL	Uniprocessor SO		CMP SO	
	Core 0	Core 1	Core 0	Core 1
G2-C	1/2/4	4/8/1	3/3/7	4/7/1
G2-L	1/2/4	2/4/4	3/3/6	3/4/1
G2-R	1/2/2	1/1/1	1/2/2	1/1/1
G2-T	2/2/4	8/4/1	2/2/4	8/4/1

Table 8.4: Cache configurations from uniprocessor static optimal and CMP static optimal.

such, the nm-variable CMP MCR problem is divided into an n-variable cache partitioning problem and an $n(m-1)$-variable cache resizing problem. Among these subproblems, we already implemented PCP to solve the n-variable cache partitioning problem (see Chapter 7).

One of the challenges in solving the $n(m-1)$-variable cache resizing problem is the convergence rate. Optimizing a problem consisting of $n(m-1)$ variables may suffer from a slow convergence rate because the number of steps in finding minima will increase as the number of cores grows. Such slow convergence will result in poor adaptability in the cache resizing. To prevent such a slow convergence rate, we divide the cache resizing problem into subproblems by grouping variables. Optimizing these subproblems might result in suboptimal solutions compared to the global optimal attained from optimizing the $n(m-1)$-variable cache resizing problem. However, there was a significant power savings achieved by the adaptability of our GCD MCR for a uniprocessor, as we discussed in Section 6.3.3. We want to preserve the adaptability of our GCD approach in the CMP multi-level cache resizing problem as well. As such, we compare the convergence rate and the power savings/performance level. For this comparison, we conduct an offline study to compare power savings and performance achieved via different grouping techniques.

Two straight forward approaches are vertical and horizontal groupings. Vertical grouping joins variables of each caching level from the same core and results in n subproblems. On the other hand, horizontal grouping joins variables in the same caching level and results in $(m-1)$ subproblems. For example, a vertical grouping divides the problem into 8 subproblems for a 8-core system, and a horizontal group-

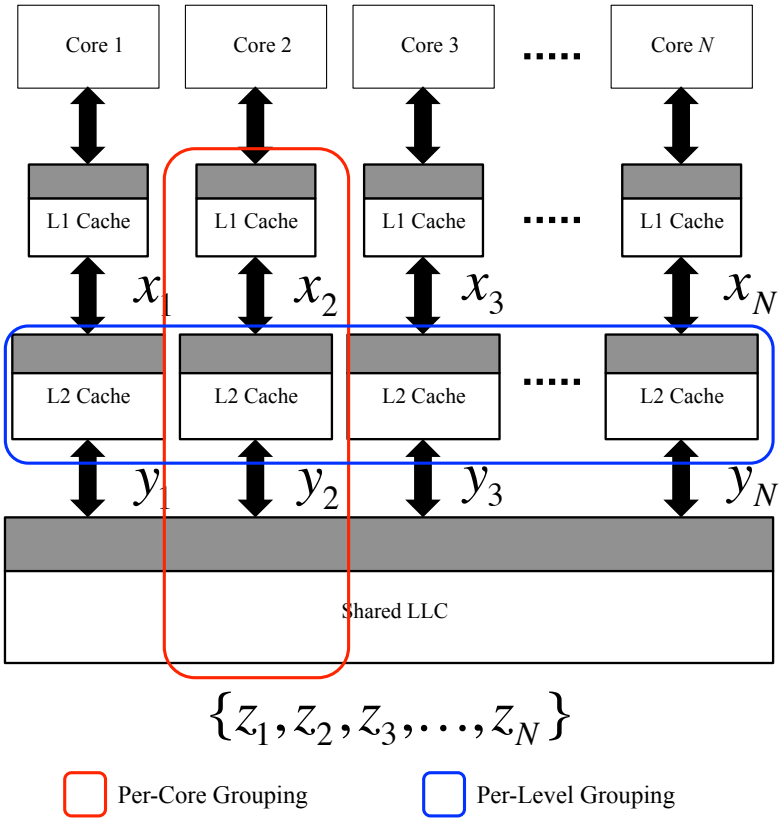


Figure 8.6: CMP multi-level cache resizing and variable grouping.

ing divides the problem into 2 subproblems for a 3-level cache hierarchy. Figure 8.6 illustrates these two different grouping techniques.

Vertical Grouping: Independent Uniprocessor MCR Vertical grouping is highly related to our previous GCD MCR technique for a uniprocessor in that its optimization scope is limited to each core. Although the number of subproblems scales linearly to the number of cores, a vertical grouping has its strength in its simplicity because it runs n uniprocessor GCD MCR independently. The difference is that the L3 capacity is not determined by the uniprocessor MCR algorithm; instead it is determined as a byproduct of L3 partitioning using our PCP algorithm.

Moreover, each solution per core reduces wasteful power consumption effectively, as we already discussed in Section 6.3.3.

Horizontal Grouping: L1– and L2–cache Groups Horizontal grouping compares PEG values across cores so that the optimization within a horizontal group results in local optimal cache configurations. Each private cache consumes dynamic power that is proportional to the number of accesses which can vary significantly across cores. As such, disabling a single way in private caches may have different PEG values. Decisions made by monitoring only each core, *i.e. vertical grouping* are prone to be destructive because each core's increased data traffic to the LLC aggravates resource conflicts in the LLC.

Horizontal grouping addresses this problem by selecting the way that provides the biggest PEG value first. This will result in saving more power by selecting ways that provide higher PEG value. As such, horizontal grouping prioritizes the ways in PEG order within the expected performance degradation estimated by Δ AMAT.

Convergence Rate Comparison We conduct an offline study to compare convergence rates between three different approaches: optimizing the entire CMP multi-level cache resizing problem, optimizing vertical groups separately, and optimizing horizontal groups separately. Figure 8.7 shows the median convergence rate of each workload group. Both vertical and horizontal groupings show reduced convergence rates compared to the convergence rate of no grouping, which increases as the number of cores grows.

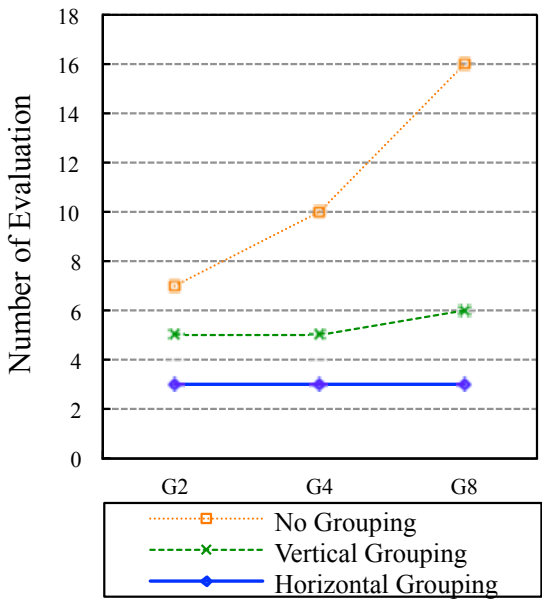


Figure 8.7: Convergence rate comparison between no gropuing, vertical grouping, and horizontal grouping.

8.3 GCD CMP Multi-Level Cache Resizing Algorithm Design

We implement the vertical and horizontal grouping for CMP multi-level cache resizing. The implementation of vertical grouping is straight forward because we can simply combine our PCP and uniprocessor MCR algorithms. The only difference between the previous implementation of PCP and the implementation for PCP and CMP cache resizing is that the new PCP will also consider an increases AMAT caused by cache resizing in the private caches. This change to PCP also occurs in our implementation of horizontal grouping. For this reason, we focus on the implementation of horizontal grouping in this section.

8.3.1 Algorithm Description

Algorithm 8 shows the main procedure of the GCD CMP multi-level cache resizing, *i.e. horizontal grouping*. Like GCD MCR for a uniprocessor, GCD CMP multi-level cache resizing algorithm collects way-counter values at epoch granularity, and each caching level has different granularity. As such, per-level cache resizing, *horizontal grouping*, is executed at different frequencies as shown in Line 8.7, 8.10, and 8.13. Moreover, the entire search procedure starts over every reset epoch counts in Line 8.16.

Loosely Coupled Performance Bound We add all $\Delta AMAT$ in an entire system to broaden search space. Unlike we strictly enforced the performance degradation in a uniprocessor GCD MCR, we allow each core exhibiting worse performance degradation. Independent control of $\Delta AMAT$ or *balance* per core prone to end up suboptimal solutions for not considering overall PEG order because relaxing performance constraints per core has better chance to explore bigger PEG gain in each core.

Balance in the Line 8.3 controls the overall system performance degradation in the relaxed manner. Each per-level cache resizing procedure, *cmp_mcr_per_level*, takes the balance as input parameters and decreases it along with cyclical way selection in the group.

Optimization within a Horizontal Group *cmp_mcr_per_level()*, as shown in Algorithm 9, implement the optimization within a horizontal group. Line 9.9 shows

cyclical iteration through the group, horizontal grouping–set of cores. As we discussed earlier, our online analysis is focused on the number of evaluations or reconfigurations, not the computation time in this iteration. In other words, this computational overhead in this iteration, $O(n)$, is actually $O(1)$ in our evaluation-time analysis domain.

Figure 8.8 shows our GCD CMP multi-level cache resizing framework. The computational overhead in each MCR logic, shaded round box, is corresponding the described Algorithm 9 and this is computational overhead in our design. On the other hand, the convergence rate, which is determined by the number of iteration of switching between each MCR logic, is corresponding the described Algorithm 8. This convergence rate dominates the speed of reconfiguration and thus, determines the power savings and performance of this algorithm.

8.3.2 Implementation

Our GCD CMP multi-level cache resizing algorithm can be implemented either entirely in hardware or partially in software with support of hardware. Entire hardware implementation needs less software modification and can be stand-alone solution, and software implementation with hardware support has fewer hardware modification and flexibility in the implementation.

Hardware Overhead The solid boxes in Figure 8.8 shows the minimal hardware modification for either hardware or software implementation. These mandatory hardware modifications consist of two components. First way counters should be

Algorithm 8: GCD CMP Multi-Level Cache Resizing Algorithm

```
8.1  gcd_cmp_mcr(perfLimit):
8.2  begin
8.3      balance = sum of each core's AMAT (get_delta_amat())
8.4      alloc = current allocation of CMP caches, LLC follows UCP partitions
8.5      /* main loop */
8.6      while true do
8.7          if not (epoch % L1_FREQ) then
8.8              cmp_mcr_per_level(L1, balance, alloc)
8.9          end
8.10         if not (epoch % L2_FREQ) then
8.11             cmp_mcr_per_level(L2, balance, alloc)
8.12         end
8.13         if not (epoch % L3_FREQ) then
8.14             pcp(alloc)
8.15         end
8.16         if not ( epoch % RESET_FREQ) then
8.17             start new search
8.18         end
8.19     end
8.20 end
```

Algorithm 9: CMP Multi-Level Cache Resizing Per-Level Algorithm

```
9.1  cmp_mcr_per_level(level, balance, alloc):
9.2  begin
9.3      /* main loop */
9.4      while balance do
9.5          foreach cpu i do
9.6              budgetUsed += Δ AMAT caused by current cache resizing
9.7          end
9.8          balance -= budgetUsed
9.9          foreach cpu i do
9.10             totalAccesses = total memory references of cpu i
9.11             wc[i] = current way counts of cpu i
9.12             peg[i] = get_max_peg(wc[i], totalAccesses, alloc[i][level], balance)
9.13         end
9.14         winner = cpu with maximum value of peg
9.15         if peg[winner].peg <= 0 then
9.16             break
9.17         end
9.18         balance = balance – peg[winner].deltaAMAT
9.19         alloc[level][winner] = peg[winner].alloc
9.20     end
9.21     return
9.22 end
```

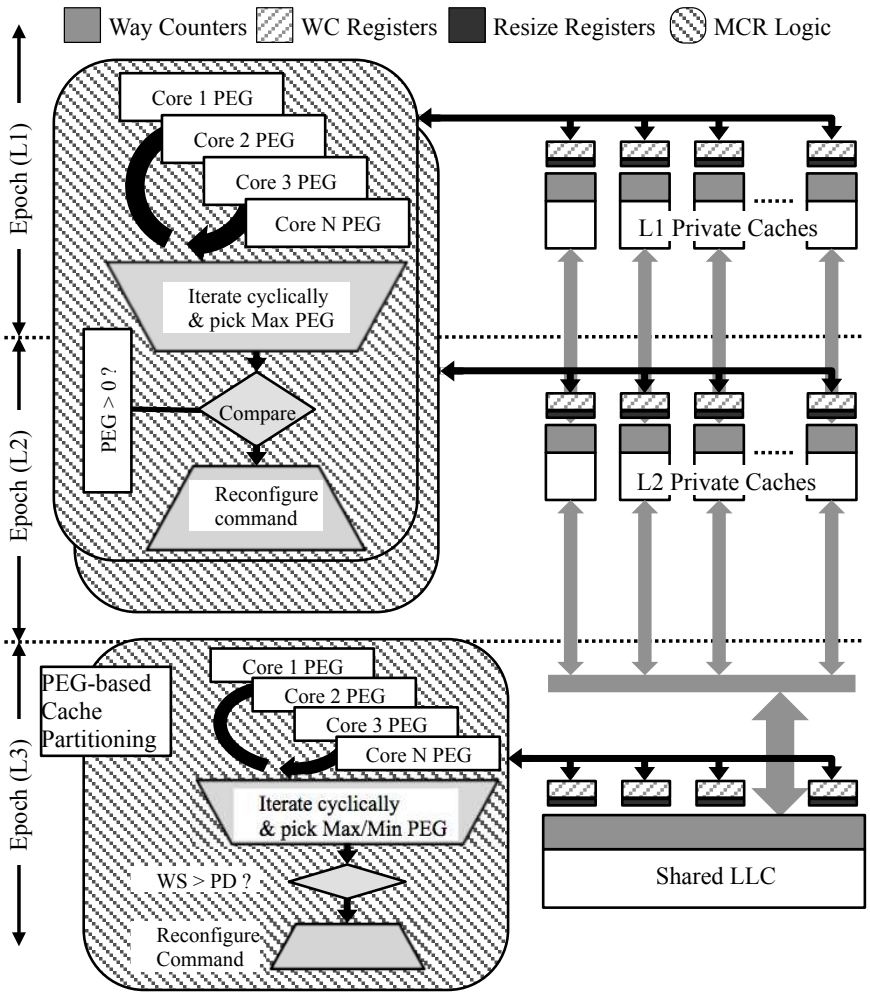


Figure 8.8: GCD CMP multi-level cache resizing framework.

added to approximate stack distances. Even software techniques can be adopted to approximate the stack distances, but our granularity is around 10-100K cycles, so we assume that hardware way counters are mandatory.

Lightly shaded boxes can be also implemented in hardware. In such case, MCR logics require way counters and PEG registers, and energy constant registers as memory and divider to compute PEG from way counter and energy constant.

Software Overhead The software implementation needs minimal hardware support. The lightly shaded boxes can be implemented inside interrupt handler routine of OS.

8.4 Evaluation of GCD CMP Multi-level Cache Resizing

In this section, we evaluate the power savings and performance degradation of our GCD CMP multi-level cache resizing. We will first evaluate our offline study. And then, we will discuss our online results.

8.4.1 Offline Analysis

Evaluation We first compare the power savings of vertical grouping (VG) and horizontal grouping (HG) CMP multi-level cache resizing. VG and HG exhibit the total system power consumption of 92.4% and 91%, compared to the total system power consumption of UCP, respectively. Comparing to the power savings of our static quasi-optimal (SQO) that exhibit the total system power consumption of 84.8% (15.2% power savings), VG and HG achieve as mush as 50% and 60% in power

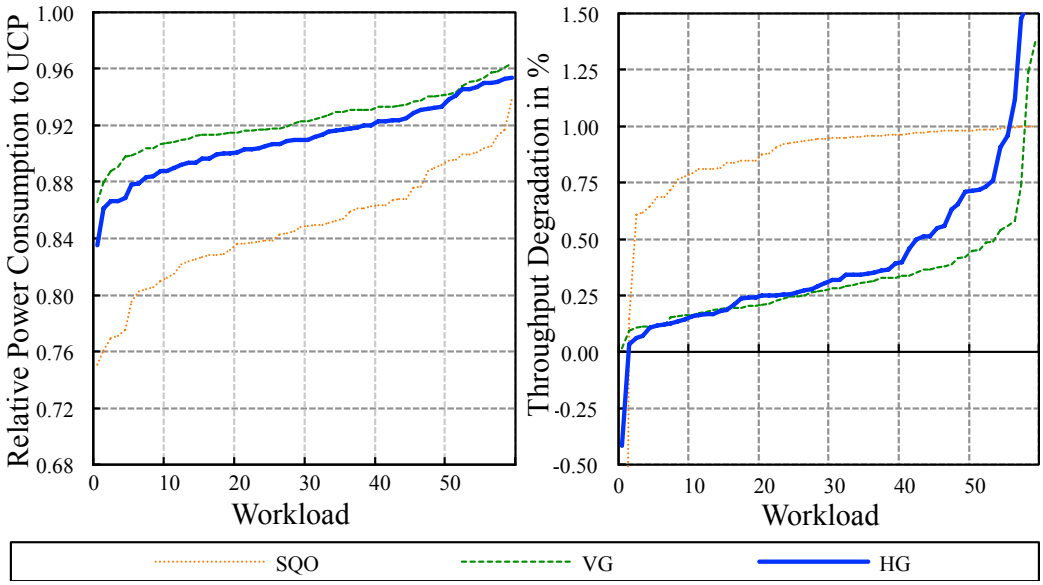


Figure 8.9: Power and system throughput comparison between static quasi-optimal (SQO), vertical grouping (VG), and horizontal grouping (HG).

savings, showing power savings of 7.6% and 9%, respectively. From the perspective of performance degradation compared to the performance level of UCP, SQO, VG, and HG show 0.8%, 0.32%, and 0.41% throughput degradation, respectively.

Discussion Our GCD-based vertical and horizontal groupings show suboptimal result compared to the result of static quasi-optimal (SQO) in Figure 8.9. There are mainly two sources of errors: power and performance approximation errors and suboptimization errors from optimizing subproblems. As we pointed out in Section 6.3.2 and 7.5.1, both AMAT-based performance approximating and way-count-based AMAT and power approximating introduce significant errors. Eliminating these errors helps us understand not only the degree of these errors, but also the limit of power savings that our approach may achieve. Likewise, we conduct a limit

study to eliminate these errors so that we can breakdown the errors that we observe in Figure 8.9 into approximation and suboptimization errors.

We eliminate errors from way-count-based AMAT and power approximating by employing our extensive simulation result in Section 4.4. We implement this ideal approach for no-grouping, vertical grouping, and horizontal grouping. Figure 8.10 shows power savings and performance degradation level of ideal no-grouping (I-NG), ideal vertical grouping (I-VG), and ideal horizontal grouping (I-HG).

I-NG, I-VG, and I-HG exhibit the total system power consumption of 85.5%, 87.5%, and 85.7%, compared to the total system power consumption of UCP, respectively. Comparing to the power savings of our static quasi-optimal (SQO) that exhibit the total system power consumption of 84.8% (15.2% power savings), I-NG, I-VG, and I-HG achieve as much as 95%, 82% and 94% in power savings, showing power savings of 14.5%, 12.5%, and 14.3%, respectively. It is notable that I-HG achieves almost same power savings to the power savings of I-NG while reducing the convergence rate significantly as we showed earlier in Section 8.2. Based on the power savings result, we can conclude that the quality of the solution from our GCD approach is as good as 95% of the SQO for I-NG. Moreover, I-HG finds the solutions of the almost same quality compared to I-NG.

From the perspective of the performance degradation level, although I-NG, I-VG, and I-HG exhibit 1.23%, 1.01%, and 1.15% performance degradation level, on average, all these approaches show more than 2% performance degradation for some cases as show in Figure 8.10. The main reason is that we use *loosely coupled performance bound* we explained in Section 8.3.1. This relaxed performance control

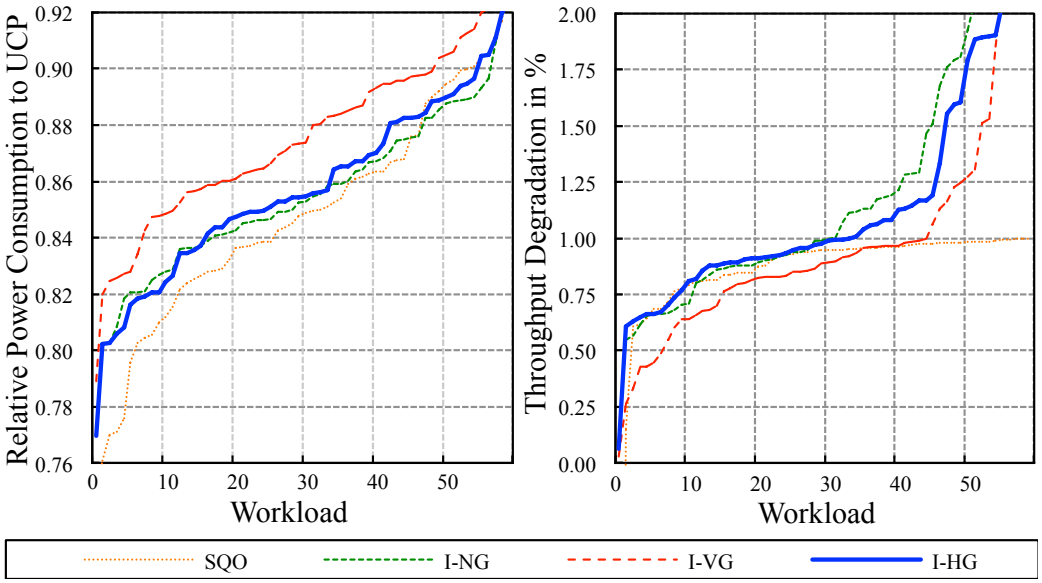


Figure 8.10: Power and system throughput comparison between static quasi-optimal (SQO), ideal vertical grouping (I-VG), and ideal horizontal grouping (I-HG).

enables a global search over the solution space, but results in worse performance degradation level than the level by which we want a performance degradation level is bound.

8.4.2 Online Analysis

We evaluate the power and performance of dynamic vertical grouping (D-VG), and dynamic horizontal grouping (D-HG) as shown in Figure 8.11. All values are normalized to the result of UCP. D-VG and D-HG exhibit the total system power consumption of 86.8% and 86.1%, compared to the total system power consumption of UCP, respectively. Comparing to the power savings of our static quasi-optimal (SQO) that exhibit the total system power consumption of 84.8% (15.2% power savings), D-VG, and D-HG achieve as much as 87% and 91% in power savings,

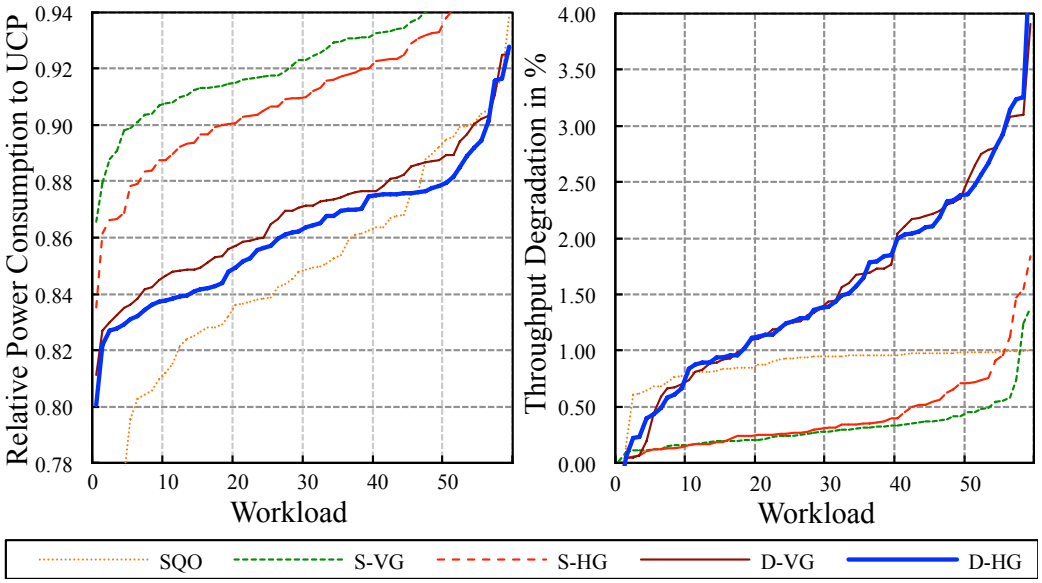


Figure 8.11: Power and system throughput comparison between static quasi-optimal (SQO), static vertical grouping (S-VG), static horizontal grouping (S-HG), dynamic vertical grouping (D-VG), and dynamic horizontal grouping (D-HG).

showing power savings of 13.2% and 13.9%, respectively. Comparing to the power savings of static implementation in the offline study (7.6% and 9% for S-VG and S-VG), D-VG and D-HG achieve 5.6% and 4.9% more power savings, respectively. It is also notable that the difference between power savings of VG and HG, 0.7%, for this online study is smaller than the difference, 1.4%, for the offline study. From the perspective of power savings, D-VG and D-HG does not exhibit significant difference. However, HG compares PEG values across cores and has better chance to explore towards the global optimal. Although, the average power savings are similar between two different grouping manners, HG shows better power savings with more cores. As shown in Table 8.5, we compare differences between two approaches per workload groups. Comparing the normalized difference (difference divided by power savings

	D-VG	D-HG	Diff.	Normalized Diff .
G2	0.870	0.866	0.004	0.028
G4	0.869	0.862	0.007	0.052
G8	0.872	0.863	0.009	0.066

Table 8.5: Performance per workload groups. Normalized difference is calculated by Diff/(1-D-HG).

of D-HG), G2/4/8 groups exhibit differences of 2.8%, 5.2%, and 6.6%, respectively. These increasing differences with more cores support that HG approximates the global optimal better than VG in that HG compares PEG vales across cores and its optimization scope expands as the number of cores grows. Our online GCD CMP MCR implementation shows 1.5% performance degradation level compared to the performance level of UCP, on average (same for both D-VG and D-HG). It is worse than the performance degradation level in the static study of 0.3% and 0.4%.

It is notable that D-VG exceeds the power savings of I-VG (13.2% vs. 12.5%) while maintaining slightly worse performance degradation level (1.5% vs. 1.01%). Likewise, D-HG shows comparable power savings to the power savings of I-DG (13.9% vs. 14.3%). Considering the additional power savings compared to the offline study and the similar power savings compare to the ideal study, we conclude that our online implementation is able to adapt to the program phase changes efficiently.

Chapter 9

Conclusion

9.1 Summary and Conclusion

In this dissertation, we propose a new greedy coordinate descent method that optimizes a CMP multi-level cache configuration without direct power/performance evaluations. Our approach predicts power and performance in the optimization, because direct evaluations may cause erroneous probing due to dynamic program phase changes. Instead of a direct power/performance feedback, we employ way counters to define *power efficiency gain* (PEG) that we use to decide search steps in the optimization. By descending according to the coordinate that has the maximum/minimum PEG value, we approximate a gradient-based optimization with *manhattan* movements over the coordinate system. We develop GCD techniques for uniprocessor MCR, LLC partitioning, and CMP MCR that efficiently search local minima. Although these techniques do not find global optimal cache configurations always, the power savings that these techniques achieve at runtime is similar to the power savings of the statically optimal cache configurations due to their dynamic adaptability.

From this study, we conclude followings. First, we find that approximated partial-derivative-based search for a coordinate descent method is effective as shown in Chapter 6. Although multi-level cache hierarchy complicates the optimization

problem with deeper caching levels, separating each caching level and searching the maximum PEG with a given performance limit is possible. This *greedy coordinate descent* effectively reduces the evaluation complexity. Second, our heuristic that approximates the partial derivative based on way counter shows feasibility for a CMP multi-level cache resizing at runtime, which is an *NP-hard* problem. Third, the PEG metric not only reduces the evaluation complexity, but also enables comparisons between different caching levels. PEG values in the same caching level enables a line search by selecting the maximum PEG value. Moreover, each maximum PEG values per caching level further enables comparisons between different caching levels which effectively removes search complexity over all possible combination of different cache sizes at different caching levels. Due to the fast convergence rate of our GCD method, we were able to achieve better power savings at runtime compared to the power savings of it static implementation. Fourth, we showed that the traditional way counters can be used for power efficiency as well.

9.2 Future Work

In this dissertation, we showed that our greedy coordinate descent method is effective to reduce wasteful power consumption from a CMP multi-level cache hierarchy. We believe that the idea presented in this dissertation can be extended in wider range of problems.

First, we can extend our study by considering task scheduling. We assumed given task scheduling to simply the CMP multi-level cache resizing problem. How-

ever, task scheduling problem to minimize the total system power consumption while maintaining high throughput is another very challenging problem. For example, if we have a set of programs to finish, task scheduling should optimize workload composition, context-switch frequency, and its implementation in OS to reduce power consumptions.

Second, we can apply our GCD approach to change number of caching levels adaptively. Cache thrashing is one of the cache interference problem caused by different working sets across threads. As such, cache bypassing which skips certain caching level of a cache hierarchy is one of techniques that can prevent such cache interference. Cache bypassing can virtually change the number of caching level. For example, a thread having a memory reference stream of a larger working set that does not efficiently utilize a last level cache can be bypassed at the LLC. Our GCD approach can be extended to dynamically reorganize the cache hierarchy to determine bypassing of a given thread at each caching level.

Third, we can apply our technique in conjunction with dynamic voltage scaling technique. Dynamic voltage scaling (DVS) technique is an architectural technique to save dynamic power consumption of CPUs and have been studied widely. DVS will eventually change computation speed and cause AMAT changes in the cache hierarchy. And this will result in non trivial performance change if they are managed separately. This is a very interesting problem in that computation and load times contribute differently across various programs and the power efficiency changes from optimal power savings from DVS and GCD CMP multi-level cache resizing are not known yet.

Fourth, we can apply our technique for distributed last level caches in a many-core system. We assume a shared LLC in this thesis, however there are other topologies in CMPs as well. For example, distributed LLCs are employed in Tilera CMPs [26]. Distributed LLCs can have an additional caching level by allowing sharing in other tile's LLC. This will complicate the cache hierarchy and extending our GCD approach can be one of the possible solutions to optimize the power consumption of such a cache hierarchy.

Bibliography

[1] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, 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.

[2] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 248–259, November 1999.

[3] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Dynamic Memory Hierarchy Performance Optimization. In *Proceedings of the workshop on Solving the Memory Wall Problem*, June 2000.

[4] Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. A Dynamically Tunable Memory Hierarchy. *IEEE Transactions on Computers*, 52(10):1243–1258, October 2003.

[5] 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 International Symposium on High Performance Computer Architecture*, 2009.

[6] 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*, Rapallo, Italy, 2000.

[7] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics & Design*, pages 90–95, 2000.

[8] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. 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 '02, pages 151–161, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Se-Hyun Yang, Michael D. Powell, Babk Falsafi, Kaushik Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[10] Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Migel P. Topham, and Bjorn Franke. Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, pages 311–322, New Orleans, LA, February 2012.

[11] 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 International Symposium on Microarchitecture*, 2006.

[12] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[13] Wanli Liu and Donald Yeung. Using aggressor thread information to improve shared cache management for cmps. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 372–383, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 213–224, Piscataway, NJ, USA, 2013. IEEE Press.

[15] Hyunjin Lee, Sangyeun Cho, and B.R. Childers. Cloudcache: Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 219–230, 2011.

[16] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. *SIGARCH Comput. Archit. News*, 38(3):419–428, June 2010.

[17] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, Seattle, WA, June 2007.

[18] Chris H. Kim and Kaushik Roy. Dynamic Vth Scaling Scheme for Active Leakage Power Reduction. In *Proceedings of the International Symposium on Design, Automation, and Test in Europe*, pages 163–167, 2002.

[19] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2002.

[20] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(7-26), 2004.

[21] Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, and Amrutur Bharadwaj. Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions. In *Proceedings of the International Symposium on Microarchitecture*, 2006.

[22] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A resource allocation model for qos management. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 298–307. IEEE, 1997.

[23] Paul Tseng. Dual coordinate ascent methods for non-strictly convex minimization. *Math. Program.*, 59:231–247, 1993.

[24] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, New York, NY, USA, 1996. ACM.

[25] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[26] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *IEEE International Solid-State Circuits Conference, 2008 (ISSCC 2008). Digest of Technical Papers.*, pages 88–598. IEEE, February 2008.

[27] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[28] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137. IEEE, 2013.

[29] Mark Horowitz. Scaling, power and the future of cmos. In *Proceedings of the 20th International Conference on VLSI Design Held Jointly with 6th International Conference: Embedded Systems*, VLSID '07, pages 23–, Washington, DC, USA, 2007. IEEE Computer Society.

[30] 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 '11, pages 365–376, New York, NY, USA, 2011. ACM.

[31] 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 Chicago, IL, 2005.

[32] Nam Sung Kim, David Blaauw, and Trevor Mudge. Leakage Power Optimization Techniques for Ultra Deep Sub-Micron Multi-Level Caches. In *Proceedings of the International Conference on Computer-Aided Design*, 2003.

[33] 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*, pages Monterey, CA, August 1998.

[34] Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[35] 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*, 12(2):167–184, February 2004.

[36] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *Proceedings of 11th Annual International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[38] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[39] Kamil Kedzierski, 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*, Saint-Malo, France, June 2010.

[40] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 948–953, New York, NY, USA, 2011. ACM.

[41] Gu-Yeon Wei and M Horowitz. A fully digital, energy-efficient, adaptive power-supply regulator. *Solid-State Circuits, IEEE Journal of*, 34(4):520–528, 1999.

[42] T D Burd, T A Pering, A J Stratakos, and R W Brodersen. A dynamic voltage scaled microprocessor system. *Solid-State Circuits, IEEE Journal of*, 35(11):1571–1580, 2000.

[43] J.W Tschanz, S.G Narendra, Y Ye, B.A Bloechel, S Borkar, and V De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *Solid-State Circuits, IEEE Journal of*, 38(11):1838–1845, 2003.

[44] C.H Kim, Jae-Joon Kim, S Mukhopadhyay, and K Roy. A forward body-biased low-leakage SRAM cache: device, circuit and architecture considerations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(3):349–357, 2005.

[45] S Mutoh, T Douseki, Y Matsuya, T Aoki, S Shigematsu, and J Yamada. 1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos. *Solid-State Circuits, IEEE Journal of*, 30(8):847–854, 1995.

[46] J.T Kao and A P Chandrakasan. Dual-threshold voltage techniques for low-power digital circuit. *Solid-State Circuits, IEEE Journal of*, 35(7):1009–1018, 2000.

[47] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An analytical model for designing memory hierarchies. *IEEE Trans. Comput.*, 45(10):1180–1194, October 1996.

[48] Abel Guilhermino Silva-Filho and Filipe Rolim Cordeiro. A Combined Optimization Method for Tuning Two-Level Memory Hierarcnhy Considering Energy Consumption. *EURASIP Journal on Embedded Systems*, 2011, September 2010.

[49] Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. Automatic Tuning of Two-Level Caches to Embedded Applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 04)*, 2004.

[50] Chuanjun Zhang and Frank Vahid. Cache Configuration Exploration on Prototyping Platforms. In *Proceedings of the 14th International Workshop on Rapid Systems Prototyping*, 2003.

[51] EmuVM. AlphaVM-free, version 1.0.2 for Windows 7. Available at. `http://www.emuvm.com/downloads.php`.

[52] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[53] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

[54] Hewlett Packard Development Company. DDR3 memory technology, Technology brief, 3rd edition. April 2012.

[55] 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 IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[56] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*, pages 3–14. IEEE Computer Society, 2007.

[57] NK Shukla, RK Singh, and M Pattanaik. Design and Analysis of a Novel Low-Power SRAM Bit-Cell Structure at Deep-Sub-Micron CMOS Technology for Mobile Multimedia Applications. *International Journal of Advanced ...*, 2011.

[58] ITRS Working Group Models, MASTAR. `http://www.itrs.net/models.html`, 2011.

[59] Dan Klein. Lagrange multipliers without permanent scarring. *University of California at Berkeley, Computer Science Division*, 2004.

[60] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[61] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[62] Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[63] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 327–, Washington, DC, USA, 2003. IEEE Computer Society.

[64] John A Nelder and Roger Mead. A simplex method for function minimization. *Computer journal*, 7(4):308–313, 1965.

[65] Stijn Eyerman and Lieven Eeckhout. Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance. *IEEE Computer Architecture Letters*, 99(RapidPosts):1, 2013.

[66] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.