SCALABLE DYNAMIC MULTI-RESOURCE ALLOCATION IN MULTICORE SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Xiaodong Wang

January 2017

SCALABLE DYNAMIC MULTI-RESOURCE ALLOCATION IN MULTICORE SYSTEMS

Xiaodong Wang, Ph.D.
Cornell University 2017

Designing chip multiprocessors (CMPs) that scale to more than a handful of cores is an important goal for the upcoming technology generations. A challenge to scalability is the fact that these cores will inevitably share hardware resources, whether it be on-chip storage, memory bandwidth, the chip's power budget, etc. Efficiently allocating those shared resources across cores is critical to optimize CMP executions. Techniques proposed in the literature often rely on global, centralized mechanisms that seek to maximize system throughput. Global optimization may hurt scalability: as more cores are integrated on a die, the search space grows exponentially, making it harder to achieve optimal or even acceptable operating points at run-time without incurring significant overheads.

In this thesis, we present *XChange* [103], a framework for scalable resource allocation in large-scale CMPs. Inspired by market mechanisms, which are widely used in real life to allocate social resources at scale, we propose to address the resource allocation problem in large-scale CMPs as a purely dynamic, largely distributed market framework. Each shared resource is assigned a virtual price, which changes over time to reflect its supply-demand relationship. Cores in the CMP act as market players: they seek to maximize their own utilities by bidding for shared resources within their budgets. Because each core works largely independently, the resource allocation becomes a scalable, mostly distributed

decision-making process. Cores in the system are able to dynamically monitor and learn their own resource-performance relationship and bid accordingly—no prior knowledge of the workload characteristics is assumed.

We show our market-based resource allocation mechanism delivers superior system efficiency and fairness against existing proposals. This approach is purely empirical, however, and thus it does not provide any guarantees on the loss of efficiency and fairness. It is well known, for example, that market mechanisms in equilibrium can sometimes be highly inefficient—this is known as *Tragedy of Commons* [44]. Therefore, we study the theoretic properties of our market-based approach, and establish a bound on the loss of efficiency and fairness in the market-based resource allocation, by introducing two new metrics, market utility range (MUR) and market budget range (MBR). Further, guided by such theoretic foundations, we propose *ReBudget* [104], a budget re-assignment technique that is able to systematically trade off efficiency and fairness in an adjustable manner.

In the process of formulating our XChange framework, we propose novel mechanisms to support fine-grain resource management. Specifically, we propose SWAP [102], a scalable and fine-grain cache management technique that seamlessly combines set and way partitioning. By cooperatively managing cache ways and sets, SWAP ("Set and WAy Partitioning") can successfully provide hundreds of fine-grained cache partitions for the manycore era. We implement SWAP as a user-space management thread on Cavium's ThunderX, a real server-grade 48-core processor, and we show that SWAP significantly improves system throughput by twice as much speedup as what we can obtain by using only ThunderX's way partitioning mechanism. This was a collaboration with Cavium engineers.

BIOGRAPHICAL SKETCH

Xiaodong Wang attended Shanghai Jiao Tong University as a undergraduate student in the department of Electronic Engineering, where he earned a B.Eng degree in 2011. He met Prof. Hsien-Hsin Sean Lee there, who motivated him to participate in an exchange program at Georgia Institute of Technology in 2010. During his time at Georgia Tech, he's inspired by doing research with Sean in the field of computer architecture and 3D memories, and therefore he decided to join the graduate program at Cornell University, pursuing a PhD degree in Computer Engineering. Through his years at Cornell, he has the opportunity to work with Prof. José Martínez as his research advisor, and focus on the scalability challenge of resource allocation in large-scale chip-multiprocessors. His paper was nominated to be the best paper candidate in HPCA 2015, and he is the winner of Cavium Octeon Trophy in 2014, and Cavium ThunderX Trophy in 2015-2016.

Dedicated to my parents, my wife, my dear friends, and everyone else who believed in me along the way.

ACKNOWLEDGEMENTS

Completing my PhD degree is one of the most challenging tasks in my life so far. Looking back over the past six years of my PhD journey, I feel extremely grateful to have so many people help me along the way. What makes me proud is not only the projects that I've done, but also the moments that I share with my family, my mentors, my friends, and my colleagues, that I'll always cherish for the rest of my life.

I should start by thanking my parents. Words simply cannot express my gratitude to their sacrifices. My father not only works tirelessly to support me financially, but also advises me on any issues that I encounter. He encourages me to think out of the box and make my own decisions, but he also pulls me back to the right track whenever necessarily. My mother dedicated her life for taking care of me. I wasn't a strong kid, both physically and mentally. She taught me to be a fighter, which is the characteristic that I attribute most of my success to. I am truly proud of my parents, and I hope I've made you proud of me too.

I want to thank my wife, Jia. Living and studying in a place thousands of miles away from your family and childhood friends is never an easy job. I am so lucky to meet you, Jia, here at Cornell. Because of you, my apartment is longer a "sleep area", but a warm home that we enjoy family life. PhD is a long journey, and I cannot believe that I will survive without you. I'm so happy to have gone through my PhD career with you, and to continue our life-long journey.

My PhD work would not have been possible without my advisor, Prof. José Martínez. There are numerous of times in my PhD that I become frustrated, either by experimental infrastructures, simulation results, paper reviews, etc. He always motivates me to get out of the swamp, and to be keen and excited again about my research projects. Whenever I met with any difficulties, I know he's always there and ready to help. We have brainstormed and debated about countless of ideas, and worked together days and nights for a dozen of deadlines. I learned from him not only technically, but also in the aspect of a human being and a good researcher. In addition, he is never refrained from offering me his most frank opinions, whether it's technical, professional, or personal. I'm deeply grateful for all his effort and great contributions.

I would like to thank my committee members, Prof. David Albonesi and Prof. Emin Sirer, for their insightful suggestions and advices. Whenever I have any questions or need a second opinion of my ideas, they're the ones that I can always rely on, and they're always willing to spend their time talking to me no matter how busy they are.

I would also like to thank Prof. Éva Tardos. I wasn't a person that is particular interested in math and theories. She showed me the beauty of them, and how the equations can help guiding pragmatic approaches. My ReBudget work wouldn't have been made possible without her selfless support.

Special thanks to my undergraduate advisor, Prof. Hsien-Hsin Sean Lee. When I was considering doing a PhD, I was clueless about what research direction I should pursue. When Prof. Lee came to my undergraduate institution, and gave us a talk about computer architecture and 3D architecture, it opened up a new world to me, and I immediately decided that it is the field that I am dedicated to. During my one-semester exchange study, Prof. Lee not only taught me the computer architecture basics, but also led me into the world of research. He truly built the foundation of my PhD career.

I want to thank my co-authors, Jeff Setter, and Shuang Chen, for their great contributions to the SWAP work in this thesis. Jeff and I first came up with the SWAP idea, and we built the first prototype together with three months of hard working. Shuang joined the team one year later, and we work tirelessly to perfect this project. I am greatly honored to have the opportunity to work with both of you, and I believe you will be very successful in your career.

My fellow CSL colleagues have been fantastic in helping me completing this arduous PhD journey. Saugata Ghose is my mentor. His enthusiasm and passion in research and teaching has a great impact on me, and I admire him as if he's my elder brother. I'm also thankful to my lab mates, Janani Mukundan, Maia Kelner, and Skand Hurkat, for being with me in the same office and supporting me throughout my PhD. Special thanks to Abhinandan Majumdar and Shreesha Srinath, for our special friendship of joining CSL at the same time. I surely believe that both of you will be tremendously successful in anything you'll do. I also have been fortunate to be friends with Yao Wang, Tao Chen, Robert Karmazin, Jonanthan Tse, and all the other CSL students.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The Scalability Challenge of Resource Allocation in CMPs

Resource allocation is a major challenge for computer systems. Back in the early days when most computers are composed of a single-core CPU, applications running in the system were time-sharing CPU cycles, as well as memory capacity, disk storage, etc. Managing these resources in a fair and efficient manner is critical, and different mechanisms have been proposed. Many of them have become the norms in the computer systems we're using today [3].

With the advancement of technology generations, more transistors are brought into the CPU chip. However, the computer industry finds it harder and harder to utilize those extra transistors to scale up the single core in the CPU. This observation, along with the failure of Dennard scaling, motivates the industry to transition from the scale-up approach to the scale-out approach, where more cores are integrated into a single chip to achieve performance growth.

A problem with this scale-out approach is that the multiple cores in a chip inevitably share hardware resources, whether it be on-chip storage, memory bandwidth, the chip's power budget, etc. The computer systems that are designed for single-core CPUs leave these resources unmanaged, allowing all the cores to freely contend shared resources. Researchers have found that such unmanaged policy significantly hurts system performance, and therefore, they propose mechanisms to manage those resources independently [18, 70, 74, 79]. Although they show great performance improvements, it is found that those proposals do not maximize the system performance and resource utilization, because resource interactions exist. For example, increasing an application's allocated cache space may reduce its memory bandwidth demand, due to the lower cache miss rate. Similarly, increasing an application's power budget could allow it to run at a higher frequency, potentially demanding higher memory bandwidth. Therefore, a coordinated allocation among the shared resources is necessary for the multi-core computer systems [12].

In recent years, cores keep piling up in a single chip. Most of the mainstream processors today integrate 4 to 8 cores, and Intel's Xeon Phi has made one step further by having more than 60 cores on a chip, and is able to run more than 240 threads simultaneously by enabling 4-way SMT [27]. Therefore, designing chip multiprocessor (CMP) systems that scale to many tens or even hundreds cores is an important goal for the upcoming technology generations. With more cores in the system, the contention for shared resources becomes worse, and therefore, a key challenge to scalability is to effectively manage shared resources among the competing cores. However, optimally partitioning CMP resources is not easy. Beckmann and Sanchez find it a NP-complete problem to just partition the shared last-level cache due to its non-concave behavior [8]. It becomes even worse for the coordinated multi-resource allocation problems, due to the interactions among the resources. Most existing proposals for on-chip resource management take a centralized approach [12, 23, 26], where an arbiter allocates resources globally using heuristics such as global hill-climbing. Unfortunately, this approach is not scalable to large-scale CMPs: First, fine-grained resource management has been shown to be highly desirable to optimize system performance, but the potential number of operating points can be large, making it time-consuming for global optimization to find the optimal allocation point.

Second, resource allocation is generally not a problem separable by resource, as important resource interactions exist. In all, the search space of global resource allocation scales exponentially with the number of cores, the number of resources, and the granularity of resources. Therefore, the global resource arbiter will either be able to explore only a small fraction of the search space, or take too long to reach an optimal decision.

1.1.1 A Market-based Approach

To address the scalability challenge of the centralized approach, we propose XChange, a novel CMP multi-resource allocation mechanism that is able to deliver scalable high throughput and fairness. We formulate the problem as a purely dynamic, largely distributed market, where the "prices" of resources are adjusted based on supply and demand. Cores dynamically learn their own resource-performance relationship and bid accordingly; no prior knowledge of the workload characteristics is assumed.

Our evaluation shows that, using detailed simulations of a 64-core CMP configuration running a variety of multiprogrammed workloads, the proposed XChange mechanism improves system throughput (weighted speedup) by about 21% on average, and fairness (harmonic speedup) by about 24% on average, compared with equal-share on-chip cache and power distribution. On both metrics, that is at least about twice as much improvement over equal-share as a state-of-the-art centralized allocation scheme [23]. Furthermore, our results show that XChange is significantly more scalable than the state-of-the-art centralized allocation scheme we compare against: less than 0.5% overhead on a

5-million-cycle allocation interval (approx. 1 ms) to reach an allocation decision, for CMP sizes anywhere from four to 128 cores. In contrast, the state-of-the-art centralized scheme we compare against takes over 30% of the allocation interval to converge under a 64-core CMP, and it exceeds the entire interval beyond 100 cores.

Although inspired by theoretical studies [40, 75, 105, 113], XChange is nevertheless heuristic by design. We present a comparison against a recently proposed, formally provable market-based resource allocation mechanism [111], and show that our heuristic approach delivers superior throughput across the board for the configurations and workloads studied.

1.1.2 Theoretic Guarantees in System Efficiency and Fairness

Although XChange delivers superior system efficiency and fairness against existing proposals it is purely empirical, however, and thus it does not provide any guarantees on the loss of efficiency and fairness. It is well known, for example, that market mechanisms in equilibrium can sometimes be highly inefficient—this is known as *Tragedy of Commons* [44].

Our contributions are as follows:

— We introduce a new *Market Utility Range (MUR)* metric, which helps us establish a theoretical bound for efficiency loss of a market equilibrium under a constrained budget. Specifically, we show that, if $\text{MUR} \geq 0.5$, then $\text{PoA} \geq (1 - \frac{1}{4\text{MUR}}) \geq 0.5$ (i.e., the efficiency is guaranteed to be at least 50% of the optimal allocation); and that if $\text{MUR} < 0.5$, then $\text{PoA} \geq \text{MUR}$.

— We introduce a new *Market Budget Range (MBR)* metric, which helps us evaluate the fairness of a market equilibrium under a constrained budget. We show that any market equilibrium is $(2\sqrt{1+\text{MBR}}-2)$-approximate envy-free.

— We propose *ReBudget*, a budget re-assignment technique that is able to systematically control efficiency and fairness in an adjustable manner. We evaluate ReBudget on top of XChange, using a detailed simulation of a multicore architecture running a variety of applications. Our results show that ReBudget is efficient and effective. In particular, it can achieve 95% of the maximum feasible efficiency. Furthermore, when combined with the analysis using MUR and MBR metrics, it can provide worst-case fairness guarantees.

1.2 Fine-Grain Cache Management

Our market-based approach relies on hardware and software support for shared resource partition. Although fine-grained power partition has already been supported by commercial chips (Intel's RAPL technique [51]), fine-grained cache partition remains to be a hard problem, especially for large-scale CMPs.

Two popular cache partition approaches are *(a)* hardware support for way partitioning, or *(b)* operating system support for set partitioning through page coloring. Way partitioning allows cores in chip multiprocessors (CMPs) to divvy up the shared cache space, where each core is allowed to allocate cache lines in only a subset of the cache ways. It is a commonly proposed approach to curbing cache interference across applications in chip multiprocessors (CMPs) [62, 79]. Unfortunately, way partitioning is proving to be not particularly scalable, as it affects cache latency and power negatively, eventually becoming impractical. Page coloring, on the other hand, achieves cache partitioning by restricting each application's page frames to certain "colors" (the shared bits between a physical address' page frame ID and cache index). In this case, page frames of each color map onto a specific subset of the cache sets. Although this approach has been adopted in real operating systems [62,64,109], it also does not scale beyond a handful of colors.

We propose SWAP [102], a fine-grained cache partitioning mechanism that can be readily implemented in existing CMP systems. By cooperatively combining the cache way (hardware) and set (OS) partitioning, SWAP is able to divide the shared cache into literally hundreds of regions, therefore providing sufficiently fine granularity for the upcoming manycore processor generation.

We implement SWAP as a user-space management thread on Cavium's ThunderX, a server-grade 48-core processor with ARM-v8 ISA [97]. To enable SWAP, we introduce small changes to the Linux page allocator, and leverage ThunderX's native architectural support for way partitioning.

Our results show that SWAP improves system throughput (weighted speedup) by 13.9%, 14.1%, 12.5% and 12.5% on average for 16-, 24-, 32- and 48- application bundles with respect to no cache management. This is twice as much speedup as what we can obtain by using only ThunderX's way partitioning mechanism.

To our knowledge, SWAP is the first proposal of a fine-grained cache partitioning technique that requires no more hardware than what's already present in commercial server-grade CMPs.

CHAPTER 2

XCHANGE: SCALABLE RESOURCE ALLOCATION IN CMPS

2.1 Introduction

Designing chip multiprocessors (CMPs) that scale to more than a handful of cores is an important goal for the upcoming technology generations. A challenge to scalability is the fact that these cores will inevitably share hardware resources, whether it be on-chip storage, memory bandwidth, the chip's power budget, etc. Studies have shown that allowing cores to freely contend for shared resources can harm system performance [12, 23, 26]. Therefore, allocating resources efficiently among cores is key to achieving good behavior.

One challenge in resource allocation is that it is generally not a problem separable by resource, as resource interactions exist [12]. For example, increasing an application's allocated cache space may reduce its memory bandwidth demand, due to the lower cache miss rate. Similarly, increasing an application's power budget could allow it to run at a higher frequency, potentially demanding higher memory bandwidth. As more and more cores are integrated on a single die, the size of this multi-resource allocation space explodes, making it harder to devise mechanisms to lock on a good allocation without incurring significant overheads. Although prior knowledge of the applications from offline profiling may curb some of the run-time overhead, this information is generally not available.

An additional important consideration is the balance between throughput and fairness. Eyerman and Eeckhout [38] argue that a good resource allocation scheme should be able to maintain a balance between single-program performance and overall system throughput. However, existing global optimization solutions deal primarily with system throughput [23, 26], potentially resulting in systems with poor fairness. On the other hand, a recently proposed proportional allocation technique by Zahedi and Lee focuses on guaranteeing strict game-theoretic fairness of the co-running applications [111], but its formulation may come in practice at a cost in throughput, as we show in our results.

Contributions

In this Chapter, we propose XChange, a novel CMP multi-resource allocation mechanism that is able to deliver scalable high throughput and fairness. We formulate the problem as a purely dynamic, largely distributed market, where the "prices" of resources are adjusted based on supply and demand. Cores dynamically learn their own resource-performance relationship and bid accordingly; no prior knowledge of the workload characteristics is assumed.

Our evaluation shows that, using detailed simulations of a 64-core CMP configuration running a variety of multiprogrammed workloads, the proposed XChange mechanism improves system throughput (weighted speedup) by about 21% on average, and fairness (harmonic speedup) by about 24% on average, compared with equal-share on-chip cache and power distribution. On both metrics, that is at least about twice as much improvement over equal-share as a state-of-the-art centralized allocation scheme [23]. Furthermore, our results show that XChange is significantly more scalable than the state-of-the-art centralized allocation scheme we compare against: less than 0.5% overhead on a

5-million-cycle allocation interval (approx. 1 ms) to reach an allocation decision, for CMP sizes anywhere from four to 128 cores. In contrast, the state-of-the-art centralized scheme we compare against takes over 30% of the allocation interval to converge under a 64-core CMP, and it exceeds the entire interval beyond 100 cores.

Although inspired by theoretical studies [40, 75, 105, 113], XChange is nevertheless heuristic by design. We present a comparison against a recently proposed, formally provable market-based resource allocation mechanism [111], and show that our heuristic approach delivers superior throughput across the board for the configurations and workloads studied.

This chapter is organized as follows: Section 2.2 motivates our approach in contrast to existing art. Section 2.3 describes the general market framework that XChange is based on. Section 2.4 and Section 2.5 present the implementation of our proposed mechanism. Section 2.7 evaluates our proposal. Section 2.8 validates our model. Section 2.9 shows the scalability of our mechanism.

2.2 Motivation of Our Approach

In the context of resource allocation of CMPs, researchers have shown that using fine-grained management of the available resources to provide optimized utilization is highly desirable as well as practical, and we discuss these proposals in the related work in Section 5.1. The obvious downside of fine-grained resource allocation in large-scale CMPs is that the number of potential operating points can be large, making it more time-consuming to search for optimal allocation points.

When it comes to multi-resource allocation, uncoordinated solutions have been shown to be inefficient, even inferior to static equal-share partitioning, due to their inability to model the interactions among resources [12]. A few solutions have been proposed to address the fine-grained multi-resource allocation problem [6, 12, 23, 26, 59], which are primarily based on centralized mechanisms that seek to optimize system throughput by essentially exploring the resource allocation space sequentially. Unfortunately, coordinated multi-resource allocation dramatically increases the size of the this allocation space. As we will show in Section 2.9, centralized approaches are likely to be unfeasible for large-scale CMPs, as they may take too long to discover an optimized operating point that can be exploited effectively. Moreover, many of these techniques focus on throughput, with less concern for fairness.

Our proposed XChange solution tackles scalability by adopting a market-based approach. In a market-based approach, participants seek to optimize their resource assignment largely independently of each other, and participants' demands are reconciled through a pricing mechanism. Under relatively weak conditions (e.g., resources are priced equally for all participants at each point in time), such competitive markets can converge iteratively to a Pareto-efficient equilibrium (i.e., no further trading is mutually beneficial) [69]. These two properties, namely largely distributed operation and Pareto-efficient equilibrium, make a market approach potentially attractive in our context.

XChange is different from other existing market-based resource allocation approaches [22, 42], which employ a "static market" view to allocate a single resource (compute service units): Users volunteer the amount of money each is willing to pay as a function of allocated service units. The central market then allocates the available computing resources so that monetary profit is maximized. This static view of a market is not useful in our context: To accomplish efficient multi-resource allocation, users should be able to adjust their bids dynamically in response to the perceived global resource contention—what is called the “price discovery” process. For example, a user can lower its bid for what turns out to be a highly contended resource (e.g., cache space) and bet on a different resource more heavily (e.g., power budget), if it concludes from supply-demand dynamics that it will get a better “bang for the buck.”

In addition, overall throughput is not the only concern to resource allocators: A measure of fairness is also highly desirable (e.g., to provide QoS). A recent proposal by Zahedi and Lee [111] applies an “elasticity-proportional” (EP) CMP resource allocation mechanism to accomplish game-theoretic fairness. Users' true resource utility is profiled, and the resulting profiles are curve-fitted to a log-linear function. The EP allocation mechanism uses these curve-fitted utility functions to provide an allocation with strong game-theoretic fairness guarantees, such as sharing incentives, envy freedom, and Pareto efficiency. However, guaranteeing game-theoretic fairness comes at a cost in system performance, and Zahedi and Lee's results indeed show that a fundamental trade-off exists between EP's game-theoretic fairness and achievable system throughput.

Our approach distances itself from pursuing provable game-theoretic guarantees, instead focusing on heuristics that can be practical and yield satisfactory levels of both throughput and fairness. We do not confine user behavior to a curved-fitted model, and hypothesize that a heuristic-based approximation of utility in the CMP, coupled with a fail-safe mechanism for outliers, should be sufficient to provide good outcomes. Intuitively, this is based in part on the fact that resource utility in CMPs, even if nonlinear, is monotonic (i.e., more of a resource yields equal or greater benefit)—a property present in many problems for which market-based solutions have been successful.[1] We measure throughput and fairness using metrics more conventionally found in the computer architecture community, namely weighted speedups and harmonic speedups.

Another limitation of Zahedi and Lee's approach [111] is that, although there seems to be no fundamental reason why the utility profiles could not be derived online somehow, its evaluation is based on profiles obtained offline and a priori. While it may be possible in data centers to profile an application before dispatching it [30, 68], we believe this assumption is unrealistic for general CMP-based systems. XChange approximates resource utilities dynamically at run-time—no prior knowledge of the workload's behavior is necessary.

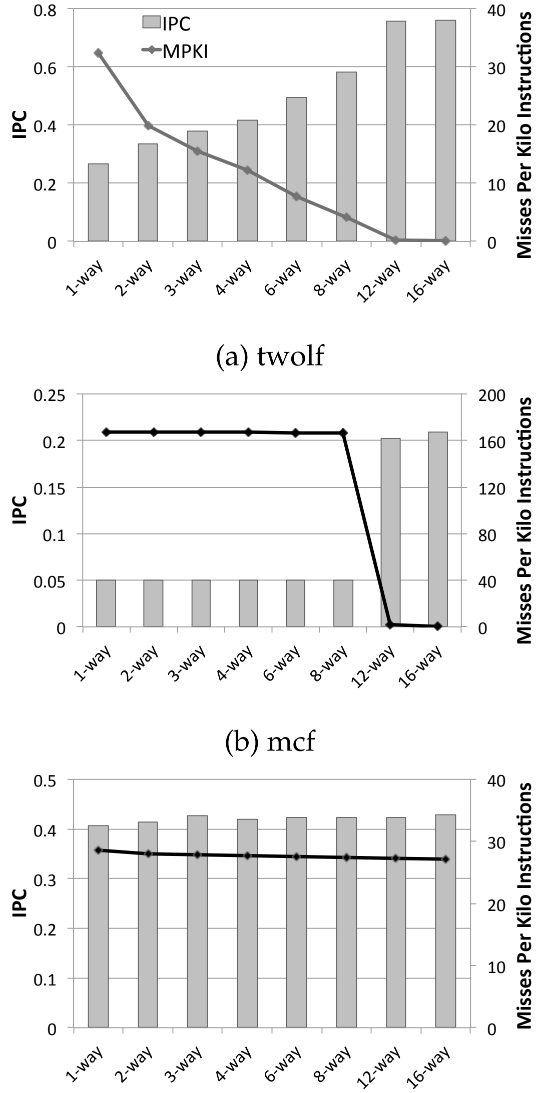
2.3 Market-Based Framework

Proper multi-resource allocation for CMPs presents the challenge of optimizing and balancing two system objectives, system throughput and fairness, as well as dealing with an allocation space which grows rapidly with the number of cores. To be truly practical, it also needs to be capable of building a resource-performance model dynamically at run-time, without the assistance of profiling or other sort of prior knowledge.

In this section, we describe the general market framework XChange is based on. We define the agents in the market as the applications running on the CMP

[1] Technically, it is possible that some resources may exhibit some kind of Bélády's anomaly, where a slightly increased resource allocation actually hurts performance in certain cases. We did not find this to be an issue in our experiments.

cores. We consider the shared resources to be the chip's power budget and the last-level cache space. We regard the scheduling question of what apps to run on the available cores as orthogonal to our objective—after all, an agent would not spend any "money" on resources if it didn't get to run.



(a) twolf

(b) mcf

(c) soplex

Figure 2.1: IPC and cache miss rate under different cache allocation, running at highest possible frequency. The x axis is the number of cache ways enabled. Section 2.6 describes our experimental setup.

2.3.1 Overview

XChange operates as a market, where each processor in the CMP can "purchase" shared resources from the system. The pricing mechanism plays a central role in the market: it conveys supply and demand information, reflects the true value of the resources, and ultimately determines who gets how much of each resource [69].

We adopt the price-taking mechanism proposed by Kelly [55]: Assume R_j represents the total amount of resource j available, and b_{ij} is the amount of money agent i bids for resource j. Then p_j, the price of resource j, is computed as the total amount of money bid by all the agents on that resource, divided by the number of resource units available:

$$p_j = \frac{\sum_i b_{ij}}{R_j} \tag{2.1}$$

The resources are then distributed proportionally to the bids each agent submits:

$$r_{ij} = \frac{b_{ij}}{p_j} \tag{2.2}$$

Here r_{ij} is the amount of resource j allocated to agent i. Note that, because of the price-taking formulation, no part of the resource is left unallocated.

At each point in time, because resource prices are readily available to agents, the agents know exactly how much of a resource they would get given the bids they place for it, and therefore, these selfish agents are able to bid optimally to maximize their own utilities. During the bidding process, the prices will fluctuate. When prices become stable because agents have no incentive to change their bids to improve their utilities, the market has converged. This results in a *Pareto-efficient* resource allocation.

In addition, to ensure market fairness, each agent is assigned a finite budget, and the total amount of its bids cannot exceed that budget. We also do not allow agents to save money: any unused budget is forfeited if the agents do not use it. This is simpler than a situation where agents are allowed to save money to later try to monopolize all the resources, which probably hurts both convergence speed and fairness.

The entire bidding process is described as follows:

1. Initially, each agent builds its local utility function—i.e., its resource-performance relationship model. It is also assigned a budget to buy resources. Meanwhile, a global resource arbiter posts the initial prices for all resources to all agents. Under such prices, each agent places bids to buy these resources. These bids are such that they maximize the agent's local utility.
2. After all agents have placed their bids, a global resource arbiter collects the bids, and adjusts the prices of the resources based on Equation 2.1. The price of highly sought-after resources will be increased, and the price of unpopular resources will be lowered to promote sales. This is a quick process.
3. The resource arbiter posts the updated prices to all agents, who then bid again under the new pricing. This process repeats itself until the market converges—i.e., the price remains stable across iterations (within 1%), and the agents have no incentive to change their bids to improve their local utility.

(We discuss more about convergence criteria in Section 2.4.3.) Finally, the resources are allocated as shown in Equation 2.2.

Prices play a key role here, as a reflection of overall system demand vs. supply. In other global optimization mechanisms for CMPs, only the *marginal* utility of each resource (i.e., the preference for that resource) is considered by the agents [23], regardless of whether it is highly contended or not. In our market, for example, if the price for resource A is high due to demand, an agent will start bidding more money on a cheaper resource B, *even though its marginal utility for resource A may be higher*, in an attempt to maximize its utility given the supply-demand circumstances.

Another major advantage of this process is that it is mostly done in a decentralized manner. Indeed, a key aspect of our market framework is that it takes advantage of individual wisdom: It allows the agents in the market to submit bids to maximize their local utility under the current resource prices, rather than submitting their utility function to a centralized entity that then performs a global search. Compared to prior centralized schemes proposed [12, 23], this process delegates the search effort to each individual agent. The only centralized work done in the system is the pricing mechanism shown in Equation 2.1, which is fairly simple and can be done efficiently. In addition, the overhead of collecting bids and posting prices is small.

One other interesting aspect of this market-based approach is that the tradeoff between system throughput and fairness can be adjusted by assigning different budgets to different agents: Intuitively, if the system prefers higher throughput, it opts to assign higher budget to the agent with higher marginal utility; if the system prefers fair allocation across agents, it opts to assign equal budget.

We discuss this issue further later in Section 2.4.3.

Challenges of building the utility model

The market model requires each agent to construct an accurate relationship between performance and resources—i.e., its local utility model. One possible solution could be to ask programmers or users to provide some "hint" to the on-chip agent about the dynamic behavior of the application. However, in general, programmers and users may provide the wrong incentive, primarily because they may be unaware of the hardware details. Profiling is an option, but this may not be feasible in practice, and in any case, the applications will be running in a different environment when it matters: different architectural configuration, competition with applications with different characteristics, etc. Notwithstanding these options, we propose to design an intelligent run-time monitoring mechanism, whose goal is to determine each agent's local utility model dynamically (and concurrently).

2.4 Mechanism: Market Participants

In this section, we describe how each individual agent dynamically models its relationship between performance and resources, and its bidding strategy in reaction to the resource price under our market framework.

All other things being equal, a simpler model is usually preferable. Here we briefly discuss a linear model to provide an intuition of how market participants operate in general. As we will see shortly, when managing CMP resources the reality is not so simple (but it is manageable).

$$u_i = \sum_j w_{ij} \times r_{ij} \tag{2.3}$$

where w_{ij} represents agent i's marginal utility for resource j, r_{ij} represents the allocation of resource j to agent i, and u_i represents the overall utility of the allocation of resources for agent i. This linear utility function, where each agent's marginal utility for each resource is constant under all circumstances, is quite simple, and many bidding strategies have been proposed in the literature. For example, PR-dynamics, which is a bidding strategy based on the linear utility model, is theoretically proven by Zhang to guarantee fast market convergence, Pareto efficiency, and also game-theoretic fairness [113].

When we attempted to use a linear market by curve-fitting the utility functions of each application, and conducted a PR-dynamics-like market, the results were poor. In order to examine why the linear utility model is a poor fit to our specific CMP resource allocation problem, we profile a few applications with varying cache capacity. (See Section 2.6 for details on the experimental setup.) Figure 2.1 shows the L2 cache miss rate (MPKI) of three representative applications under different cache way allocation, and their corresponding IPC. All the applications run at the same frequency. We find that the cache-performance behavior of *soplex* and *twolf* can fit into the linear model pretty well: *soplex* doesn't benefit from more L2 (flat curve), and *twolf*'s IPC increases almost linearly with more cache ways.

However, *mcf* shows a step function in IPC and cache capacity: once it secures 12 ways (1.5MB), its working set can fit into the cache, and its miss rate drops to almost zero, showing a sudden 200% performance increase. Such IPC-cache utility curve does not fit a linear utility curve well—in fact, it is not even convex. The existing literature does not provide easy game-theoretic guarantees for agents that behave like *mcf*. We empirically observed that some other applications have similar behavior (admittedly, *mcf* is a bit extreme).

In XChange, we choose to abstain altogether from pursuing approaches with strong game-theory guarantees. Our approach is inspired by the fact that the First Welfare Theorem has relatively weak requirements to guarantee that any market equilibrium is Pareto-efficient: XChange is a market where agents are price-takers by design (i.e., they must accept the prices imposed by the market at each point in time); agents in XChange exhibit monotonic utility (i.e., more of a resource is better) by the nature of our problem; and agents in XChange always put forward their best bid (the one that they believe maximizes their utility given the current prices).

In the rest of this section we describe how we model XChange's utility function, in part by borrowing and combining successful hardware estimation mechanisms from the existing literature. As our evaluation will show, the model yields very good results for the configurations and workloads studied (Section 2.7).[2]

2.4.1 Utility Model

Existing literature frequently characterizes workload behavior by dividing its total execution time into memory phase (core stalled waiting for memory) and

[2] The First Welfare Theorem states *sufficient* conditions for Pareto-efficiency under any market equilibrium. Thus, even market equilibria that do not strictly abide by such conditions may in principle be Pareto- or quasi-Pareto-efficient.

compute phase. In XChange, we borrow this simple compute-vs.-memory classification to characterize the impact of each of the shared resources on application behavior. Cache and off-chip bandwidth are mostly related to the length of the memory phase: a larger L2 allocation will lower the cache miss rate, while more memory bandwidth will mitigate the penalty of cache misses. At the same time, a higher power budget allocation will allow a core to run at a higher frequency, and thus the compute phase will be scaled down proportionally [18,70].

We define the agent's utility as the workload's execution time, i.e., the sum of its compute and memory phase, measured in cycles at nominal frequency (4GHz in our setup). We approximate compute and memory phases as being relatively independent—e.g., changing the core's power allocation should not affect the wait in the memory system due to cache misses and bandwidth traffic. This is of course a simplification: a lower clock frequency at the core, for example, *will* make the core issue memory requests at a slower speed, and thus affect the effective memory level parallelism, and the length of memory phase. However, it allows for simpler and faster models that, as our results will show, do deliver solid gains.

Cache Utility

We now describe how we derive our utility model for shared cache allocation, by combining two existing performance estimation mechanisms. Miftakhutdinov et al. develop a model to estimate the execution time of a program's memory phase [70]. In the model, a per-core memory critical path counter CP_{global} is maintained. When a memory request leaves the core and gets into the last-level cache, it copies CP_{global} to the its own counter CP_{local}. After Δt

cycles, the memory request is served, and the critical path counter is set as $CP_{\text{global}} = \max(CP_{\text{global}}, CP_{\text{local}} + \Delta t)$. The value of counter CP_{global} reflects the length of the memory phase. (More details can be found in that paper.)

Unfortunately, this scheme is only able to estimate the length of memory phase under the current cache and bandwidth allocations. To estimate the cache's marginal utility, we need to be able to calculate the effect that a change in cache allocation has on the memory phase. Qureshi and Patt's UMON sampled cache tag array [79] can be used to predict the cache miss rate under all possible cache allocations, although is not able to directly predict the length of memory phase.

Thus, we extend the technique developed by Miftakhutdinov et al. by incorporating UMON. The simplifying assumption we make is that memory-level parallelism (MLP) doesn't change with different cache allocations, and therefore can be computed by dividing the aggregate service time by the length of the memory phase:

$$MLP = \frac{N_h \times t_h + N_m \times t_m}{CP_{\text{global}}} \tag{2.4}$$

where N_h and N_m are the number of hits and misses under the current cache allocation, respectively, and t_h and t_m are the hit and miss latencies, respectively.

In order to predict the length of memory phase under j cache ways, $CP_{\text{global}}(j)$, we compute the aggregate memory service time without MLP, by using the prediction of the number of hits and misses, $N_h(j)$ and $N_m(j)$ respectively, from UMON.[3] With MLP, the length of memory phase under j cache ways can

[3] UMON with dynamic set sampling (DSS) is only able to predict miss rate, but we can multiply miss rate by the

be computed as:

$$CP_{\text{global}}(j) = \frac{N_h(j) \times t_h + N_m(j) \times t_m}{MLP} \tag{2.5}$$

Therefore, assuming an agent is allocated i ways of cache, if it is given one less cache way, the increase in its execution time can be computed as:

$$MU_{\text{cache}}(i) = CP_{\text{global}}(i-1) - CP_{\text{global}}(i) \tag{2.6}$$

We define $MU_{\text{cache}}(i)$ as the agent's marginal utility for cache at i ways. (Note that lower CP_{global} is better, thus the order of the operands.)

Power Utility

The agent's marginal utility for power can be modeled based on the fact that the length of the compute phase tends to scale linearly with the processor frequency. By reading the appropriate hardware performance counters, the agent can collect the statistics from the last interval: length of compute phase t_{exe}, average operating frequency f_0, energy consumption E_0, and operating voltage V_0. Then the length of compute phase $t_{\text{exe}}(f)$ under new frequency f is:

$$t_{\text{exe}}(f) = t_{\text{exe}} \times \frac{f_0}{f} \tag{2.7}$$

The power is estimated as follows:

$$P(f) = \frac{\frac{E_0}{V_0^2} \times V_f^2}{t_{\text{exe}}(f) + t_{\text{mem}}} \tag{2.8}$$

total number of cache accesses to obtain the number of hits and misses.

Here, V_f is the voltage under new frequency f, and t_{mem} is the length of memory phase under current cache partition.

Therefore, assume an agent is operating at frequency f, we define its marginal utility for power as follows:

$$MU_{\text{power}}(f + \Delta f) = \frac{t_{\text{exe}}(f) - t_{\text{exe}}(f + \Delta f)}{P(f + \Delta f) - P(f)} \tag{2.9}$$

where Δf is the frequency increment of one DVFS step.

Bandwidth Utility

The marginal utility for memory bandwidth could be derived similarly, by taking effective memory latency into account when computing the length of memory phase; however in this chapter for simplicity we assume an equal-share distribution across cores. This allows us to compare in the evaluation directly against state-of-the-art schemes for multi-resource allocation, which also allocate shared cache and power budget simultaneously [23]. Note that other resources, such as on-chip network bandwidth, can also be plugged into our utility model, as long as their resource-performance relationship can be accurately modeled.

2.4.2 Bidding Strategy

Agents in XChange conduct a simple local hill-climbing algorithm to find their optimal bids. Because the agent is working locally and independently, the complexity of this local hill-climbing does not increase with the number of cores. Notice that hill-climbing cannot generally guarantee the optimality of the solution, and thus the sufficient conditions of the First Welfare Theorem cannot be formally guaranteed. Nevertheless, again our experiments show that the bids produced in this way are of good quality.

We have established earlier in the paper that cache utility is generally not convex. This may cause hill-climbing to get stuck at a local optimum. For example, as is shown in Figure 2.1b, *mcf*'s marginal utility on allocating more cache ways is zero almost everywhere except for one point (8 to 12 ways). If the hill-climbing search starts by purchasing one cache way and the power consumption of the minimum possible frequency (800 MHz), it is almost guaranteed that the agent will bid heavily on power, because the marginal utility on cache is virtually zero in that region.

On the other hand, there is generally no such "knee" in the performance response to frequency (and thus power) changes; plus, its marginal utility diminishes as frequency increases, because power scales cubically with frequency, while compute time scales linearly (see Equation 2.9). These two "opposing" but otherwise monotonic behaviors enable us to design a "guided" hill-climbing search, which starts searching from the maximum affordable cache. Thus, our proposed local hill-climbing algorithm works as follows:

1. The price of resources is broadcast to the agents.
2. Each agent starts by purchasing its bare-minimum power (assuming the core is operating at 800 MHz), and leaves all the remaining budget to cache.

3. The agent decreases its cache bid by one way, and uses the saved money to buy extra power. By comparing the marginal utilities, the agent can decide whether the trade is worthwhile. If so, it accepts the trade and this step is repeated; otherwise, the agent denies the trade, and it submits the current bids to the market.

This algorithm should deal with *mcf*'s "step-like" cache behavior very well. In Figure 2.1b, the step of *mcf* is at 12 ways. Our hill-climbing starts from the maximum affordable cache. Suppose *mcf* can at most buy 10 cache ways because it is highly contended; then the algorithm will eventually end up trading 9 cache ways for power, and will never really worry about climbing up to 12 cache ways, simply because it is not affordable. On the contrary, if *mcf* can afford more than 12 cache ways, it will iterate as described above to decide, at each point, whether cache or power is more beneficial.

Our *guided* hill-climbing approach is efficient, because it walks through the search space linearly. However, this is true because only one resource in the system has a non-convex utility (the cache). In a more general case where multiple resources are not convex, more sophisticated algorithms such as Qureshi and Patt's Lookahead [79] would probably be needed.

2.4.3 Design Issues

In this section, we discuss three practical issues that must be addressed: bankruptcy, market convergence, and wealth redistribution.

Bankruptcy

Sometimes, the price for power can be so high in the market that an agent cannot even afford to buy the minimum power to operate at 800 MHz. In such an event, the agent will file for "bankruptcy." The agent will be excluded from the bidding process, and it will be allocated the bare minimum: one way of the cache, and allowed to operate at 800 MHz for the next interval. The other agents will bid for the remaining resources. The bankrupt agent will re-join the bidding process at the next interval.

Convergence

Many practical studies show that a market similar to ours is likely to operate quite well [40, 90]. Still, we do experimentally observe some circumstances where prices continue to oscillate by more than 1% (our convergence criterion). We could consider a market with a more relaxed convergence threshold, e.g. 5% fluctuation. This may eliminate some of the non-convergent cases, and also reduce the number of bidding iterations. However, it may lead to a less optimal allocation. Other convergence criteria exist in the literature; for example, utility fluctuation [40]. However, we observed experimentally that there was no practical difference between this and our original criterion in terms of system throughput or convergence rate.

One reason for continued oscillation is that cache allocation is done at the granularity of cache ways, and thus the utility function is not continuous. Agents may be swinging between two neighboring cache ways across iterations, resulting in a non-trivial fluctuation in cache price.

There comment on two potential solutions to deal with this situation. First, we can introduce a *price-smoothing* mechanism, by incorporating the price in the last iteration (p_j^{last}):

$$p_j = \alpha \times p_j^{\text{last}} + (1 - \alpha) \times \frac{\sum_i b_{ij}}{R_j} \tag{2.10}$$

In this way, the history of the price is factored in, which helps agents to better understand the contention of the resource in the market. We empirically pick α to be 0.2, and we observe that it can greatly improve the market convergence rate.

Another option is to adopt a *price-anticipating* mechanism [40] instead of our *price-taking* approach. In this mechanism, although an agent does not know how others will change their bids, it realizes that the increase/decrease of its bids will change the resource prices, according to Equation 2.1. Therefore, during the local search for optimal bids, the agent will no longer consider the price to be a fixed number: it will factor in the impact of its changing bid on the resource price when it tries to trade one cache way for power.

In our experimental setup, both solutions show similar system throughput and convergence rate. Because price-anticipating agents increase the complexity when bidding, in the rest of the paper we adopt the *price-smoothing* technique.

In any case, if the market ultimately cannot converge after a while, we have to announce that our market fails to converge. In XChange, we set the cut-off threshold to be 30 iterations; if the prices still fluctuate by more than 1% at that point, we terminate the bidding process. In that case, resources are allocated as follows: first, each agent estimates its utility under the current resource prices and its bids; then, each agent estimates its utility under an equal-share allocation. If one of the agent prefers equal share, we enforce the system to fall back to equal-share allocation for all agents. Otherwise, the resources are allocated according to the agents' last bids. In this way, such a "fail-safe" mechanism virtually guarantees that the allocation decision is at least as good as equal-share. In fact, our experiments show that in most cases, agents prefer the market outcome rather than equal-share. This is especially true if multiple equilibria exist, and the market is simply oscillating among them.

Wealth Redistribution

In our initial design, our market framework treats all agents equally, by assigning to them the same initial budget. However, in our experiments, we find that an equal budget constraint might not be efficient enough from both a system and a user perspective. For example, *libquantum* is easily satisfied with very few resources. Because it's a highly memory-intensive application, its core stalls most of the time. As a result, it can operate at 4 GHz while consuming very little power (much lower than an equal-share power allocation). In addition, its working set can never fit into the shared cache, and allocating more than one cache way to it does not bring any significant benefit. But because it has the same budget as the other agents, it is likely to disrupt the market by preventing the other agents from obtaining resources that would contribute to a higher system throughput.

We propose a simple heuristic, namely to assign to each agent a budget proportional to its "potential" in performance gains:

$$B \propto (1 - \frac{t_{min}}{t_{max}}) \tag{2.11}$$

where t_{min} is the estimated execution time when the application is running alone (and thus enjoys all the chip's power and the maximum number of cache ways that UMON is able to monitor—see Section 2.5), and t_{max} is the estimated execution time when the application is running with minimum resources (one cache way and the lowest possible frequency). These quantities are not measured, but rather computed using Equation 2.5 and Equation 2.7 at the beginning of each partition interval; therefore, they do not lead to additional overhead.

This wealth redistribution technique biases budgeting toward the applications that have higher potential. As we discuss later in Section 2.7, this result in higher overall throughput, at the expense of some fairness. In those circumstances where fairness among cores are highly preferred, this wealth redistribution mechanism can be easily turned off.

2.5 Implementation

We propose to implement XChange as a combination of hardware and software. The hardware is responsible for performance monitoring and modeling, and the software is responsible for conducting the market's bidding and pricing mechanism.

Component	Quantity	Width	Bits
UMON shadow tag	16×64	28	28,672
UMON hit counter	16	32	512
DL1 CP_{global} counter	1	32	32
Per request CP_{global} counter	16	32	512
Total			29,728

Table 2.1: Per-core hardware overhead of online performance modeling.

2.5.1 Hardware

Table 2.1 details the per-core hardware overhead of our online performance modeling mechanism. As is discussed in Section 2.4, XChange relies on UMON shadow tags to predict cache behavior of applications. We employ a dynamic set sampling (DSS) technique [79] and sample 64 out of 2k sets. We further limit the stack distance to 16, because we empirically observe that no application can afford more than 4× its equal-share allocation. In addition, for the workloads we study, we observe that their marginal utility for cache is mostly zero beyond 16 ways. However, in more general cases, a deeper stack distance may be needed to more accurately characterize the workloads' cache behavior, at the cost of higher storage overhead.

On top of that, in order to track the length of critical memory path CP_{global}, the L1 data cache of each core requires a global critical path counter CP_{global}, and each memory request needs a counter to save a copy of CP_{global} (as a field of DL1's MSHR). Further, it needs each processor to keep track of how much dynamic energy it consumes in the past interval. Because modern processors already have this feature built in [82], we exclude this from our hardware overhead.

In all, the per-core hardware overhead of XChange amounts to about 3,700 bytes.

2.5.2 Software

We propose to implement XChange's bidding-pricing mechanism as a part of an OS kernel module. In some Linux-based SMP systems, all cores are simultaneously interrupted by an APIC timer every 1 ms to conduct a kernel statistics update routine. We propose to piggyback on this interrupt to incorporate our market mechanism. We assume a shared-memory model, and designate a master core to be responsible for collecting the bids (reading shared variables) and computing the price. The whole procedure works as follows:

1. Every 1 ms, after each core has finished its kernel update routine, the master core posts an initial price.
2. All the cores start to search for their optimal bids using the local hill-climbing technique explained in Section 2.4.2.
3. After a global barrier to ensure that all bids are computed, the master core collects the bids, and computes the price. If the prices do not change (within 1%) compared to the previous iteration, the market converges, and the resources are allocated using Equation 2.2. Otherwise, repeat Step 2.

Because we are using a shared-memory model for the market mechanism, no special hardware is needed for inter-core communication. The execution time overhead of this procedure is discussed in Section 2.9.

Priorities and Real-Time Issues

In a real system there may be applications with different priorities. High-priority applications probably expect to enjoy more CPU cycles and access to more of the on-chip resources. Our market framework can handle this by assigning higher budgets to these applications, and therefore increase their purchasing power. How exactly to calculate the appropriate budgets is left for future work.

Another issue that high-priority or real-time applications may face could be caused by the fact that our market framework involves all the cores for the bidding-pricing procedure. Although the overhead is small (Section 2.9), these types of applications may be affected undesirably. One way to handle such cases may be to delegate the bidding on behalf of such time-sensitive applications on low-priority cores. Other solutions may be possible.

Yet another issue for real-time applications may be that the resulting resource allocation may be insufficient to meet hard deadlines. We propose address this by providing those applications with enough resources, and then excluding them from the market. (Their resource demands could be provided externally, or they could be estimated using the utility model we described in Section 2.4.1.)

2.6 Experimental Methodology

2.6.1 Architectural Model

We evaluate XChange using a heavily modified version of SESC [81]. The CMP configurations with 8 (small-scale) and 64 (large-scale) cores, and the out-of-order core parameters are shown in Table 2.2. We also faithfully model Micron's DDR3-1600 DRAM timing [49], shown in Table 2.3.

We use Wattch [16] and Cacti [87] to model the dynamic power consumption of the processors and memory system. The static power consumption is approximated as a fraction of the dynamic power, and this fraction ratio is exponentially dependent on the system temperature [21]. Intel has adopted a similar approach for its Sandy Bridge power management [82]. We rely on Hotspot [89] integrated with SESC to estimate the run-time temperature of our CMP system.

Our baseline CMP system is able to regulate three shared on-chip resources: L2 cache, off-chip memory bandwidth, and power budget. We distribute the power budget across the chip via per-core DVFS. When a processor exceeds its allocated power, it is forced to slow down until its power consumption drops within its share. We guarantee that each core receives at least one cache way; the remaining cache is distributed based on the resource allocation decision. Finally, we implement the Fair-Queue (FQ) memory scheduler proposed by Nesbit *et al.* [74] to regulate off-chip memory bandwidth. The service share rate will be designated at the memory controller by the resource allocator.

Table 2.2: System configuration.

Chip-Multiprocessor System Configuration	
Number of Cores	8 / 64
Power Budget	80W / 640W [a]
Shared L2 Cache Capacity	4MB / 32MB
Shared L2 Cache Associativity	32 / 256 ways [b]
Memory Controller	2 / 16 channels
Core Configuration	
Frequency	0.8 GHz - 4.0 GHz
Voltage	0.8V - 1.2 V
Fetch/Issue/Commit Width	4 / 4 / 4
Int/FP/Ld/St/Br Units	2 / 2 / 2 / 2 / 2
Int/FP Multipliers	1 / 1
Int/FP Issue Queue Size	32 / 32 entries
ROB (Reorder Buffer) Entries	128
Int/FP Registers	160 / 160
Ld/St Queue Entries	32 / 32
Max. Unresolved Branches	24
Branch Misprediction Penalty	9 cycles min.
Branch Predictor	Alpha 21264 (tournament)
RAS Entries	32
BTB Size	512 entries, direct-mapped
iL1/dL1 Size	32 kB
iL1/dL1 Block Size	32 B / 32 B
iL1/dL1 Round-Trip Latency	2 / 3 cycles (uncontended)
iL1/dL1 Ports	1 / 2
iL1/dL1 MSHR Entries	16 / 16
iL1/dL1 Associativity	direct-mapped / 4-way
Memory Disambiguation	Perfect

[a] We anticipate that the CMP systems with different number of cores will not be fabricated under the same technology. For simplicity, in our evaluation, we use a chip TDP of 10W per core. IBM's Power8 reportedly consumes twice as much per core (it has 12 cores).

[b] In the evaluation, we partition the shared last-level cache at the granularity of cache ways [79]. In an actual implementation, any of the fine-grained cache partition mechanisms proposed in the literature could be used (e.g., PriSM [67], Vantage [85]).

Workload Construction

We use a mix of 25 applications from SPEC2000 [91] and SPEC2006 [92] to evaluate our proposal. Each application is cross-compiled to a MIPS executable, using gcc 4.6.1 at -O2 optimization level. The bundles of applications are executed until every application has committed 200 million instructions (8 cores), or 50 million instructions (64 cores). When an application finishes, we stop measuring its performance, but continue executing and engaging it in global resource allo-

Table 2.3: DRAM parameters.

Micron DDR3-1600 DRAM	[49]
Transaction Queue	64 entries
Peak Data Rate	12.8 GB/s
DRAM Bus Frequency	800 MHz (DDR)
Number of Channels	2 / 16
DIMM Configuration	Dual rank
Number of Banks	8 per rank
Row Buffer Size	1 KB
Address Mapping	Page Interleaving
Row Policy	Open Page
Burst Length	8
tRCD	10 DRAM cycles
tCL	10 DRAM cycles
tWL	7 DRAM cycles
tCCD	4 DRAM cycles
tWTR	6 DRAM cycles
tWR	12 DRAM cycles
tRTP	6 DRAM cycles
tRP	10 DRAM cycles
tRRD	6 DRAM cycles
tRTRS	2 DRAM cycles
tRAS	28 DRAM cycles
tRC	38 DRAM cycles
Refresh Cycle	8,192 refresh commands every 64 ms
tRFC	128 DRAM cycles

cation, to ensure that it continues to exert pressure on the resources shared with other applications in the bundle. For each application, we use Simpoints [17] to pick the most representative program slice.

The multiprogrammed workloads we use are shown in Table 2.4. We classify the 25 applications into *Power-sensitive*, *Cache-sensitive*, and *Memory-sensitive* using profiling, and then create bundles that constitute a varied mix of applications in each category. When the number of cores exceeds the number of apps in a bundle, the bundle is replicated across the chip. For example, 8 copies of VCUGXNTZ will run to occupy all the cores in a 64-core CMP.

Our evaluation is based on multiprogrammed workloads because we anticipate to allocate resources at the granularity of applications. For multithreaded workloads, we can either treat each thread as an individual agent in the mar-

Table 2.4: Multiprogrammed workloads, combining cache-, processor- and memory-sensitive applications.

ROAFLDGV	vp**r** - tw**o**lf - **a**psi - mc**f**	C^4
	mi**l**c - GemsF**D**TD - **g**romacs - **v**ortex	M^2P^2
XNIBDWFA	sople**x** - libqua**n**tum - lesl**i**e3d - **b**waves	M^4
	GemsF**D**TD - s**w**im - mc**f** - **a**psi	M^2C^2
MHKULBFP	ga**m**ess - **h**mmer - sixtrac**k** - w**u**pwise	P^4
	mi**l**c - **b**waves - mc**f** - amm**p**	M^4
FOARLNBW	mc**f** - tw**o**lf - **a**psi - vp**r**	C^4
	mi**l**c - libqua**n**tum - **b**waves - s**w**im	M^4
AZFPIBDW	**a**psi - b**z**ip2 - mc**f** -amm**p**	C^4
	lesl**i**e3d - **b**waves - GemsF**D**TD - s**w**im	M^4
XIDWCHAF	sople**x** - lesl**i**e3d - GemsF**D**TD - s**w**im	M^4
	calculix - **h**mmer - **a**psi - mc**f**	P^2C^2
NXLIGEOR	libqua**n**tum - sople**x** - mi**l**c lesl**i**e3d	M^4
	gromacs - h264r**e**f - tw**o**lf - vp**r**	P^2C^2
LNIBDWOT	mi**l**c - libqua**n**tum - lesl**i**e3d - **b**waves	M^4
	GemsF**D**TD - s**w**im - tw**o**lf - ar**t**	M^2C^2
ROPAFZXN	vp**r** - tw**o**lf - amm**p** - **a**psi	C^4
	mc**f** - b**z**ip2 - sople**x** - libqua**n**tum	C^2M^2
LXNIBDRO	mi**l**c - sople**x** - libqua**n**tum - lesl**i**e3d	M^4
	bwaves - GemsF**D**T**D** - vp**r** - tw**o**lf	M^2C^2
VCUGXNTZ	**v**ortex - **c**alculix - w**u**pwise - gromacs	P^4
	sople**x** - libqua**n**tum - ar**t** - b**z**ip2	M^2C^2

ket, or combine all the threads of one app as one agent to bid and share the resources.[4]

Performance Metrics

A key issue in resource allocation is the figure of merit. Eyerman and Eeckhout propose that two metrics be reported for a CMP system running multiprogrammed workloads: One to represent a system perspective, which cares about system throughput; and one to represent a user perspective, which cares about the average turnaround time of an individual job. The proposed metrics are weighted speedup and average slowdown of co-running applications (the reciprocal of harmonic speedup), respectively:

[4] Skewing resources among threads in a multithreaded application (e.g., to alleviate synchronization imbalance) is beyond our scope, can be incorporated orthogonally to our approach, and has been studied elsewhere [11,35].

$$\text{Weighted Speedup} = \frac{1}{N}\sum_{i=1}^{N}\frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}} \tag{2.12}$$

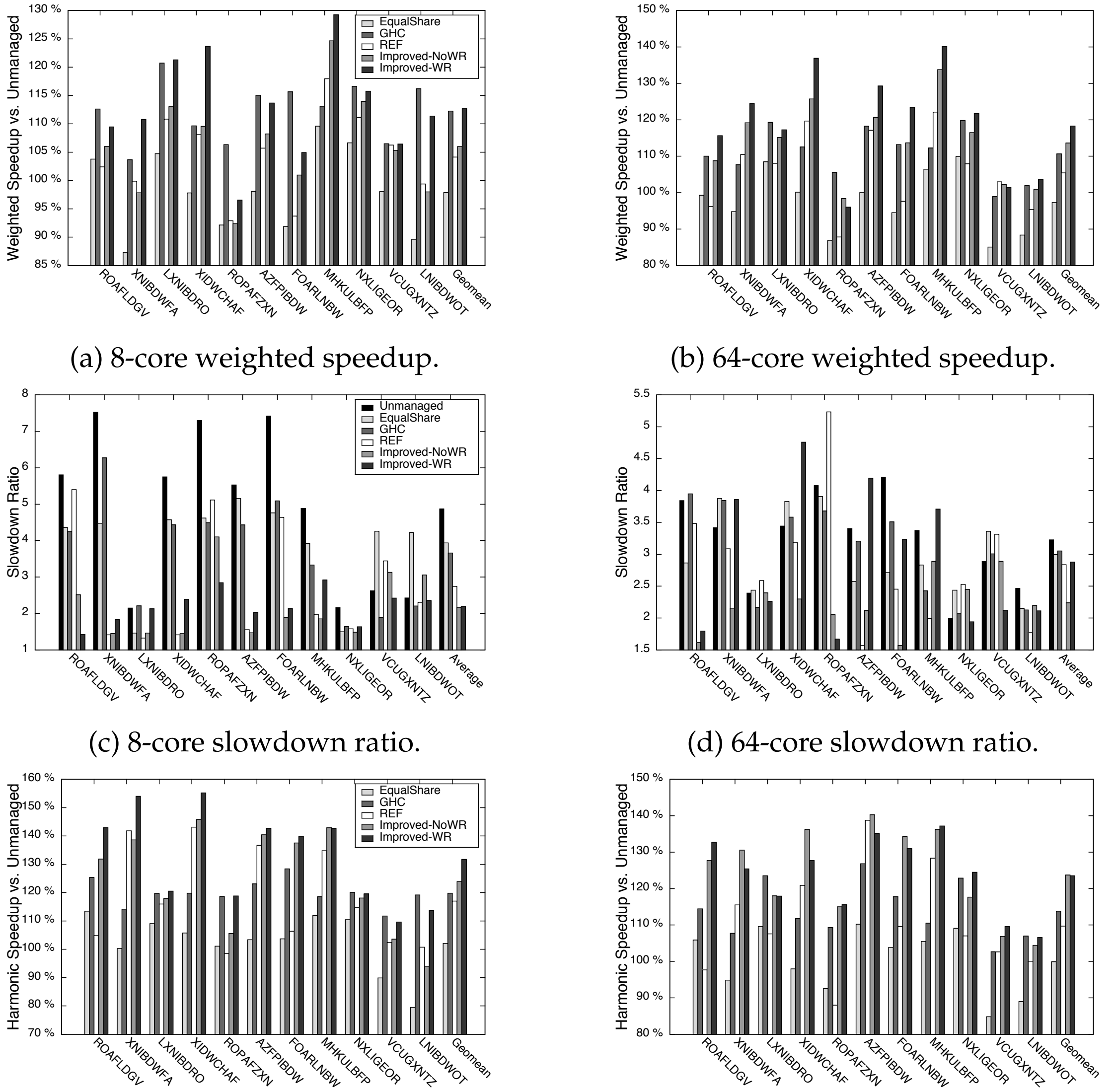
$$\text{Harmonic Speedup} = \frac{N}{\sum_i \frac{IPC_i^{\text{alone}}}{IPC_i^{\text{shared}}}} \tag{2.13}$$

A system could achieve high throughput (i.e., weighted speedup) by starving one or two applications while benefiting all the others; however, system fairness (i.e., harmonic speedup) would suffer as a result, providing a bad experience to some users. A side-by-side comparison of weighted and harmonic speedups would expose this behavior. To isolate the fairness component, we also report separately the ratio between maximum and minimum slowdowns across the bundle [34]:

$$\text{Slowdown Ratio} = \frac{\max_i \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}}}{\min_i \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{alone}}}} \tag{2.14}$$

2.7 Evaluation

Figure 2.2 reports system throughput (weighted speedup), slowdown ratio, and fairness (harmonic speedup) for an equal-share allocation (EqualShare), two competing mechanisms (GHC [23] and REF [111]), as well as XChange with (-WR) and without (-NoWR) wealth redistribution. System throughput and harmonic speedup results are normalized to an unmanaged allocation (Unmanaged). Unmanaged adopts LRU as the policy to manage shared cache, and full-chip DVFS rather than per-core DVFS, to guarantee that the chip's power con-



(a) 8-core weighted speedup.

(b) 64-core weighted speedup.

(c) 8-core slowdown ratio.

(d) 64-core slowdown ratio.

(e) 8-core harmonic speedup.

(f) 64-core harmonic speedup.

Figure 2.2: Comparison of system throughput (weighted speedup; higher is better), slowdown ratio (lower is better), and fairness (harmonic speedup; higher is better) among EqualShare, GHC, REF, XChange-NoWR, and XChange-WR, under different CMP configurations. System throughput and harmonic speedup results are normalized to Unmanaged.

sumption does not exceed its TDP.

The results are obtained by modeling the allocation algorithms faithfully, however the timing overhead of running these algorithms is set to zero in all cases. In the next section, we show that XChange's actual overhead is absolutely

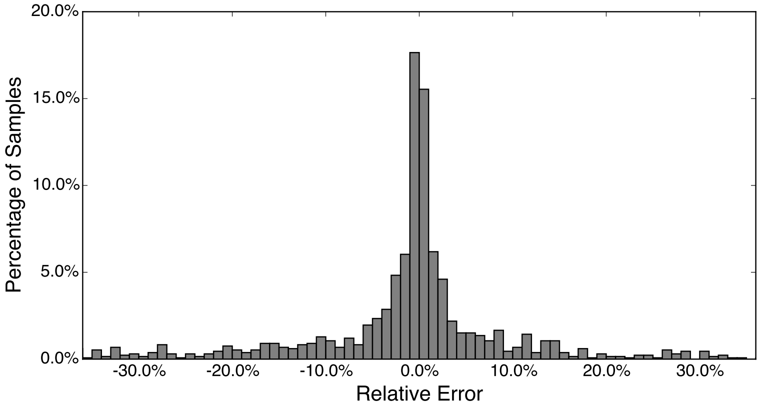


Figure 2.3: Accuracy of XChange in predicting the length of memory phase.

and relatively very low, and that it scales much better than GHC as the number cores/apps increases. Furthermore, the results for *REF* assume prior app profile knowledge [111]. Thus, any comparison with the competing mechanisms here tends to favor those and go against XChange.

2.7.1 XChange vs. Unmanaged

We first compare XChange against the Unmanaged baseline. Figure 2.2 shows that, on average, both XChange-NoWR and -WR improve system throughput significantly—by 13.62% (6.01%) and 18.30% (12.67%), respectively, for the 64 (8) CMP configuration. Looking at each individual bundle, we find that, although Unmanaged is almost universally inferior to XChange in terms of weighted speedup, it modestly outperforms XChange for ROPAFZXN in both the 8- and the 64-core configurations. We now look at this case a bit more closely.

Bundle ROPAFZXN has six out of eight cache-sensitive applications. In Unmanaged, two of the cache-sensitive applications manage to hoard most of the cache space, letting the other four starve. The market-based approaches with built-in fairness (XChange but also REF and EqualShare) naturally do not exhibit this behavior.

On the other hand, the XChange configurations yield clearly superior slowdown ratio and harmonic speedup for that bundle over Unmanaged, in fact at a level that the other fairness-aware configurations fall well short of. On average across all bundles, and for the 8-core configuration, XChange-NoWR and -WR outperform Unmanaged in harmonic speedup (23.87% and 31.74%, respectively), and also slowdown ratio (2.17 and 2.19, respectively, vs. 4.87). The results are similar for larger 64-core configuration.

Overall, XChange outperforms Unmanaged almost universally in all three metrics.

2.7.2 XChange vs. EqualShare

We now compare XChange against the equal-share allocation (EqualShare). Figure 2.2 shows that, for the 64-core configuration, both XChange-NoWR and XChange-WR improve system throughput by 16.33% and 21.01% on average, respectively. In fact, XChange is superior in all the experiments. Results for the smaller 8-core configuration are similarly significant.

Looking a bit more closely at a representative bundle ROAFLDGV, we find that in XChange-NoWR, but more so in XChange-WR, *mcf* is able to stay atop the "cache utility step" and obtain 12 cache ways. In XChange-NoWR, *mcf* spends most of its budget to accomplish this, and as a result power is some-what sacrificed, for an overall dampened performance gain. On the other hand, XChange-WR observes that *mcf*'s potential is high compared to the other apps, and consequently it assigns it a higher budget. Then, *mcf* can afford to "purchase" more power to run faster (Section 2.4.3), which results in higher speedups.

Notice that XChange-NoWR and -WR also yield superior fairness to EqualShare (2.24 and 2.88, respectively, vs. 3.0 average slowdown ratio). For example, *libquantum* is both cache- and power-insensitive, and thus its slowdown under any resource allocation is negligible. On the other hand, applications that are power-hungry (e.g., *calculix*) or cache-hungry (e.g., *art*, *mcf*) will experience a significant slowdown in EqualShare. The allocations by the XChange configurations are more balanced in that regard.

Overall, the combination of higher throughput and improved fairness makes XChange-NoWR and -WR significantly outperform EqualShare by 23.78% and 23.60%, respectively, in average harmonic speedup.

2.7.3 XChange vs. GHC, REF

Configuration *GHC* corresponds to Chen and John [23] with their online performance modeling, and *REF* corresponds to Zahedi and Lee with prior app profile knowledge, as evaluated in that work [111]. When compared against GHC and REF, XChange-WR is superior in every metric, and the all-out winner. This holds for 64- as well as 8-core experiments. The harmonic speedup summarizes this well, with XChange-WR's 23.53% (31.74%) average easily outdoing GHC's 13.80% (19.80%) and REF's 9.70% (17.03%) in the 64 (8) setup.

When looking at REF more closely, we find that in many cases it is usually too biased toward maintaining game-theoretic fairness guarantees, and realized throughput gains suffer as a result. Although these may generally translate into a better user experience, our mechanism succeeds at making more aggressive decisions to maximize system throughput, while still striving for spreading fairly the impact across all applications' execution times.

2.7.4 Overall Effect of Wealth Redistribution

As discussed in Section 2.4.3, we introduce a wealth redistribution technique to further improve system throughput, possibly at some expense of fairness. Our simulations prove this intuition: XChange-NoWR achieves the best slowdown ratio (2.23 and 2.17 for 64- and 8-core, respectively) over all other techniques. In the meantime, XChange-WR achieves the best weighted speedup, and is consistently 5% better than XChange-NoWR. On the whole, XChange-WR slightly outperforms -NoWR in harmonic speedup.

Section 2.7.2 describes an example where *mcf* benefits by the budget redistribution. Let us briefly discuss another example, XIDWCHAF, in the 8-core configuration. In XChange-WR, the budget of cache- and power-sensitive apps such as *apsi* and *calculix* are offered higher budget than memory-bound apps, because their potential in deriving speedups from resources is higher. As a result, XChange-WR's weighted speedup over Unmanaged is about three times higher than XChange-NoWR's (23.68% vs. 9.56%, respectively), but on the other hand its slowdown ratio increases from 1.45 (XChange-NoWR) to 2.39.

2.8 First-order Model Validation

Although we have shown significant improvements for both throughput and fairness, a measure of XChange's model accuracy is a useful insight. A key aspect of XChange's utility model is the estimation of the memory phase, which relies on the simplifying assumption that MLP remains unchanged across different cache allocations for any one application. To validate this memory phase estimation, we run each application alone with all possible cache allocations. Each run will give us the real length of memory phase under that specific cache capacity, and the estimates for length of all the others.

Figure 2.3 shows the accuracy of the estimation. The average error is 7.63%, indicating that our estimation is reasonably accurate. In general, we find that accuracy decreases when predicting for cache allocations that are more distant from the current allocation (e.g., predicting the length of memory phase under one cache way when the core currently owns eight cache ways).

We also find that another source of error is UMON: With very limited L2 allocated cache size, L1 cache lines will be more often invalidated due to replacements in the L2 cache. As a result, for some applications, the L1 miss rate will increase as L2's allocation decreases, which is not captured well by UMON.

2.9 Scalability

Our simulations so far have excluded from all the configurations studied the overhead of searching through the resource allocation space. In this section, we study the scalability of XChange against GHC when that overhead is factored in. The hardware setup is as follows: an N-core CMP consumes $10N$ W of power and holds a $4N$-way, $0.5N$ MB L2 cache. Memory bandwidth is set to equal-share with FQ scheduling [74]. We limit the amount of cache that a single core can appropriate to 16 ways (2 MB) due to the UMON hardware overhead discussed in Section 2.5. The hardware configuration is the same as the simulation described in Section 2.6, and the synchronization/communication overhead across cores is included in all cases.

Recall from Section 2.5.2 that we anticipate the resource allocation mechanisms to be implemented in the kernel. We actually implement GHC and XChange as programs that we run on our simulation platform, and use the number of cycles each algorithm takes to converge as the metric to measure scalability.

As shown in Table 2.5, the total cycle count of GHC grows essentially quadratically. Recall that GHC is inherently sequential: A single core is responsible for the entire search. During the hill-climbing period, that core has to stop its normal execution to make the resource allocation decision on behalf of the entire CMP. With 64 cores on the chip, it would take that core 34% of a 5-million-cycle interval to come up an allocation decision. During that time, all the other cores would be running in a obsolete, probably suboptimal operating point. Note that, even in cases in which the performance of the old and new

# cores	8	32	64	128	256
GHC					
Cycles (K)	43	484	1,697	6,418	24,903
% interval	0.87%	9.69%	33.95%	128%	498%
XChange-WR					
Cycles (K)	9.47	12.49	15.89	22.64	52.70
% interval	0.19%	0.25%	0.32%	0.45%	1.05%

Table 2.5: Search overhead for GHC and XChange-WR. Interval is 5 million cycles.

allocations for the interval were similar, the overall performance would be no better than the one reported earlier in the evaluation. For a CMP with more than 64 cores, it is simply unfeasible to apply GHC for the interval chosen.

In contrast, the XChange market-based mechanism comes to an allocation decision much more quickly. This is mainly for two reasons: (1) because most of the work is done concurrently across all cores; and (2) because the local allocation space each core needs to search is relatively small.

Table 2.5 shows the average cycle count for XChange-WR to converge, and the percentage of the partitioning interval every core will diverge from normal execution to compute allocation decision. For CMP systems with fewer than 128 cores, the system downtime of all market-based models is less than 0.5%. Above 128 cores, the cycle count begins to increase more or less linearly with the number of cores. This is because the master core needs to collect and sum up all the bids from the agents in the system to compute the resource price, and this centralized step starts to dominate the overall cycle count.

A potential way to alleviate this is to parallelize the price computation, which is basically a reduction operation over the bids from the agents, into a tree fashion. Another option is to make the partition interval longer (also for GHC), but this may make the market too insensitive to application phase changes. We leave these and other possible options as future work.

2.10 Summary

We have proposed XChange, a market-based mechanism to dynamically allocate multiple resources in CMPs. By formulating the CMP as a market, where each core pursues its own benefit, the system is able to maintain a good balance between system throughput and fairness. Our evaluation shows that, compared against an equal-share allocation, our market-based technique improves system throughput (weighted speedup) on average by 21%, and fairness (harmonic speedup) on average by 24% in a 64-core CMP system. Compared with a state-of-the-art centralized allocation scheme [23], that is at least about twice as much improvement over the equal-share allocation.

We have also shown that our market-based mechanism is largely distributed, where agents concurrently strive to maximize their individual utility. As a result, our approach converges significantly faster than the state-of-the-art centralized optimization technique we compare against.

CHAPTER 3

REBUDGET: TRADING OFF EFFICIENCY VS. FAIRNESS IN MARKET-BASED MULTICORE RESOURCE ALLOCATION VIA RUNTIME BUDGET REASSIGNMENT

3.1 Introduction

Devising scalable chip-multiprocessor (CMP) designs is an important goal for the upcoming manycore generation. A key challenge to scalability is the fact that these cores will share hardware resources, be it on-chip cache, pin bandwidth, the chip's power budget, etc. Prior work has shown that freely contending for shared resources can penalize system performance [12, 23, 26]. Thus, allocating resources efficiently among cores is key.

Unfortunately, single-resource, and more generally uncoordinated resource allocation, can be significantly suboptimal, due to its inability to model the interactions among resources [12]. A few solutions have been proposed to coordinate resource allocation across multiple resources, and their performance estimation methods range from trial runs [6, 26, 59], to artificial neural networks [12], and to analytical models [23–25]. Unfortunately, these all rely on centralized mechanisms (e.g., global hill-climbing) to optimize system throughput, essentially exploring the global search space sequentially, which may be prohibitively expensive, particularly in large-scale systems.

More recently, a number of market-based approaches have been introduced. Chase et al. propose a static market [22], where the players reveal to the resource supplier the amount of money each is willing to pay as a function of allocated service units, and the central market then allocates the available computing resources so that monetary profit is maximized. Still, because the maximization process is done by the supplier centrally, it is unclear whether it could deal with a large-scale system efficiently.

In the context of distributed computing clusters, Lai et al. propose a market-based solution to resource allocation that allows players to adjust their bids dynamically in response to the others bids to that resource [60]. Resource allocation is done in a largely distributed manner, which enables the system to scale better than centralized approaches. Recently, in our XChange work [103], we also propose one such dynamic market in the context of CMPs, and similarly show that XChange is scalable due to its largely distributed nature: Instead of making the resource allocation decision globally, each core in the CMP is actively optimizing its resource assignment largely independently of each other, and participants demands are reconciled through a relatively simple pricing strategy.

XChange also shows that it can achieve a good balance between system efficiency and fairness. The study is purely empirical, however, and thus it does not provide any guarantees on the loss of efficiency and fairness. It is well-known, for example, that market mechanisms in equilibrium can sometimes be highly inefficient—this is known as *Tragedy of Commons* [44]. Therefore, a number of research efforts have focused on quantifying the efficiency loss compared to the optimal resource allocation, known as the *Price of Anarchy (PoA)*. For example, Zhang studies a market where all players have the same amount of money (budget) to purchase resources, and he finds that the overall system efficiency in such a market can be low ($1/\sqrt{N}$ of the maximum feasible utility, where N is the number of market players), but fairness is high (0.828-approximate envy-free, a measure of fairness) [112]. This is consistent with our empirical observations in the XChange work [103].

Recently, Zahedi and Lee propose a resource allocation mechanism named *elasticities proportional* (EP) for CMPs [111], which does provide game-theoretic guarantees such as Pareto efficiency, envy-freeness, etc. However, such guarantees rely on the assumption that an application's utility can be accurately curve-fitted to a Cobb-Douglas function, where the coefficients are used as the "elasticities" of resources. Our XChange work shows that EP can in fact perform worse than expected when such curve-fitting is not well suited to the applications. In addition, although EP is proven to be Pareto-efficient, its efficiency loss compared to global optimality is not quantified.

To improve system efficiency while sacrificing some fairness, our XChange work discusses a *wealth redistribution* technique, which varies the players' budget based on an estimation of their potential for performance gain. However, XChange's wealth redistribution is an on/off technique, providing no "knob" to control the efficiency vs. fairness trade-off. It is also not backed up by a theoretical result that would provide bounds for this trade-off. To the best of our knowledge, there is no theoretic study that is able to quantify the loss of *both* efficiency and fairness under an arbitrary budget assignment.

Contributions

The contributions of this chapter are as follows:

— We introduce a new *Market Utility Range (MUR)* metric, which helps us establish a theoretical bound for efficiency loss of a market equilibrium under a constrained budget. Specifically, we show that, if MUR $\geq$ 0.5, then PoA $\geq (1 - \frac{1}{4\text{MUR}}) \geq 0.5$ (i.e., the efficiency is guaranteed to be at least 50% of the optimal allocation); and that if MUR < 0.5, then PoA ≥ MUR.

— We introduce a new *Market Budget Range (MBR)* metric, which helps us evaluate the fairness of a market equilibrium under a constrained budget. We show that any market equilibrium is $(2\sqrt{1 + \text{MBR}} - 2)$-approximate envy-free.

— We propose *ReBudget*, a budget re-assignment technique that is able to systematically control efficiency and fairness in an adjustable manner. We evaluate ReBudget on top of XChange, using a detailed simulation of a multicore architecture running a variety of applications. Our results show that ReBudget is efficient and effective. In particular, it can achieve 95% of the maximum feasible efficiency. Furthermore, when combined with the analysis using MUR and MBR metrics, it can provide worst-case fairness guarantees.

3.2 Market Framework

This chapter adopts the general market-based resource allocation framework proposed in our XChange work [103]. In this section, we describe our efficiency and fairness models in the context of that framework.

Assume a market consisting of N players and M resources. Each player i has a utility function $U_i(\mathbf{r_i})$ when it is allocated $\mathbf{r_i} = (r_{i1}, r_{i2}, \ldots, r_{iM})$ resources. We assume that a player's utility functions is concave, nondecreasing, and continuous. Note that this may not always hold in the CMP context (e.g., the utility of the allocated cache space [8,103]). We describe how we address this issue later in Section 3.4.

Every player i is allowed to bid b_{ij} for resource j, and the sum of its bids cannot exceed its budget B_i (i.e., the total amount of money it is allowed to spend): $\sum_j b_{ij} \leq B_i$. The market reconciles those bids by adopting a *proportional* allocation scheme, which is widely used [60,112] and considered fair. The market first collects bids from all the players, and then determines the price of each resource j as follows:

$$p_j = \frac{\sum_{i=1}^{N} b_{ij}}{C_j} \tag{3.1}$$

where C_j represents the total amount of resource j. As a result, player i gets r_{ij} units of resource j proportionally to its bid: $r_{ij} = \frac{b_{ij}}{p_j}$.

The essence of this type of market-based approach is that it is a largely distributed mechanism: the players independently traverse their local search space to find the bids that maximize their own utilities, bringing the market toward a resource allocation that is Pareto-optimal [75]. In order to find such optimal bids, each player needs to solve an optimization problem, which can be modeled as follows: Given the price p_j of resource j announced by the market, player i is able to compute the sum of other players' bids to that resource: $y_{ij} = \sum_{i' \neq i} b_{i'j} = p_j \times C_j - b_{ij}$. By making the simplification that other players do not change their bids and therefore y_{ij} are constants, player i is able to make a prediction on the amount of resource r_{ij} it can get if it changes its bids from b_{ij} to b'_{ij}:

$$r_{ij} = \frac{b'_{ij}}{b'_{ij} + y_{ij}} C_j \tag{3.2}$$

Combining this equation with player i's utility function $U_i(\mathbf{r_i})$, the player can obtain its utility function vs. its bids: $U_i(\mathbf{b_i}) = U_i(\frac{b_{ij}}{b_{ij}+y_{ij}} C_j)$. Then the optimization problem a player needs to solve is:

$$\begin{aligned} &\text{maximize} \quad U_i(\mathbf{b}_i) \\ &\text{subject to} \quad \sum_j b_{ij} \leq B_i \end{aligned} \tag{3.3}$$

Using the Lagrangian multiplier method, we conclude that if such optimal bids exist, there exists a player-specific constant $\lambda_i > 0$ such that, for any resource j:

$$\frac{\partial U_i}{\partial b_{ij}} \begin{cases} = \lambda_i & \text{if} \quad b_{ij} > 0 \\ < \lambda_i & \text{if} \quad b_{ij} = 0 \end{cases} \tag{3.4}$$

Intuitively, we define λ_{ij} to be the rate of utility change (marginal utility) if player i changes its bid on resource j by one unit: $\lambda_{ij} = \frac{\partial U_i}{\partial b_{ij}}$. From Equation 3.4, if player i submits non-zero bids on different resources j, the λ_{ij} for all those resources are the same, and equal to λ_i in Equation 3.4. For resources with zero bids, their λ_{ij} is necessarily smaller than λ_i. Otherwise, it contradicts the condition that the bids maximize the player's utility.

Let's use an illustrative example. Assume player i bids on two resources, with $\lambda_{i1} = 1$ and $\lambda_{i2} = 2$. If the player moves one unit of bid from resource 1 to 2, its utility can be increased by 1 unit (-1 from resource 1 and +2 from resource 2). As a result, the current bids are not optimal, and the player can keep improving its utility by adjusting its bids until λ_{ij} are equal, or the bids on one of the resources drop to zero, beyond which no further profitable movement is possible.

3.2.1 Market Equilibrium

Market equilibrium is a state where all the players have no incentive to change their bids to improve utilities, and the resource prices remain stable. It is a desirable state because it is proven to be Pareto-optimal (i.e., no other resource allocation can make any one individual better off without making at least one individual worse off [75]). In order to find a market equilibrium, we adopt an iterative bidding–pricing process, similar to the one used in our recent XChange work [103]. Such a process can be divided into two steps: (1) the market broadcasts the current resource prices to all the players, and (2) the players adjust their bids to maximize their own utilities. These two steps are repeated iteratively until the market converges—i.e., for any player, its utility changes negligibly between two iterations. (In our implementation we detect this globally by monitoring prices instead, and assume convergence when they fluctuate within 1%.)

Zhang [112] has shown that a market equilibrium always exists in a *strongly competitive market*, where for any resource j, there always exists at least two players placing non-zero bids.

Lemma 1. *An equilibrium always exists for a strongly competitive market. The market equilibrium may not be unique.*

3.2.2 Efficiency

Given the existence of a market equilibrium, its system efficiency, also known as social welfare, is an important metric.

Definition 1. *The efficiency of a system is defined as the sum of players' utilities: Efficiency* $= \sum_i U_i(\mathbf{r}_i)$.

Nissan et al. [75] show that the efficiency of a market equilibrium can be low. Papadimitriou introduces the concept of *Price of Anarchy* (PoA) [76], which is the lower bound of the efficiency of a market equilibrium compared with that of the optimal allocation. Mathematically, let $\mathbf{r}_i^*$ denote a feasible resource allocation which maximizes the system efficiency, Ω be the set of resource allocation in market equilibrium (recall that a market equilibrium may not be unique), and $\mathbf{r}^n \in \Omega$ be a market equilibrium outcome. Let us also define optimal efficiency OPT $= \sum_i U_i(\mathbf{r}_i^*)$, and efficiency in market equilibrium Nash$(\mathbf{r^n}) = \sum_i U_i(\mathbf{r}_i^n)$, the Price of Anarchy is then defined as:

Definition 2. $PoA = \min_{\mathbf{r^n} \in \Omega} \frac{Nash(\mathbf{r^n})}{OPT}$

Note that PoA is a lower bound, which means that the efficiency of any market equilibrium $\mathbf{r^n}$ is guaranteed to be greater than PoA × OPT.

Zhang [112] studies PoA in a market with a proportionally balanced budget, where a player is given a budget in proportion to its maximum utility, i.e., the utility when it owns all resources. Zhang then shows:

Lemma 2. *The equilibrium in a market with a proportionally balanced budget has a Price of Anarchy* $PoA = \Theta(\frac{1}{\sqrt{N}})$.

Recall that N is the number of players; thus, Lemma 2 tells us that PoA will worsen with the number of players, and thus it can be prohibitively low in a large market.

3.2.3 Fairness

Envy-freeness (EF) is widely used to evaluate fairness of a resource allocation in real life [14,99], and it's recently introduced by Zahedi and Lee in the context of resource allocation in CMPs [111]. It is a metric to evaluate how much a player desires others' resources compared to what it owns.

Definition 3. *Envy-freeness (EF) of an allocation* $\mathbf{r} = (\mathbf{r}_1, \dots \mathbf{r}_N)$ *is* $EF(\mathbf{r}) = \min_{i,j} \frac{U_i(\mathbf{r}_i)}{U_i(\mathbf{r}_j)}$.

By definition, a resource allocation is envy-free when $EF \geq 1$—i.e., players prefer their own resources to those of others (at worst, they like them equally). Although a market equilibrium is provably envy-free under some form of utility constraints [111], in general it is not. And although it might seem that a market equilibrium would be generally fair as long as every player is endowed with the same budget, Zhang shows that this is not guaranteed [112]. As a result, Zhang defines c-approximate envy-free ($c \leq 1$) as follows:

Definition 4. *A market is c-approximate envy-free, if the envy-freeness of any market equilibrium is larger than c, i.e., for any player i,* $U_i(\mathbf{r}_i) \geq c \cdot \max_j U_i(\mathbf{r}_j)$.

It is straightforward that the equilibrium is more *"fair"* if c is closer to 1 (players envy others less). Zhang then proves [112]:

Lemma 3. *If each player in the market has the same budget, market equilibrium is at least 0.828-approximate envy-free. The bound is tight in the worst case.*

Note that the bounds of efficiency and fairness proven by Zhang [112] may not apply at the same time. Recall that Lemma 2 requires a proportionally balanced budget assignment (i.e., a player's budget is proportional to its maximum achievable utility), while Lemma 3 assumes every player has an equal budget. However, note that in the multicore resource allocation problem that we study, the utility function is in fact a value normalized to the maximum utility (discussed in Section 3.4.1). As a result, the maximum utility is 1 for all the players, and therefore these two markets are equivalent within the scope of this chapter.

Thus, by combining Lemma 2 and Lemma 3 in our context, we find although a market with equal budget for all players has a good fairness guarantee, its efficiency can be low ($1/\sqrt{N}$ of optimal allocation in the worst case). In the following sections, we study how budget assignment across players affects the theoretical bound of efficiency and fairness, and how to utilize such theory to design a budget re-assignment scheme to trade off efficiency and fairness systematically.

3.3 Theoretical Results

In this section, we introduce two new metrics that will serve our goal: *Market Utility Range (MUR)* and *Market Budget Range (MBR)*. By measuring MUR and MBR in the market equilibrium, we can quantitatively understand the bound of loss in efficiency and fairness. In addition, MUR and MBR can be used as a guidance to adjust the budgets across players so that we can control the trade-off between efficiency and fairness more effectively.

3.3.1 Efficiency

According to Equation 3.4, in a market with a budget constraint, if player i bids optimally to maximize its utility, its marginal utility of bids $\frac{\partial U_i}{\partial b_{ij}}$ is a player-specific value λ_i, identical for all resources j with non-zero bids. Our intuition is that, the larger the λ_i *variation* across players, the higher "potential" there is to increase system efficiency by budget re-assignment, and therefore the lower PoA efficiency guarantee the current market has.

Consider the following examples: Assume a budget-constrained market with two players (A and B), such that $\lambda_A = 1$ and $\lambda_B = 3$ in equilibrium. It is intuitive that if the market moves 1 unit of budget from A to B, the market's overall efficiency in equilibrium (which is the sum of A's and B's utilities) is likely to increase (e.g., by 2 units, -1 from A and +3 from B). Consider, instead, that $\lambda_A = 1$ and $\lambda_B = 2$. In that case, the same budget re-assignment also points toward a market efficiency increase, but the improvement may be lower than in the first case. Finally, consider that λ_A and λ_B are equal. In that case, the intuitive expectation is that a budget re-assignment will not have an effect in overall market efficiency.

Consequently, let's define *Market Utility Range (MUR)* as follows:

Definition 5. *Maximum Utility Range is the maximum variation of marginal utility λ_i across the market players, $MUR = \frac{\min_i \lambda_i}{\max_i \lambda_i}$*

By using such definition, we can prove that:

Theorem 1. *If $MUR \geq 0.5$, the Price of Anarchy of the market equilibrium $PoA \geq (1 - \frac{1}{4MUR}) \geq 0.5$, i.e., the overall market efficiency is guaranteed to be at least 50% of optimal allocation; If $MUR < 0.5$, $PoA \geq MUR$.*

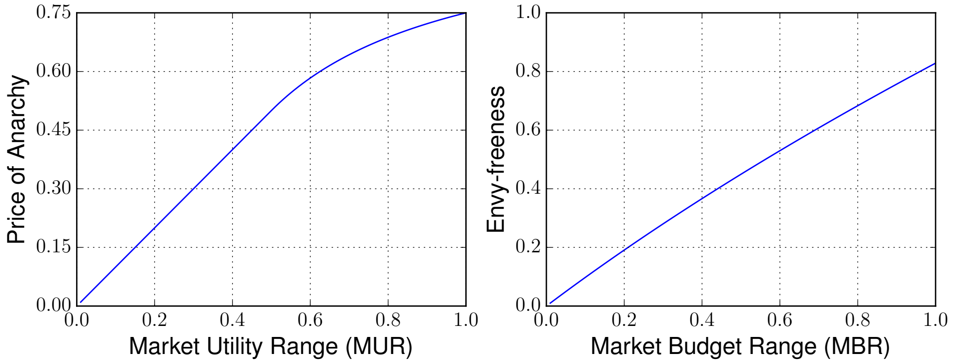


Figure 3.1: Relationship between Price of Anarchy and Market Utility Range (left), and Envy-freeness and Market Budget Range (right), based on Theorem 1 and Theorem 2, respectively.

(The detailed proof can be found later in Section A.1.)

Not only does MUR provide a lower bound of overall market efficiency, it can also be used to guide budget re-assignment to help improve the overall market efficiency. As shown in the examples above, by moving a portion of budget from a player i with lower λ_i to another player i' with higher $\lambda_{i'}$, MUR moves toward 1.[1] As a result, the PoA guarantee increases, and the actual market efficiency hopefully increases accordingly.

3.3.2 Envy-freeness

A likely side effect of assigning different budgets to different players is that it may impact fairness negatively. It is straightforward that the player with the highest budget is able to purchase more resources than others, and therefore it is likely to be "envied" by others. Hence, we hypothesize that a valuable indicator of envy (or envy-freeness) of a market in equilibrium is the variation

[1] It is provable that player i's marginal utility of bids λ_i decreases monotonically with a larger budget.

of the budget across players:

Definition 6. *Market Budget Range is the maximum variation in budget across players, $MBR = \frac{\min_i B_i}{\max_i B_i}$.*

Note that MBR is defined as the minimum budget divided by the maximum budget, so that larger budget variation means lower MBR value. Based on this definition, we can prove the following:

Theorem 2. *A market equilibrium with budget range MBR is $(2\sqrt{1+MBR}-2)$-approximate envy-free.*

(The detailed proof can be found in Section A.2.)

The key insight here is that, by combining Theorem 1 and Theorem 2, we can attempt to adjust the trade-off between efficiency and fairness: As Figure 3.1 shows, the more aggressively we re-assign the budget to make MUR closer to 1, the higher system efficiency we may achieve. However, it creates a larger variation in players' budgets, which in turn may hurt fairness. Note that such budget re-assignments do not *guarantee* an actual improvement in either the efficiency or the envy-freeness. Nevertheless, our expectation is that, by tightening the efficiency/envy-freeness bounds according to our MUR and MBR theorems, the resulting allocation at equilibrium will tend to move in the desired direction. Therefore, we envision that an algorithm, by using MUR and MBR together, can try to fine-tune a market's trade-off between efficiency and fairness. We show one such algorithm ReBudget in Section 3.4.2.

Finally, recall Zhang studies the *Price of Anarchy* in a *balanced* market [112], but he doesn't show its *envy-freeness* guarantee. But we can now prove that, in the context of multicore chips (described in Section 3.4.1):

Theorem 3. *In a balanced market, where for any player i, $U_i(\mathbf{C}) = 1$, $X_i = \sigma(U_i(\mathbf{C}) - U_i(\mathbf{0}))$, the market equilibrium is guaranteed to be 0.718-approximate envy-free.*

(The detailed proof can be found in Section A.3.)

3.4 ReBudget Framework

In this section, we first describe the basic framework for market-based resource allocation. We then describe ReBudget, a practical heuristic based on the theoretical results in Section 3.3 to assign budgets across players, so that an adjustable trade-off between system efficiency and fairness can be accomplished.

3.4.1 Market-based Approach

Our basic market-based resource allocation framework, described in Section 3.2, is a dynamic proportional market. In this framework, the goal is to find a market equilibrium using an iterative bidding–pricing procedure, after which resources are allocated proportionally to bids. This mechanism is detailed in Section 3.2.1.

Shared cache space and on-chip power are two of the most frequently targeted resources in the literature [15, 18, 70, 79, 93, 106], and our evaluation of ReBudget will focus on these two resources. Our mechanism, however, is a general framework: As long as the resource's utility function can be accurately modeled, and such utility function is non-decreasing, continuous, and concave (or can be made concave), the results of this thesis can be applied. (Note that prior studies show that the utility of cache is often non-continuous—e.g., if par-

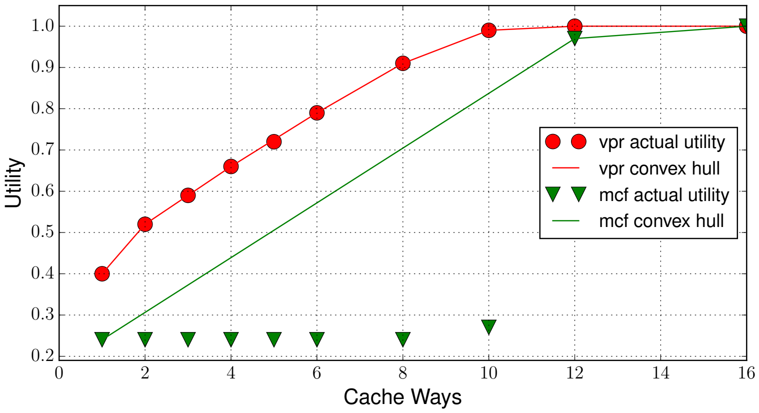


Figure 3.2: Normalized utility under different cache allocation, running at the highest possible frequency. The x-axis is the number of cache ways enabled. Section 3.5 describes the setup.

titioned by cache ways—and non-concave [8, 103], which is not consistent with our theoretical assumptions. We describe how to address such issue later in Section 3.4.1.)

At any point in time, we guarantee that each core will be given a minimum amount resources: one cache region (128 kB), and the power to run at minimum frequency (800 MHz in our setup, Section 3.5) for free. The remaining cache capacity and power budget are allocated using market-based mechanism. This is to guarantee that each application is able to at least run regardless of its purchasing power.

We now address the two major challenges in designing the market: how to model the players' utility, and how the players bid to maximize their utility.

Utility Function

In the multicore resource allocation problem that we study, we define an application's utility to be its IPC, normalized to the IPC when it's running alone (and thus owns all the resources): $U_i(\mathbf{r_i}) = \mathrm{IPC}(\mathbf{r}_i)/\mathrm{IPC}_{\mathrm{alone}}$. Thus, the possible values of U_i are between 0 and 1. To figure out the performance–resource relationship of the utility function, we adopt the monitoring technique of our recent XChange work [103]: We divide the total execution time of an application into compute and memory phases. The length of the memory phase under different cache allocations is estimated using UMON shadow tags [79] and a critical path predictor [70]. The length of the compute phase and the corresponding power consumption is estimated using the power model developed by Isci et al. [18]. The sum of both phases is an estimation of the execution time given a particular cache-power allocation. Note that this is all modeled dynamically online; no prior off-line profiling is needed whatsoever [103].

Recall that in Section 3.2, in order to apply the theoretical results, the utility function of a player is required to be concave, continuous, and non-decreasing in shape. However, in computer architecture, this is not always true. On the one hand, power is known to be concave [18,70], and fine-grained enough to regard it as continuous. For example, Intel's RAPL technique allows setting the CPU power at a granularity of 0.125W [51]. On the other hand, it is well-known that cache capacity is a non-concave, and non-continuous resource [8,79].

Figure 3.2 shows the cache utilities of two representative applications, *mcf* and *vpr*. The markers are the utilities (normalized IPC) of each application when it is given different cache ways (no change in power budget). From the figure, we can make two observations:

First, such a utility function is not continuous, as it is partitioned by cache ways which are relatively coarse-grained. To make it continuous, we adopt *Futility Scaling* by Wang and Chen [101], a feedback control mechanism that can precisely keep the partition size close to a target at the granularity of cache lines. We empirically set the allocation granularity to 128 kB, and we call this a *cache region*.

Second, cache utility may not be concave. Although *vpr* shows a concave utility function, *mcf* clearly is not: its normalized utility is flat at 0.2 for 1 to 10 cache ways, and suddenly increases to 1.0 once it secures 12 ways (1.5MB). This is because *mcf*'s working set size is 1.5 MB, and 12 cache ways or more will satisfy its need by reducing the L2 cache miss rate to be almost zero. To address this issue, we apply *Talus*, a technique to convexify cache behavior [8]. Talus works roughly as follows: First, based on an application's actual cache utility, its "convex hull" is derived, which is the convex set of the cache utility. The cache allocation points on the hull are called "point of interest" (*PoI*), which are the desired allocations. Next, to make cache utility continuous on the convex hull, Talus divides the cache partition of a core into two "shadow" partitions. Given an arbitrary cache partition target, Talus first finds its two neighboring *PoIs*, and then adjusts the size of the shadow partitions accordingly. The access stream is also divided correspondingly into the two shadow partitions. More details can be found in Talus [8]. As shown in Figure 3.2, Talus effectively convexifies the cache behavior to a convex hull, which satisfies the requirement of being concave and non-decreasing.

Bidding Strategy

Now that we have constructed a utility function for each player, according to the market's bidding–pricing procedure (Section 3.2), the next problem is how each player finds its optimal bids to maximize its utility. Because both cache and power utilities are concave, heuristics such as hill climbing are appropriate in finding optimal solutions [8]. Therefore, we adopt a simple hill-climbing technique as follows:

1. Each player i splits its budget B_i into equal bids b_{ij} across all resources. In addition, S, which is the amount by which a player will shift its budget across resources, is set to be half of the bid.
2. Each player i computes the marginal utility λ_{ij} of all resources j. According to the optimality condition in Equation 3.4, if a player's bids are such that they maximize the player's utility, then the marginal utilities of all resources which receive non-zero bids are necessarily identical—in other words, the player has no incentive to re-allocate its budget across resources. Otherwise, if λ_{ij} varies for different resources under the current bids, the player will move an amount S of money from a resource k with lower λ_{ik} to another one k' with higher $\lambda_{ik'}$, and such a move will tend to equalize the marginal utility of these two resources (recall that the player's utility function is concave, which means that marginal utility λ_{ij} decreases as bid b_{ij} increases).
3. S is halved, and the process is repeated, until one of the following two conditions is met: *(a)* Either λ_{ij} of the resources stays the same (within 5% difference); or *(b)* S is smaller than 1% of the total budget. Because S decreases exponentially with every step, and λ_{ij} is monotonic, such an algorithm will quickly reach an optimal bids to the resources.

3.4.2 Budget Re-assignment Algorithm

The mechanism discussed so far applies to individual players in the context of a general budget-constrained market-based mechanism. In this section, we describe ReBudget, a heuristic that works on top of the above mechanism. ReBudget assigns different budgets to players in an adjustable manner, so that a trade-off between system efficiency and fairness can be made.

As is discussed Section 3.3, MUR and MBR are good metrics to indicate multicore efficiency and fairness. Using MUR, we can identify the lower bound of system efficiency, as shown in Theorem 1. Moreover, due to the concavity of utility, player i's λ_i increases if its budget B_i is reduced. As a result, by reducing the budget of a player with low λ_i, system MUR, which is the maximum variation of marginal utility λ_i across the market players, will move closer to 1, potentially yielding a higher efficiency. On the other hand, however, by creating a larger budget variation across players, MBR will decrease (recall from Section 3.3.2 that larger budget variation will decrease MBR), therefore opening the door for the level of fairness (i.e., envy-freeness) to decrease as well.

In ReBudget, we attempt to maximize efficiency while guaranteeing a certain level of fairness. The system administrator can set a lowest acceptable envy-freeness level, and using Theorem 2, the minimum MBR can be computed. Then, the budgets of the players are re-assigned based on their λ_i value under market equilibrium: those with lower λ_i will get a reduction in budget. We define a player to be "low λ_i" if its λ_i is smaller than 50% of the maximum λ_i—recall that, in Theorem 1, we find that when MUR is smaller than 0.5, the PoA guarantee starts to decrease linearly. However, at any point in time, the budget variation across players has to be maintained higher than the set MBR value.

We design an iterative method with exponential back-off:

(1) MBR is computed based on the lowest acceptable fairness level set by the administrator. To start the bidding process for the market, each player is assigned an equal budget B. The amount of budget re-assignment, named *step*, is initialized to be $(1 - \text{MBR}) \cdot \frac{B}{2}$. (2) Players then use their budget to conduct the algorithm we described in Section 3.4.1 to reach a market equilibrium. (3) λ_i of each player is collected. If a player i's λ_i value is lower than 50% of the maximum λ_i in the market, its budget is reduced by one *step*. (4) *step* is halved, and the algorithm returns to (2) to find a market equilibrium again. When *step* is smaller than 1% of each player's initial budget, or when no player's budget is decreased, the resulting market equilibrium is the final outcome.

This algorithm has two advantages: (1) The highest possible budget of any player is B, and the lowest possible budget is $\text{MBR} \cdot B$ (if the player gets a budget decrease in all iterations). Therefore, the maximum variation of players' budget will stay within MBR, and the fairness level set by the designer is guaranteed. (2) The exponentially decreasing *step* ensures that the process is fast, so that the market is still efficient and scalable to deal with large-scale systems.

The above exponential back-off greedy algorithm is a show case of how MUR and MBR can be used to guide budget re-assignment to optimize the system. As we will show in Section 3.6, such an algorithm is in fact fast and efficient in practice.

3.4.3 Implementation

We now discuss the hardware and software implementation of ReBudget. On the hardware side, ReBudget requires:

1. Hardware monitors to model an application's utility–resource relationship. As is discussed in Section 3.4.1, we adopt the same monitoring hardware as we do in XChange. As a result, the overhead is 3.7 kB per core [103], and the total overhead of the monitors is less than 1% of the total cache.
2. Extra states per partition and per cache line for partitioning the cache. We adopt Futility Scaling to partition the L2 cache. and it incurs around 1.5% storage overhead of the total cache [101].

On the software side, in order to handle the changing resource demands due to context switches and application phase changes, our budget re-assignment algorithm described in Section 3.4.2 is triggered every 1 ms to re-allocate resources. Similar to XChange, such an algorithm can be piggybacked to the Linux kernel's APIC timer interrupt, with a low runtime overhead [103].

3.5 Experimental Methodology

3.5.1 Architectural Model

We evaluate ReBudget using SESC [81], a highly detailed execution-driven simulator, which we modified in-house to suit our experimental setup. We model

4-way out-of-order cores; Table 3.1 shows the most important parameters of the CMP. We also faithfully model Micron's DDR3-1600 DRAM timing [49].

We use Wattch [16] and Cacti [87] to model the dynamic power consumption of the processors and memory system. We adopt Intel's power management approach for Sandy Bridge [82] to approximate the static power consumption as a fraction of the dynamic power that is exponentially dependent on the system temperature [21]. The run-time temperature of the chip multiprocessor is estimated using Hotspot [89] integrated with SESC.

Our multicore chip is able to regulate two shared on-chip resources: power budget and shared last-level cache. The power budget is regulated via per-core DVFS, similar to Intel's RAPL technique [51]. Each core can run at a frequency between 800 MHz and 4 GHz, as long as the total power consumption remains within p×10 W, where p is the number of processor cores. The last-level (L2) cache is partitioned using Futility Scaling by Wang and Chen [101], at the granularity of 128 kB (one cache region). The total L2 cache capacity is p×512 kB. Due to the overhead of UMON shadow tags [79], we limit its stack distance to be 16, i.e., the shadow tags can estimate the miss rate for the cache with capacities from 128 kB to 2 MB. We empirically observe that very few of our applications benefit from cache regions larger than 2 MB, and even if they do, such a large cache region is usually not affordable given the limited budget of each player. With a dynamic sampling rate of 32, the shadow tags take up 3.6 kB per core, which is less than 1% of the L2 cache size.

We guarantee that each core will have at least one cache region, and sufficient power budget to allow it to run at the minimum frequency (800 MHz). The remaining resources will be entirely distributed (i.e., no leftovers) based on a

Table 3.1: System configuration.

Chip-Multiprocessor System Configuration	
Number of Cores	8 / 64
Power Budget	80 W / 640 W[a]
Shared L2 Cache Capacity	4 MB / 32 MB
Shared L2 Cache Associativity	16 / 32 ways
Memory Controller	2 / 16 channels
Core Configuration	
Frequency	0.8 GHz - 4.0 GHz
Voltage	0.8 V - 1.2 V
Fetch/Issue/Commit Width	4 / 4 / 4
Int/FP/Ld/St/Br Units	2 / 2 / 2 / 2 / 2
Int/FP Multipliers	1 / 1
Int/FP Issue Queue Size	32 / 32 entries
ROB (Reorder Buffer) Entries	128
Int/FP Registers	160 / 160
Ld/St Queue Entries	32 / 32
Max. Unresolved Branches	24
Branch Misprediction Penalty	9 cycles min.
Branch Predictor	Alpha 21264 (tournament)
RAS Entries	32
BTB Size	512 entries, direct-mapped
iL1/dL1 Size	32 kB
iL1/dL1 Block Size	32 B / 32 B
iL1/dL1 Round-Trip Latency	2 / 3 cycles (uncontended)
iL1/dL1 Ports	1 / 2
iL1/dL1 MSHR Entries	16 / 16
iL1/dL1 Associativity	direct-mapped / 4-way
Memory Disambiguation	Perfect

[a] We anticipate multicore chips with different number of cores will not be fabricated under the same technology, and thus expect the power consumption per core to decrease with the technology. For simplicity, in our evaluation, we use a chip TDP of 10 W per core and a 65-nm power model.

market-based resource allocation decision.

Workload Construction

We use a mix of 24 applications from SPEC2000 [91] and SPEC2006 [92] to evaluate our proposal. Each application is cross-compiled to MIPS ISA with -O2 optimization using gcc 4.6.1. For all simulations, we use Simpoints [17] to pick the most representative 200-million dynamic instruction block of each application.

Our evaluation is based on multiprogrammed workloads because we anticipate to allocate resources at the granularity of cores. For multithreading workloads, we can still allocate the resources at thread granularity if each thread is running on a different core.[2] Another choice is to allocate resources at the granularity of applications. All the threads of one application may share the same resources, which is a reasonable assumption, because the demand of the threads tend to be similar across threads of a parallel application in many programming models.

To construct our multiprogrammed workloads, we classify the 24 applications into four classes based on profiling: *Cache-sensitive* (C), *Power-sensitive* (P), *Both-sensitive* (B), and *None* (N). Then, we create six categories of multiprogrammed workloads: *CPBN*, *CCPP*, *CPBB*, *BBNN*, *BBPN*, and *BBCN*. For each category, we randomly generate 40 workloads for 8- and 64-core configurations. The random generation works as follows: for an 8-core (64-core) configuration, 2 (16) applications are randomly selected from each application class (e.g., *CPBN* means 2 (16) applications in each of *C*, *P*, *B*, and *N*, whereas *CCPP* will have 4 (32) applications in C and 4 (32) in *P*).

[2] Skewing resources within threads in a multi-threaded application (e.g., to alleviate synchronization imbalance) is beyond our scope, can be incorporated orthogonally to our approach, and has been studied elsewhere [11,35].

Performance Metrics

A key issue in resource allocation is the figure of merit. As is discussed in Section 3.4.1, we define an application's utility to be its IPC normalized by the IPC when it's running alone: $U_i(\mathbf{r_i}) = \mathrm{IPC}(\mathbf{r}_i)/\mathrm{IPC}_{\mathrm{alone}}$. The system efficiency can therefore be computed as:

$$\text{Efficiency} = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{\mathrm{IPC}_i(\mathrm{r}_i)}{\mathrm{IPC}_i^{\mathrm{alone}}} \tag{3.5}$$

We realize that this is exactly *weighted speedup*, a common metric to measure system throughput, and has been widely accepted by the community. Therefore, our *Price of Anarchy* study becomes meaningful in computer architecture: it guarantees the lower bound of throughput in market equilibrium.

A system could achieve high throughput (i.e., weighted speedup) by starving one or two applications while benefiting all the others; however, system fairness would suffer as a result, providing a bad experience to some users. Therefore, we evaluate fairness using *envy-freeness* shown in Definition 3, which is a widely used metric in economy, and has recently been introduced by Zahedi and Lee to evaluate fairness in multicore chips [111].

3.6 Evaluation

We evaluate our proposal in two phases. In the first phase, we analytically study the effectiveness of ReBudget by assuming that the applications' utility functions can be perfectly modeled and convexified. For this phase only, we extensively profile each application using our simulation infrastructure. (Recall that

ReBudget does not require off-line profiling; our second phase models utility functions at run-time using hardware monitors, as described in Section 3.4.1.)

We sample 90 cache+power configuration points, with {1-6, 8, 10, 12, 16} cache regions (10 possible allocations) and {0.8,1.2,1.6,. . ., 4.0} GHz (9 possible allocations). For each point, we collect average IPC and power consumption.[3] Then, we derive the convex hull of cache and power, and assume that their utilities are perfectly concave and continuous. Finally, we analytically evaluate the system efficiency and fairness by applying ReBudget and other competing mechanisms to all 240 bundles.

In the second phase, we evaluate ReBudget in a simulated CMP environment (SESC). The application's utility is monitored at run-time, using the technique described in Section 3.4.1. We apply *Talus* [8] and *Futility Scaling* [101] to make cache behavior concave and continuous. Because of practical simulation time constraints, we randomly select one application bundle per category, and run it using detailed simulations. We use these results to validate our first phase evaluation.

We conduct all the experiments on 8- and 64-core CMP configurations, and find that the results are similar. Therefore, we omit the results for the 8-core configuration, and focus on the large-scale 64-core configuration.

The allocation mechanisms evaluated are as follows: *EqualShare*, where resources are equally partitioned among all processor cores. Two XChange mechanisms [103]:[4] *EqualBudget*, where resources are partitioned using a market-

[3] Without loss of generality, our evaluation assumes that allocating more than 16 cache regions (2 MB) does not yield any significant additional utility to any application. This is reasonable for the input sets of the applications studied, and it allows us to complete profiling in a reasonable time.

[4] Note we convexify applications' utility, which is an improvement over the original

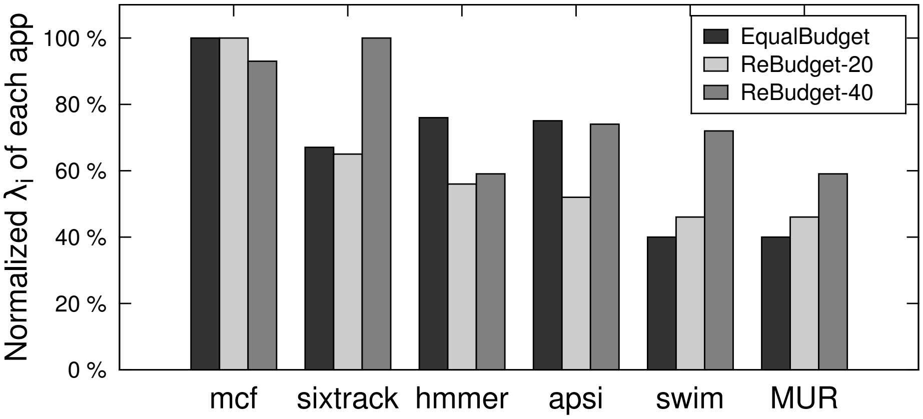
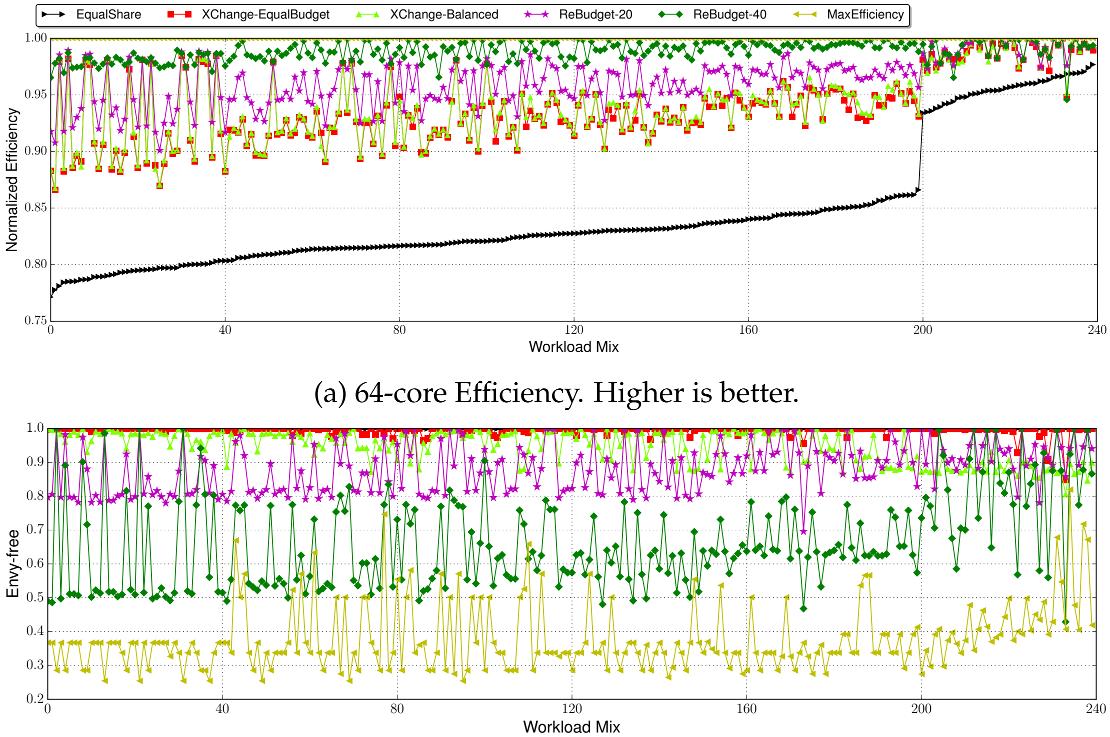


Figure 3.3: Marginal utility λ_i of each application in a sample *BBPC* bundle. The multiple copies of the same application in the bundle behave essentially the same way, so only one of each is shown. λ_i is normalized to the maximum λ_i in the bundle. MUR metric is also shown.

based procedure assuming equal budgets for all players; and *Balanced*, where each player receives a budget proportional to the utility difference between its maximum (2 MB L2 cache and 4.0 GHz frequency) and minimum (128 kB and 800 MHz) possible allocations, normalized to the former. *ReBudget-step*, where resources are allocated using our ReBudget mechanism such that, at the end of the first iteration (where all players run with equal budget), each player is assigned either its original budget or *step* less. (Recall that *step* is halved in each subsequent iteration.) The initial budget is set to be 100 for all players. Finally, *MaxEfficiency*, which is the resource allocation maximizing system efficiency, is obtained by running an infeasible very fine-grained hill-climbing search (recall that all utilities are concave).

XChange.



(a) 64-core Efficiency. Higher is better.

(b) 64-core Envy-freeness. Higher is better.

Figure 3.4: Comparison of system efficiency (weighted speedup) and envy-freeness among the proposed mechanisms in a 64-core configuration. System efficiency results are normalized to MaxEfficiency. Workloads are ordered by the efficiency of EqualShare.

3.6.1 Efficiency

For the first phase evaluation, Figure 3.4 reports the efficiency and envy-freeness for EqualShare, MaxEfficiency, as well as XChange's EqualBudget and Balanced, and ReBudget with different aggressiveness (i.e., *step*) based on run-time feedback. The bundles are ordered by the efficiency of EqualShare. We observe that the efficiency of EqualShare compared to MaxEfficiency suddenly increases for the last 40 bundles. Most of these bundles fall into the category of BBPN. EqualShare works well for BBPN workloads because 75% of the apps in the bundle are power sensitive, and equally distributing the power in EqualShare works reasonable well. In addition, although the efficiency can be improved by giving more cache to “B” apps, they are not as cache-sensitive as “C” apps. Therefore, EqualShare in cache performs better for BBPN bundles than others such as CPBN.

EqualBudget vs. EqualShare

We first compare the efficiency between EqualBudget market-based mechanism with EqualShare allocation. Figure 3.4a shows that in a 64-core configuration, 37% of the workloads in EqualBudget are able to achieve 95% of the social welfare of the MaxEfficiency, and over 90% bundles are within 90%. This proves that the market-based mechanism is robust, efficient, and scalable in this setup.

However, there are still 10% of the applications below 90% of the welfare in optimal allocation. We observe that over half of these workloads fall in the category of *BBPC*, where the number of applications favoring both resources (50%) is much higher than the number of applications favoring power (25%) or cache (25%) only. This is reminiscent of the well-known *Tragedy of Commons* [44]. MaxEfficiency strongly favors the apps which prefer only one resource, and the “B” apps, which need both resources, are sacrificed for the sake of higher system efficiency. The EqualBudget mechanism allows all the applications to fairly contend with each other, even though the price is an efficiency that is not as good as MaxEfficiency’s.

We look closely at an 8-core experiment using a BBPC bundle that contains four “B” apps (*apsi* and *swim*, 2 copies each), two “C” apps (2 copies of *mcf*), and two “P” apps (*hmmer* and *sixtrack*). The overall efficiency of such bundle with EqualBudget is 90% of MaxEfficiency, and we find its MUR = 0.40. Figure 3.3 shows λ_i value of each app at market equilibrium. We can find that with EqualBudget, "B" app *swim* has the lowest λ_i value, indicating that it is over-budgeted and not use its money efficiently; on the other hand, "C" app *mcf* has the highest λ_i value, showing that a budget increase can lead to a high utility gain.

XChange-Balanced

XChange's Balanced budget assignment is an intuitive way to distribute budget among players to improve efficiency [103]. However, Figure 3.4a shows that it does not outperform EqualBudget in efficiency too much, but in fact it loses in fairness, as shown in Figure 3.4b. The reasons are: (1) With the exception of "N"-type apps, which are not sensitive to any resources, the performance difference between minimum and maximum utility of most apps are similar, especially when we give out minimum resources for each player for free. Therefore, the budget assignment is not very different from EqualShare. (2) Blindly setting the player's budget B_i proportionally to his "potential," while ignoring the shape of utility and MUR metric, is ineffective.

ReBudget

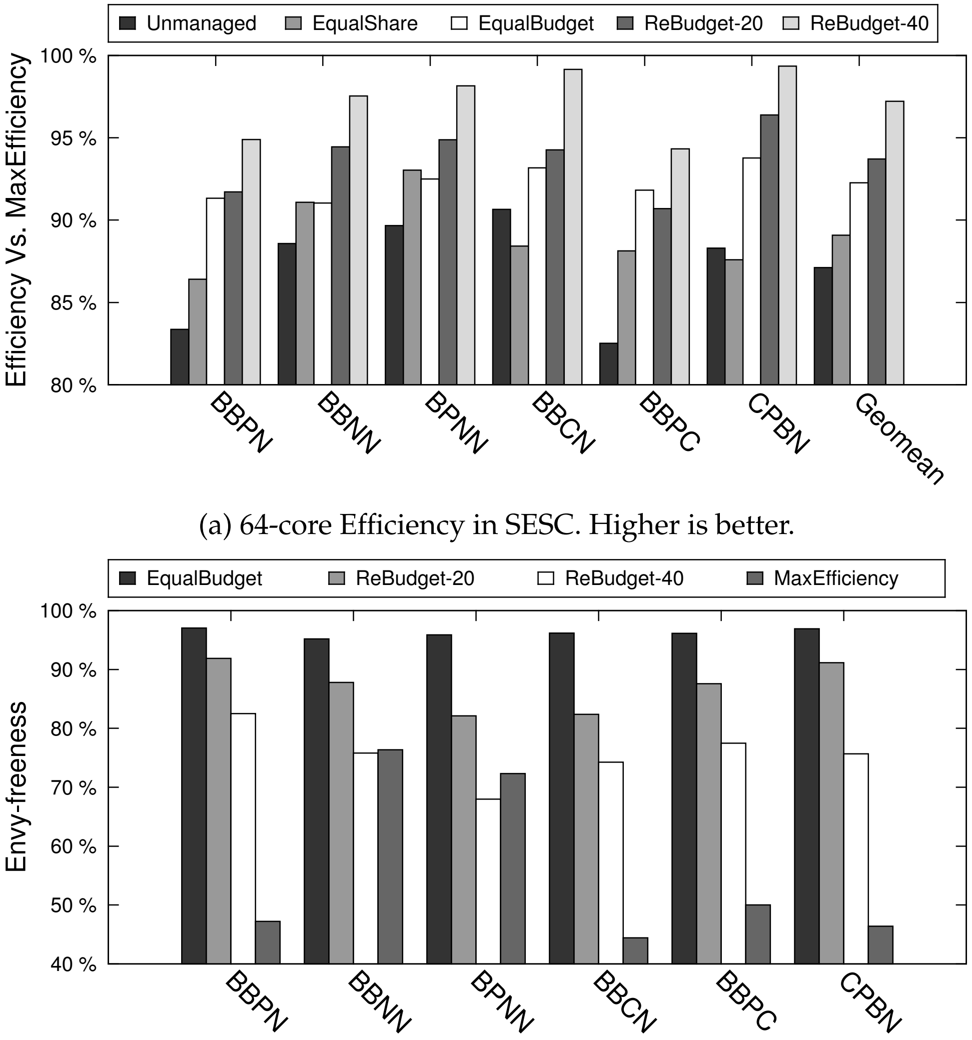
We evaluate the our ReBudget mechanism proposed in Section 3.4.2. We test different aggressiveness, by setting the amount of budget decrease at each iteration (i.e., *step*) to be 20 and 40. Figure 3.4a clearly shows that by re-assigning the budget more aggressively, efficiency will improve for all bundles. Also, for all the 240 bundles, ReBudget-40 achieves 95% of system efficiency of MaxEfficiency.

We look closely at the same 8-core bundle we study in Section 3.6.1. For ReBudget-20, Figure 3.3 shows that *swim*, whose λ_i at 0.40 is the lowest under EqualBudget, has its value increased to 0.46, because its budget drops from 100 (every player's initial budget) to 61.25 units (minimum budget under ReBudget-20). On the other hand, *mcf*, which has the highest λ_i, has its budget unchanged (100). The budget of other apps are lowered to around 80, because their λ_i values are significantly lower than *mcf*. Note that λ_i of *apsi* and *hmmer* decrease, even though their budgets are reduced. This is because the budget cut of *swim* makes the prices of resources drop significantly. As a result, although *apsi* and *hmmer*'s budgets are reduced, they can actually afford more resources, and their λ_i decreases. Overall, the MUR of ReBudget-20 increases to 46%, compared to 40% in EqualBudget. Correspondingly, the efficiency of ReBudget-20 increases to 96% of MaxEfficiency.

In ReBudget-40, *swim*, whose λ_i is still the lowest in ReBudget-20, get a further budget cut to 20. As a result, its λ_i value increases from 0.46 to 0.72, as shown in Figure 3.3. *mcf* in this case is no longer the highest in λ_i value: *sixtrack*'s budget is decreased to 30 and it starts to request for more money. Therefore, MUR of the system is increased to 0.59, and the efficiency is now 99% of MaxEfficiency.

3.6.2 Fairness

We use envy-freeness as the metric to evaluate the fairness, as is discussed in Section 3.2.3. We first look at the fairness comparison between EqualBudget



(a) 64-core Efficiency in SESC. Higher is better.

(b) 64-core Envy-freeness in SESC. Higher is better.

Figure 3.5: Comparison of system efficiency (weighted speedup) and envy-freeness among the proposed mechanisms in a simulated 64-core configuration. System efficiency results are normalized to MaxEfficiency.

and MaxEfficiency. As expected, Figure 3.4b shows that EqualBudget is almost envy-free, where in the worst case, it is still 0.93-approximate envy-free. On the contrary, MaxEfficiency is unfair, which is typically 0.35-approximate envy-free.

Regarding the XChange-Balanced mechanism, the envy-freeness of most workloads stays at 0.9, where in the worst-case it is 0.86-approximate envy-free. It is not as good as EqualBudget, and considering its trivial efficiency gain, and no control over the aggressiveness in making trade-off between efficiency and fairness, we consider such a mechanism to be ineffective.

On the other hand, the envy-freeness of ReBudget has a direct relationship with its aggressiveness. Figure 3.4b shows that the typical envy-freeness of ReBudget-20 and ReBudget-40 are 0.8 and 0.5, respectively, and none of the bundles violates the theoretic guarantee provided by Theorem 2 (0.53 and 0.19). We notice that there is a gap between the theory and reality. This is because what Theorem 2 states is a theoretic lower bound, which should stand in all cases. Such bound is tight, and it is not hard to construct a market to reach it[5]. Although it does not happen on the applications we use, it could happen in the real life. In addition, we show the envy-freeness of ReBudget-20 is consistently higher than ReBudget-40 for all bundles. Therefore, besides the theoretic guarantee, MBR can be used as an accurate indicator of system fairness.

Combined with the findings in Section 3.6.1, we can conclude that the more aggressive budgets are adjusted, the higher efficiency, and correspondingly the lower fairness it is achieved. This is appealing, because system designers and administrators can use the *step* as a "knob" to trade off one for the other.

3.6.3 Simulation Results

Besides the above analytical results, we implement ReBudget in architectural simulator SESC. Figure 3.5 shows the system efficiency and envy-freeness of the competing mechanisms. Such results are consistent to our analytical evaluation above: ReBudget improves system efficiency over EqualBudget by sac-

[5] Zhang shows an example in EqualBudget case [112].

rificing fairness, and the more aggressive budget is re-assigned, the more efficiency improvement it is achieved. On the other hand, EqualBudget achieves the highest in envy-freeness, and MaxEfficiency, which targets at maximizing system efficiency, is the worst in fairness. ReBudget successfully maintains its rank between these two extremes, and aggressiveness gradually hurts system performance as expected.

3.6.4 Convergence

A very important aspect of the market-based mechanism is how fast our algorithm is in finding the equilibrium allocation. To the best of our knowledge, there is no theoretic lower bound on the convergence time. However, in reality, we find that EqualBudget and XChange-Balanced converge within 3 iterations for 95% of the bundles. ReBudget mechanism spends a few more iterations, because it needs to re-converge after budget adjustment. However, the exponential back-off in budget change guarantees that the ReBudget process converges fast. These findings are in line with prior studies (e.g., Feldman et al. find the convergence time for a dynamic market is ≤ 5 iterations [40]). We also adopt a fail-safe mechanism for the very rare cases that market cannot converge: we simply terminate the equilibrium finding algorithm after 30 iterations.

3.7 Summary

In this chapter, we have introduce two new metrics, *Market Utility Range (MUR)* and *Market Budget Range (MBR)*, which help us establish a theoretical bound for the loss of efficiency and fairness of a market equilibrium under a constrained budget, respectively. We have proposed *ReBudget*, a budget re-assignment technique that is able to systematically control efficiency and fairness in an adjustable manner. We evaluate ReBudget on top of our earlier XChange proposal for market-based resource management in CMPs, using a detailed simulation of a multicore architecture running a variety of applications. Our results show that ReBudget is efficient and effective. In particular, when combined with the proposed MUR and MBR metrics, ReBudget is effective at maximizing efficiency under worst-case fairness constraints.

CHAPTER 4

SWAP: EFFECTIVE FINE-GRAIN MANAGEMENT OF SHARED LAST-LEVEL CACHES WITH MINIMUM HARDWARE SUPPORT

4.1 Introduction

Performance isolation is an important goal in server-class environments for a variety of reasons, including throughput, quality of service, and even security. Partitioning last-level caches in chip multiprocessors (CMPs) across applications is a popular approach to reducing or eliminating interference across applications co-running on a CMP. It is a mechanism that can help (1) maximize resource utilization and system throughput, or trade off throughput vs. fairness [103,104]; (2) provide quality-of-service (QoS) for latency critical workloads [66]; (3) protect the system from timing channel attacks, where a malicious program is able to steal the secure information of another application, such as the encryption key, by sharing the last-level cache [10]. A few approaches to partitioning the cache space have been proposed.

Way partitioning allows cores in chip multiprocessors (CMPs) to divvy up the last-level cache's space, where each core is allowed to insert cache lines to only a subset of the cache ways. It is a commonly proposed approach to curbing cache interference across applications in chip multiprocessors (CMPs) [79]. Unfortunately, way partitioning is proving to be not particularly scalable, as it affects cache latency and power negatively, eventually becoming impractical. Consider that multiple current and upcoming server chip multiprocessor (CMP) lines already comprise twenty, thirty, or even more cores; examples include Intel's 22-core E5-2600 v4, IBM's 24-core Power-9, Cavium's 48-core ThunderX, or Qualcomm's 64-core Hydra. Although some of these processors do include hardware support for way partitioning, the granularity is too coarse to allow for separate partitions for more than a handful of applications. Cavium's ThunderX processor, for example, possesses 48 cores, however its last-level cache is limited to "only" 16 ways. Similarly, Intel's v4 CMP allows for no more than 20 different partitions across 22 cores.

Another approach to achieving cache partitioning is to restrict each application's page frames to certain "colors" (the shared bits between a physical address' page frame ID and cache index). In this case, page frames of each color map onto a specific subset of the cache sets. Although this approach has been adopted in real operating systems [62, 64, 109], it also does not scale beyond a handful of colors.

A few architectural mechanisms for probabilistic fine-grain cache partitioning have been proposed [67, 85, 101]. However, these implementations require extra hardware support, do not provide true isolation, and have not yet been adopted in any commercial CMP to our knowledge.

Contributions

We propose SWAP, a fine-grained cache partitioning mechanism that can be readily implemented in existing CMP systems. By cooperatively combining the cache way (hardware) and set (OS) partitioning, SWAP is able to divide the shared cache into literally hundreds of regions, therefore providing sufficiently fine granularity for the upcoming manycore processor generation.

We implement SWAP as a user-space management thread on Cavium's ThunderX, a server-grade 48-core processor with ARM-v8 ISA [97]. To enable SWAP, we introduce small changes to the Linux page allocator, and leverage ThunderX's native architectural support for way partitioning.

Our results show that SWAP improves system throughput (weighted speedup) by 13.9%, 14.1%, 12.5% and 12.5% on average for 16-, 24-, 32- and 48- application bundles with respect to no cache management. This is twice as much speedup as what we can obtain by using only ThunderX's way partitioning mechanism.

To our knowledge, SWAP is the first proposal of a fine-grained cache partitioning technique that requires no more hardware than what's already present in commercial server-grade CMPs.

The chapter is organized as follows: Section 4.2 provides background and comments on related work. Section 4.3 describes SWAP's mechanism and design challenges. Section 4.4 discusses the hardware and software implementation. Section 4.5 explains our evaluation framework, and Section 4.6 evaluates this SWAP proposal.

4.2 Background

4.2.1 Way Partitioning

One popular method to manage a set-associative cache is partitioning cache ways. It is a desirable approach because: (1) each core can be assigned an independent slice of the cache space, thereby reducing cache interference among co-running applications; (2) adjusting allocations is relatively inexpensive and can be accomplished lazily (i.e., cache lines in ways no longer part of an application's partition can still be accessed in place, until they are evicted). As a result, chip manufacturers have begun to adopt such a technique into their server processors [46].

Despite all these advantages, a well-known limitation of way partitioning is that it cannot by itself support more than a handful of applications at a time [84]. This is because cache associativity cannot scale easily with number of cores, as physical constraints result in increased latency and energy consumption.

4.2.2 Page Coloring

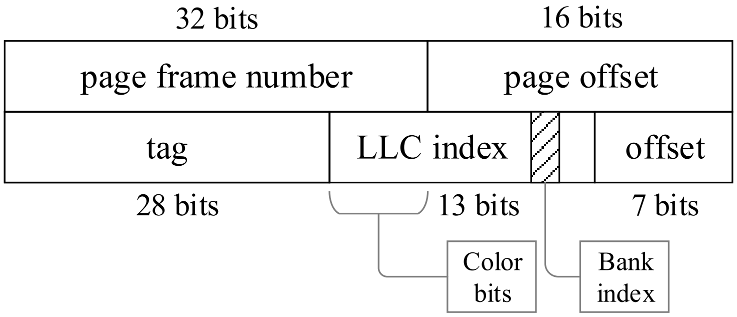


Figure 4.1: Example of physical address mapping for page coloring, corresponding to Cavium's 48-core ThunderX architecture used in this study.

Page coloring [98] has been extensively used in industry and research community to improve the performance of the memory hierarchy. Instead of partitioning the cache "vertically" as in way partitioning, page coloring partitions the cache "horizontally" by sets. When an application requests a new page from the system, the OS will select a free page from its memory pool, and map the application's virtual address to the physical address of the page. In doing so, the OS may select a page frame whose page frame number (PFN) is of the appropriate "color"—the overlapping bits between the page frame number and last-level cache's set index (Figure 4.1). By constraining the color bits of the pages belonging to an application in this way, the OS may constrain an application's cache use to a subset of the cache sets.

Unfortunately, page coloring is hardly scalable, and it can incur significant overheads if recoloring is needed. Consider, for example, that a PFN's default size of 64KB allows for four color bits in Cavium's 48-core ThunderX (Figure 4.1). Sixteen colors is hardly sufficient to provide adequate isolation across 48 cores. One might consider reducing the page size to increase the number of colors, however this is typically counterproductive in the server market [47].

Even if a small page size were practical, page coloring still may not be able to provide fine granularity by itself: A well-known limitation of page coloring is that, by imposing page color restrictions on an application, only a portion of the system memory is accessible to this application [114]. This may result in an out-of-memory (OOM) error, even though the system may be awash with pages of other colors.

Finally, re-partitioning the cache space by page coloring is a costly process: If a page color is taken away from one application, all the associated page frames have to be migrated to page frames in the application's other colors, the appropriate TLB and cache entries flushed, etc.

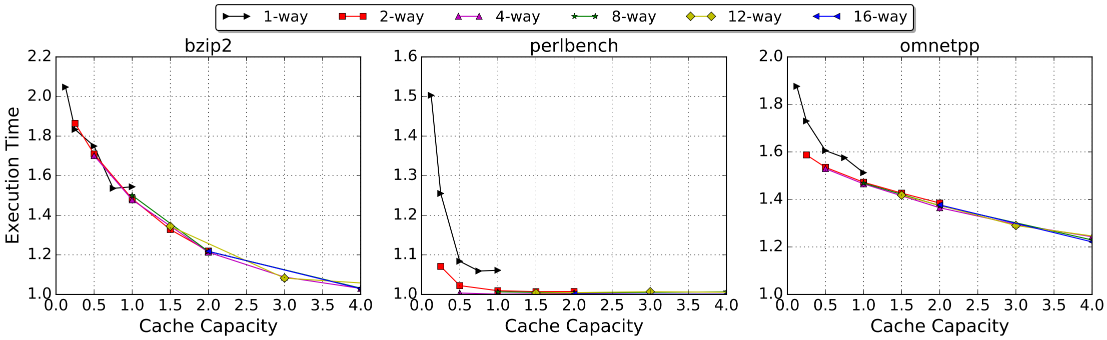


Figure 4.2: The relationship between execution time (normalized to the execution time with the entire 16MB cache) and cache capacity. Each cache capacity may be consisted with different number of cache ways and colors, which is shown in the different curves.

4.3 Mechanism

SWAP combines both set and way partitioning so that we can partition the shared last-level cache in a two-dimensional manner into many tens or even hundreds of regions, and then assign those regions to running applications. In Cavium's ThunderX 48-core processor, for example, the number of cache ways and possible page colors is 16 each. Therefore, ThunderX's shared L2 cache can be partitioned into 256 independent regions. Note that, theoretically, assignments may be chosen to overlap, but there is sufficient granularity to keep them disjoint, which is generally preferable.

4.3.1 Challenges

Although combining set and way partitioning to enable fine-grained cache partitions may be intuitive, in practice several important challenges needed to be addressed to make it practical. We discuss these next.

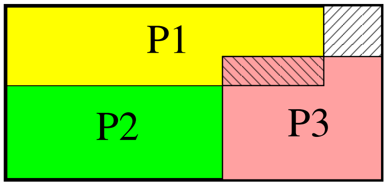


Figure 4.3: An example of misaligned cache partitions that, on the one hand, leaves some cache space unassigned while, on the other hand, it forces some assignments to overlap.

Partition placement. Correctly allocating a partition involves more than just picking the right size. On the one hand, partitioning by cache ways and page colors constrains the possible shapes and sizes of each cache partition. For example, in the ThunderX processor, it is infeasible to create a partition of 17 cache regions in the L2 cache with 16 ways and 16 colors. On the other hand, given a desired partition size, there are multiple possible combinations of sets and ways to form a rectangle with that size. For example, 4×4, 2×8, and 1×16 allocations all offer the same capacity. Even if the partition size and shape of each application is feasible and known, placement of the partitions is a challenge in its own right.

Figure 4.3 shows an example of partitions that are not successfully placed. Due to the poor alignment, there is some wasted cache space on the top right corner, and overlapping between partitions of P1 and P3, resulting in cache interference between the two. As part of our proposed solution, we describe later how we optimize the choice of partition placement.

Memory Pressure. As discussed in Section 4.2, page coloring not only limits the number of cache sets an application can use, but also the amount of physical memory that it can access. The memory system could be awash with free physical frames of a particular color, and yet those would not be available to other applications that have been assigned a different color. Because SWAP employs page coloring, it is potentially subject to this problem. Although many colors potentially enable a fine-grain management of cache set allocations, it constrains each application to a small slice of the physical memory. Fortunately, this is not a major concern for SWAP, because SWAP adopts a coarse-grained page coloring technique, achieving fine granularity by combining it with way partitioning. In the ThunderX platform we study, for example, each page color covers 4GB of the 64GB available main memory, and we assign at least two page colors to each application (Section 4.3.2). As a result, we never observed out-of-memory exceptions in any of our experiments.

Recoloring Overhead. Another major concern of page coloring is the potentially heavy cost associated with dynamic recoloring. When a color is taken away from an application, for example, all the pages with that color from that application have to be remapped across the remaining assigned colors. Page remap operations are cumbersome: they involve TLB and cache flushes, a page copy from its old memory location to the new one, and an update of the corresponding page table entry. Although efforts have been made to alleviate such overhead, for example by performing "lazy" page migration, recoloring overheads are generally non-negligible [62]. Therefore, SWAP needs to be carefully designed to avoid giving/taking away colors to/from applications whenever possible.

Increased Conflict Misses in Way Partitioning. One disadvantage of cache way partitioning is that it reduces the effective cache associativity of each partition, potentially increasing the number of conflict misses [85,101]. Because SWAP inherits this disadvantage, we investigate the relationship between execution time and the number of cache ways in the context of our experimental setup (Section 4.5), by statically sampling 30 different cache way+color configurations, with {1, 2, 4, 8, 12, 16} cache ways and {2, 4, 8, 12, 16} page colors. As a result, the effective cache capacity ranges from 128KB to 16MB. Figure 4.2 shows the execution time of each configuration, normalized to the execution time w/ 16MB cache, for three sample SPEC applications. We find that, if the cache partition is formed by only one cache way, the number of conflict misses increases dramatically, and therefore the application's execution time suffers by up to 40% increase. On the other hand, for most applications, as long as their assigned partition has more than two cache ways, their performance is largely determined by the size of the assigned partition.

4.3.2 Algorithm

We propose a novel cache allocation mechanism to address these challenges. The mechanism starts by collecting the miss-ratio curve (MRC) of each application. The way the MRC is collected, whether using offline data or an online profiler, is orthogonal to the mechanism and has been addressed elsewhere [29–31,79]. It then runs the *lookahead* algorithm proposed by Qureshi and Patt [79] to decide the optimal partition size of each core (in the unit of cache regions), so that the sum of partition size of each core is the total cache capacity. Note that we guarantee 2 regions of cache space (128KB) for each core. More details will be explained in Section 4.5. Note that the lookahead algorithm only determines the size of each partition given the total cache capacity, not how these are achieved in terms of ways vs. colors; this will be decided later by our placement algorithm, which we describe next.

Cache Partition Placement

An ideal partition should satisfy the following requirements: (1) partitions are aligned well with each other, without any wasted or overlapping cache regions, and (2) dynamic resizing should affect the fewest number of partitions during phase changes.

In SWAP, cache partitions are classified into multiple coarse-grain classes according to their size. Those in the same class are given the same number of colors. If the size of a partition changes within the range of its class, the number of colors it is given remains unchanged. The hope is that the partition may be able to keep its original page colors, to avoid any time-consuming recoloring. The general classification criterion for K colors ($K = 16$ in ThunderX) and S cache size (16MB in ThunderX), is as follows: (1) partitions of size larger than or equal to $S/4$ are afforded all K colors; (2) partitions whose size lies within $[S/8, S/4)$ are allowed $K/2$ colors; (3) partitions that fall within $[S/16, S/8)$ are assigned $K/4$ colors; and so forth, down to a minimum of two colors. In the case of ThunderX, this classification comprises four classes $C_i, i \in \{16, 8, 4, 2\}$, where i represents the number of colors assigned to applications in that class.

For placement (i.e., what specific colors each application receives), partitions with more colors are placed first and to the "left" (as represented by a rectangle of set rows by way columns) of partitions with fewer colors. Let us use an example (illustrated by the top row of Figure 4.4) to explain how this placement policy, combined with the classification criterion, can solve the alignment issue. In this example, P1 is an 8-color-class partition, and P2 and P3 are both 4-color-class partitions. Because P1 has more colors, it will always be placed to the left of P2 and P3. Thus, the right boundary of P1 and the left boundary of P2 and

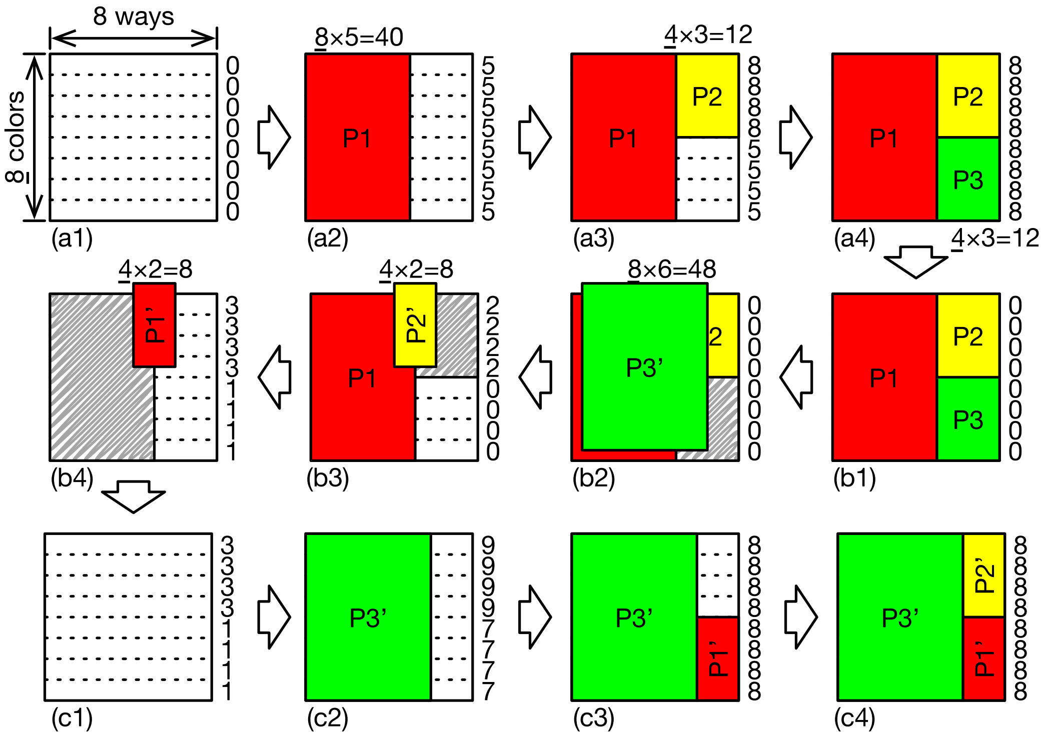


Figure 4.4: A sample process of placing partitions based on their sizes and classes. The figure assumes eight colors and eight ways. The top row show an initial partition; center and bottom rows show the process of dynamically repartitioning based on changing application demands.

P3 are aligned.

Based on these two policies, the placement algorithm works as follows:

1. Each color maintains a "usage" counter (initialized to 0, as is shown in step a1 of Figure 4.4), which measures the number of cache ways of that color which have been already assigned.

2. The partitions are first classified into different classes according to the criteria mentioned above. Then the number of cache ways is trivially computed, by dividing the partition size by the number of colors, rounded down to an integer (not necessarily a power of 2).

3. We place cache partitions, in order from larger to smaller. For each partition, SWAP tries to find a set of consecutive colors with as little usage as possible. After the set of colors is determined, the partition will update the usage counter of its assigned colors, and the algorithm will move on to the next partition. Step a2 to a4 of Figure 4.4 shows such an example with 8 cache ways and 8 colors, where the sizes of the three partitions, P1, P2, and P3, are 40, 12, and 12, respectively. Because P1 is the largest partition and is allocated more than half of the available regions, it is a C_8 partition, and is assigned all 8 colors according to the classification criterion. The usage counter of all the colors are updated to 5, since P1 receives five ways. The system then picks the top four colors for P2 (P2 is in class C_4), and it increases their usage counter by 3 each (the number of ways allocated to P2). When looking to place P3 in class C_4, the least used colors are picked, again in this case updating their counter to 3 each, since that is the number of ways allocated to P3 as well.

4. If a core is given the minimum cache space (2 regions), SWAP assigns one way and two colors to it, which may significantly hurt its performance due to conflict misses. As a result, in that case we may horizontally coalesce two or more minimum-sized partitions, using the same set of colors. Although colescing may introduce some interference, we experimentally observe that it greatly reduces conflict misses (and that this kind of applications are often cache-insensitive anyway).

4.3.3 Reducing Recoloring Overhead

Our algorithm for recoloring strives to minimize color re-assignments. Specifically: (1) if a partition stays within its class, it should stick with its prior color assignment; (2) if a partition is downgraded to a class with fewer colors, its new colors should be a subset of its prior color set, so that only the "orphaned" pages need to be migrated; (3) if a partition is upgraded so that more colors are made available to it, it should attempt to add the set of colors that have the least "pressure" (smallest usage counter) at the time the partition is (re)placed. We continue to use Figure 4.4 (middle and bottom rows) to show this repartitioning process, where the sizes of P1, P2, and P3 and changed from 40, 12, 12 regions, to 8, 8, and 48 regions, respectively.

1. The partitions are classified into different classes as before.

2. Before placing the partitions, we first reset the usage counter (step b1), and then estimate new usage for each color as follows: (1) If a partition is upgraded to a class with more colors (P3 in the example), we do not increase the usage of any color, as is shown in step b2 of Figure 4.4. This is because the new partition will explicitly seek to expand into the least used colors. (2) If a partition stays in the same class (P2), we increase the usage of each color by the number of ways the partition will receive (step b3). This is to discourage other partitions that migrate into the same class from occupying these colors during placement. The goal is to allow applications that stay in the same class to keep their colors. (3) If the partition's class is downgraded (P1), the estimated new usage of the colors it currently maps to is increased by the number of cache ways the partition will receive, multiplied by the ratio of new to old number of colors for that

partition. For example, P1 previously owned all 8 colors in Figure 4.4, it is now downgraded to a 4-color class. The usage counter of colors 0-7 will be updated by the number of ways the new partition will receive, multiplied by 0.5. The rationale here is that the new partition will subset all former colors with equal probability, so on average each such color will see its usage affected equally and proportionally to the new allocation. (Recall that at this point we still do not know *which* colors will be picked.)

3. We start placing the cache partitions from larger to smaller, following the original algorithm, only that the expected usage is already initialized as explained above, and thus not computed from zero, but adjusted during actual placement. For example, in step c3 of Figure 4.4, we assign colors 4-7 to downgraded partition P1, because those colors show lower estimated usage (since colors 0-3 are "reserved" by P2). On the other hand, as a partition that remains in its same class, P2 will again pick its former colors (step c4). After placing each partition, usage for each color is adjusted to reflect the actual usage by that partition, by compensating with respect to the estimated usage previously calculated and accounted for, as is shown in step c3.

It is possible that a partition whose class is either unchanged or downgraded may find the expected usage of its previously assigned colors high enough that the partition may not be able to get the number of cache ways it needs. In that case, we allow the partition to move to a new set of colors that can accommodate its size; specifically, the partition will seek to move to a set with minimum calculated usage.

4.4 Implementation

In this section we describe the existing hardware support that we leverage to implement SWAP, the software changes that we make to the operating system, and the interaction between them.

The ThunderX 48-core CMP is an ARM-based processor aimed at the server/datacenter market. It provides the ability to allocate the shared L2 cache by cache ways, up to 16 partitions. ThunderX provides a special register per core, which specifies the cache ways that a core can *insert* cache lines into. (Cores can still *access* lines in any cache way.) Once cache ways are assigned to cores (see Section 4.3.2), SWAP configures the per-core registers so that the assignment may be enforced.

In order to further partition the cache by sets, we implement page coloring [62, 64, 110] in the Linux kernel that runs on the ThunderX system, by modifying its buddy memory allocator to fit our needs. We color user pages only; kernel pages are allocated using Linux's default mechanism.

In the buddy system, free physical pages are stored in multi-level free lists, where the kth-order free list contains pages which is composed of 2^k consecutive 64KB pages. We create multiple *bins* out of each list, with each bin caching pages of a specific color.

When a page fault occurs to a user application, the kernel first selects a page color in a round-robin fashion among all the allowable colors for that application. Then, it fetches a page of that color. When a bin is running out of pages, SWAP requests more free pages from the Linux buddy system and uses them to refill the bins.

A potential issue with page coloring is that some of modern processors adopt *hashed indexing*, where the index to the last-level cache is XORed with bits in the physical address [63]. Fortunately, because the physical address of a free page is readily available in the kernel, its color can be easily computed by hashing the appropriate bits.

SWAP works well with the large page sizes often found in server settings– in ThunderX's case, 64KB. It can also work well with smaller page sizes (e.g., standard 4KB pages), as long as the number of bits assigned for coloring is kept small, to skirt the issues of memory pressure and recoloring overhead described before. SWAP *as is* would not be able to leverage page coloring for very large "superpage" sizes supported in some architectures (e.g., 512MB for ThunderX), as the page offset would be very long, and therefore there would be no overlap between the page number and the cache index. Very large superpages are usually relegated to the uncommon case of servers with terabytes of physical memory [50], and produce undesirable side effects [32, 54, 58, 72, 77]. For example, database vendors often recommend users to turn off large superpage support [58, 72], because many database workloads tend to exhibit sparse rather than contiguous memory access patterns. Large superpages may also cause the system to run out of memory [32].

We follow a lazy approach to page migration for dynamic recoloring [62]: When a color is added to or taken away from an application, we eagerly walk through the application's page table and redistribute the application's pages across the colors assigned to it. For each page marked for migration to a different color, we reset the access flag (AF) in the page table entries (PTE) of the

Table 4.1: CMP configuration.

ThunderX CN8800 [2, 19]	
Number of Cores	48
Frequency	2.0GHz
L1 ICache	78 kB, 128B cache line size
L1 DCache	32 kB, 128 cache line size
L2 Cache	16 MB, 16-way set associative, 128B cache line size
Memory Controller	64GB, 4 channels, DDR4 2133, aggressive bank reordering

application's pages of that color, and set one other unused bit in each such PTE (we call it the Pending bit). Naturally, the corresponding TLB and data cache entries are also flushed. However, the application's marked pages are not immediately migrated. Rather, as pages for that application with AF=0 are accessed (which generates a page fault), if the Pending is set, the page will be migrated to its new color at that point (and the Pending bit will be reset). Then, the AF bit will be set, and the page fault handler will complete.

4.5 Experimental Setup

4.5.1 Hardware Platform

We evaluate SWAP on a Cavium ThunderX CN8800 rack server. The configuration of the processor is shown in Table 4.1.

ThunderX supports hardware cache way partitioning, as is described in Section 4.4. We also develop a set of microbenchmarks similar to what Saavedra et al. [83] propose to verify the specifications related to the memory hierarchy (cache capacity, associativity, etc).

In addition, we check whether there is an overlap between the color bits and the memory channel and bank bits, as page coloring may restrict a core's accessibility to the memory channels/banks. To do this, we run the microbenchmarks proposed by Yun et al. [110] to detect the location of those bits, and we find that the memory channel and bank bits reside within the page offset, and therefore there is no overlap with the color bits.

4.5.2 Software Platform

We prototype SWAP in the ThunderX platform running Ubuntu Trusty Tahr 14.04 with kernel version 3.18.0. SWAP runs as a user space management process, which includes (1) the algorithm described in Section 4.3.2 to decide the allowable cache region of each application; (2) the ability to write hardware registers to reconfigure cache way partition, and to interact with the underlying Linux kernel for page coloring (the implementation details are described in Section 4.4); and (3) a performance tracking thread which is triggered every 2 seconds to read hardware performance counters, such as the number of L2 cache misses.

4.5.3 Workload Construction

We use a mix of 22 applications from SPEC2000 [91] and SPEC2006 [92] to create multiprogrammed workloads for evaluation. Each application is compiled natively to an ARM executable, using gcc 5.1.0 with -Ofast optimization. We clas-

Table 4.2: Multiprogrammed workloads evaluated for simulation. Combining cache-insensitive (I), cache-sensitive (S), and thrashing (T) applications.

MP1	vpr - twolf - art - lbm	S^4
	vpr - ammp - bzip2- libquantum	S^2T^2
MP2	milc - soplex - lbm - art	T^4
	leslie3d - bwaves - GemsFDTD - bzip2	T^2S^2
MP3	vpr - twolf - milc - libquantum	S^4
	ammp - bzip2 - bwaves - soplex	T^4
MP4	mcf - milc - libquantum - leslie3d	T^4
	bwaves - GemsFDTD - twolf - swim	T^4S^2
MP5	mcf - soplex - libquantum - leslie3d	T^4
	bwaves - lbm - swim - art	T^2S^2
MP6	gamess - hmmer - milc - mcf	I^4
	tonto - h264ref - lbm - art	T^2S^2
MP7	twolf - art - leslie3d - bwaves	S^4
	bzip2 - mcf - GemsFDTD - libquantum	T^4
MP8	vpr - twolf - libquantum - milc	S^4
	ammp - art - mesa - sixtrack	T^2I^2
MP9	twolf - vpr - lbm - libquantum	S^4
	bzip2 - omnetpp - mesa - gobmk	T^2I^2
MP10	milc - soplex - h264ref - vpr	T^4
	libquantum - leslie3d - perlbench - mcf	S^2I^2

sify the 22 applications into *Cache-sensitive (S)*, *Cache-insensitive (I)*, and *Thrashing (T)* using offline profiling, and then create ten 8-application bundles that consist of a mix of applications from these three categories, as shown in Table 4.2. When the number of active cores exceeds the number of applications in a bundle, the bundle is replicated across the chip. For example, 4 copies of MP1 would run in a 32-core configuration.

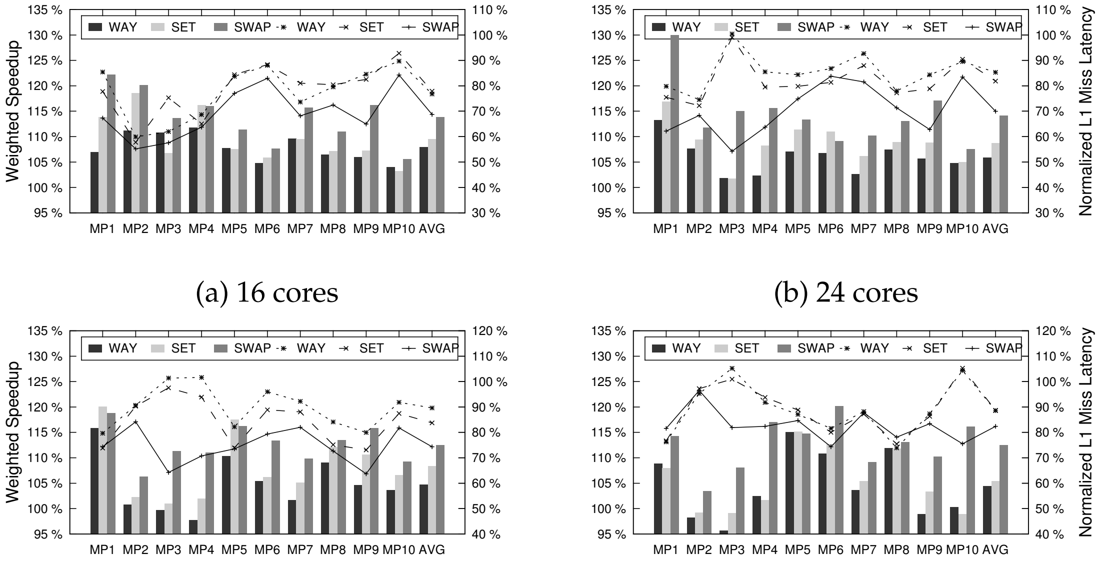
SWAP needs an estimate of the application's cache miss rate vs. capacity curve (MRC), which is used by the lookahead algorithm to produce the optimal size of each partition (described in Section 4.3.2). In server-class environments, profile information can be obtained efficiently in a variety of ways, as addressed elsewhere [29–31]. Alternatively, it could be collected using additional hardware support (e.g., UMON [79]). In this paper, we use the applications' miss-per-kilo-cycle (MPKC) profile, by sampling 30 different cache way+color configurations, with {1, 2, 4, 8, 12, 16} cache ways and {2, 4, 8, 12, 16} page colors (the effective cache capacity ranges from 128KB to 16MB).

Besides SPEC, we also use a latency-critical workload, namely *memcached* from Cloudsuite [41], to study how SWAP guarantees QoS. Due to the lack of 10Gbit Ethernet support, we run the memcached server and clients on the same chip to avoid Ethernet becoming the bottleneck. Although packets are not physically transmitted via Ethernet, they still go through most of the OS networking layers, and therefore the cache behavior of the memcached server remains the same. In addition, in order to guarantee isolation between clients and server, we allocate 2 exclusive cache ways to all the client threads, which we find is good enough to issue requests in a timely manner. As recommended by Cloudsuite, we run one instance of the memcached server with 4 threads, and the QoS target is set such that 95% of the requests are serviced within 10ms [37]. The memcached client runs with 8 threads, and we configure the issue rate to 190K requests per second[1].

4.5.4 Performance Metrics

We use weighted speedup and L1 miss latency to evaluate our fine-grained cache partition, both of which can be obtained by SWAP's performance tracking thread described above. Weighted speedup measures the overall system throughput [38]. It is the arithmetic mean of the ratio between IPC_i^{shared} and IPC_i^{alone} for all applications i, where IPC_i^{shared} is the IPC obtained while running application i in a loaded system, and IPC_i^{alone} is the IPC when running unmolested.

[1] We find that 190K is the maximum issue rate for the memcached server to meet its QoS target even if it is given the entire cache.



(a) 16 cores

(b) 24 cores

(c) 32 cores

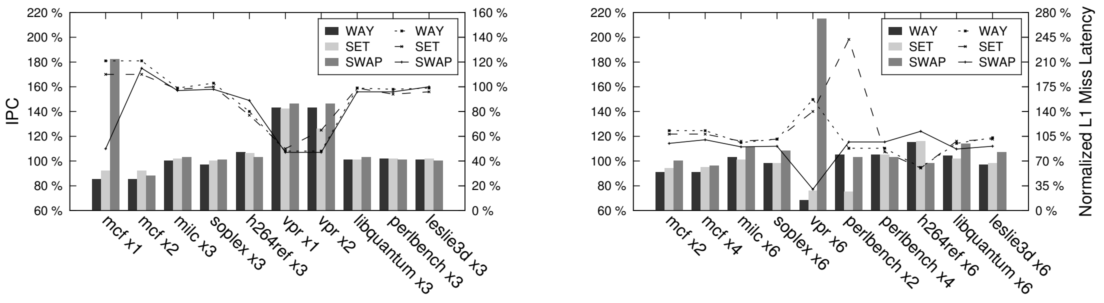
(d) 48 cores

Figure 4.5: Comparison of system throughput (weighted speedup) and L1 miss latency for Baseline, WAY, SET and SWAP. Both weighted speedup and L1 miss latency are normalized to Baseline. The bars show the weighted speedup, while the lines show the L1 miss latency normalized to baseline.

We also use L1 miss latency to show the source of performance improvement. L1 miss latency directly correlates with the number of cycles that processor pipeline is stalled by long latency memory operations, and is computed as L2 access latency + L2 miss rate × memory latency. An effective cache management technique should not only decrease the L2 miss rate of each application, but also reduce the overall memory contention, which further improves the L1 miss latency, and thus the IPC.

4.6 Evaluation

We evaluate our SWAP proposal against a Baseline configuration, where the shared L2 cache is freely contended by all 48 cores. We also compare SWAP



(a) 24 cores

(b) 48 cores

Figure 4.6: The breakdown of a sample bundle MP10 running on 24 and 48 cores in ThunderX. The bars show the IPC (normalized to IPC_{alone}), and the lines show the normalized L1 miss latency of each application.

with a best-effort cache way partitioning (WAY) and page coloring technique (SET).

Our evaluation is done in four scenarios: First, we study the case of static partitioning, where cache repartitioning is not needed. Second, we study a scenario with real-time evolving workloads, with applications coming and going, and where the dynamic cache partitioning is involved to react to the changing cache demands. We also study the overhead of the dynamic SWAP in the ThunderX platform. Third, we study how SWAP guarantees the quality of service (QoS) of latency-critical workloads, and improves the throughput of background batch applications at the same time. Note that all of the above experiments are done on a real ThunderX rack server. Finally, we compare SWAP with recently proposed probabilistic cache partitioning in the simulator.

4.6.1 Static Partitioning

In the static partitioning experiments, we run SWAP on 16, 24, 32 and 48 cores of a ThunderX 48-core processor, with the applications bundles detailed in Section 4.5.3. SWAP first reads the MPKC profile of each application, and then computes the size, shape, and placement of each core's cache partition in the shared L2, based on the algorithm discussed in Section 4.3.2. When an application finishes before the whole bundle has finished, the same application is again instantiated on the same core. It naturally inherits the cache partition of the core, and therefore no repartition is needed. The purpose of this experiment is to measure the partitioning quality of SWAP.

We first compare SWAP with Baseline (no cache partitioning involved), and the results are shown in Figure 4.5. SWAP consistently outperforms Baseline for all the bundles in all configurations, and the improvement does not decrease with more active cores, showing a good scalability in large-scale CMPs. On average, SWAP improves system throughput over Baseline by 13.9%, 14.1%, 12.5%, and 12.5% 16-, 24-, 32-, and 48-core configurations, respectively. We also find that on average, SWAP reduces the chip's overall L1 miss latency by 31.3%, 30.1%, 25.7%, and 17.6% for the core configurations we study.

We also compare SWAP with utility-based way partitioning (WAY) [79] and page coloring technique (SET) [62]. Because there are only 16 cache ways or page colors in ThunderX, it is impossible to give a disjoint cache partition to each application in either mechanism, if the number of active cores is larger than 16. We therefore design a variation of way partitioning that allows for judicious sharing of cache ways for larger configurations as follows:

We begin by reserving a small number of cache ways as the "dump area." Then, we run Qureshi and Patt's lookahead algorithm [79] to allocate the remaining cache ways. The lookahead algorithm iteratively finds the application that has the highest marginal utility on cache capacity, and assigns the cache ways to it. We run such algorithm until all the cache ways are allocated (except for the “dump” area). Then, all the remaining applications are assigned the “dump” area. In addition, as discussed in Section 4.3.1, a partition with one cache way usually introduces an excessive number of conflict misses, hurting the application’s performance. As a result, we adopt an approach similar to what Liu et al. propose [64], which coalesces the neighboring partitions if one of the them has only 1 cache way. Such coalescing rule greatly helps reduce conflict misses, and we find it significantly improves the performance of WAY.

We also study the effect of varying the size of the “dump area” from 2 to 6MB. We find that in general, the “dump area” should be as small as possible. For 16, 24, and 32 cores, reserving 2 ways performs the best. However, for 48 cores, reserving 4 ways produces the most speedup because there are more than 30 applications in such “dump area.”

Our coarse-grained page coloring scheme (SET) works similarly to WAY. However, because constraining the page colors also limits the amount of physical memory accessible by the applications, more colors should be reserved to avoid an out-of-memory error (OOM). Our study shows that reserving 2, 4, 4, and 6 colors can prevent OOM and produce the most speedup for 16-, 24-, 32-, and 48-core configurations respectively.

Figure 4.5 shows the weighted speedup of WAY, SET, and SWAP normalized to Baseline. Although WAY and SET perform well at small core counts, their partitioning quality degrades as the number of active cores increases. We look closely at a representative bundle MP10; Figure 4.6 shows the normalized IPC and L1 miss latency of each application in the bundle in 24- and 48-core configurations. In the 24-core configuration, both SET and WAY can provide a

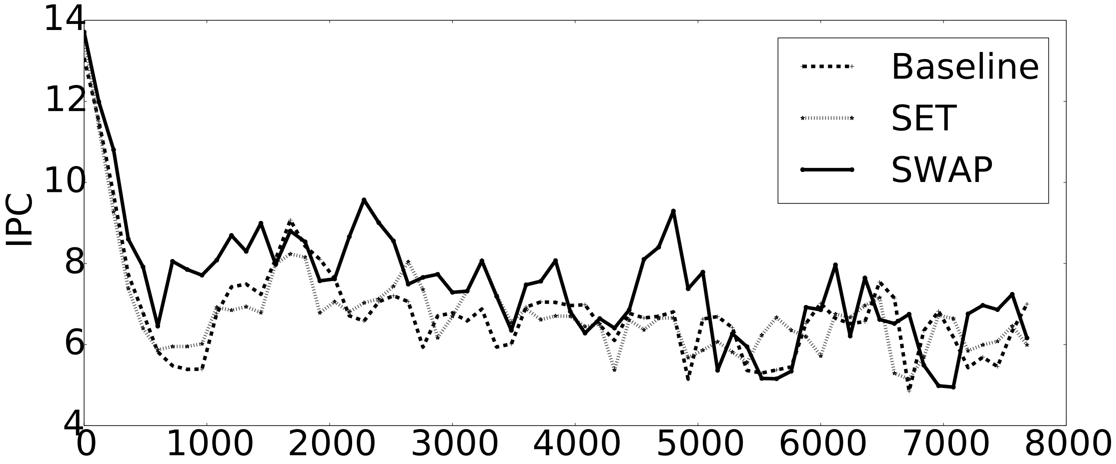


Figure 4.7: Real time throughput of Baseline, SET and SWAP of Sequence 2 in the 48-core case over time (seconds).

2MB partition for each instance of the cache sensitive application vpr. SWAP, on the other hand, can provide a tighter 1.5MB partition to each instance of vpr, and the resulting savings are given to another sensitive application, mcf (whose partition is in the dump area under SET and WAY). Although this improves the overall system throughput by only 2% in the 24-core configuration, the effect is amplified at higher core counts. Moreover, by reducing the overall L2 miss rate, SWAP greatly alleviates memory contention in the 48-core configuration, and thus helps even the non-sensitive applications. As a result, SWAP leads SET and WAY by 16%. Overall, SWAP outperforms WAY and SET by 4.36%, 5.44%, 4.3%, and 7.14%, respectively, for 16-, 24-, 36-, and 48-core configuration.

4.6.2 Dynamic SWAP with Changing Workloads

It is not necessary to invoke cache repartition in the static experiments so far, because the applications running on a core are fixed in each bundle. In this section, we evaluate SWAP in the scenario of workloads that come and go. Again, we keep the number of active cores to be fixed (16, 32, and 48 for this experiment). Instead of running a fixed bundle, we generate a long sequence of SPEC applications, and we inject the applications from the top of the sequence to the system until the number of active cores reaches the desired number. When an application finishes, we fetch the next application from the sequence, and schedule it to the currently idle core. This is similar to the scenario in clusters or data centers, where a sequence of applications is waiting in the task queue for available cores. Because the new application may show different cache characteristics, dynamic cache repartitioning is needed. For example, assume that an application with a large cache partition completes, and that a cache-insensitive one is introduced into the system. The unwanted cache capacity of the new application will be re-distributed to the other cores in the system, which triggers a system-wide repartition.

For our 16-, 32-, and 48-core configurations, we construct a sequence of 32, 64, and 96 applications, respectively, which contains a mix of applications in different categories (I, S, T). All the applications in the sequence have to finish at least once, and when all of them finish, we terminate the experiment and report the system throughput of the entire sequence. When the fetch reaches the end of the sequence, it will start over from the head of the sequence, and therefore no core will be idle.

We construct two sequences, both of which include 16 distinct SPEC benchmarks (out of 22 that we use in this paper). Table 4.3 shows the SWAP's improvement over Baseline and WAY in terms of weighted speedup. SWAP improves the weighted speedup by 8% and 17% for the two sequences in the 16-core cases, and the improvements increase to 11% and 20% for the sequences in

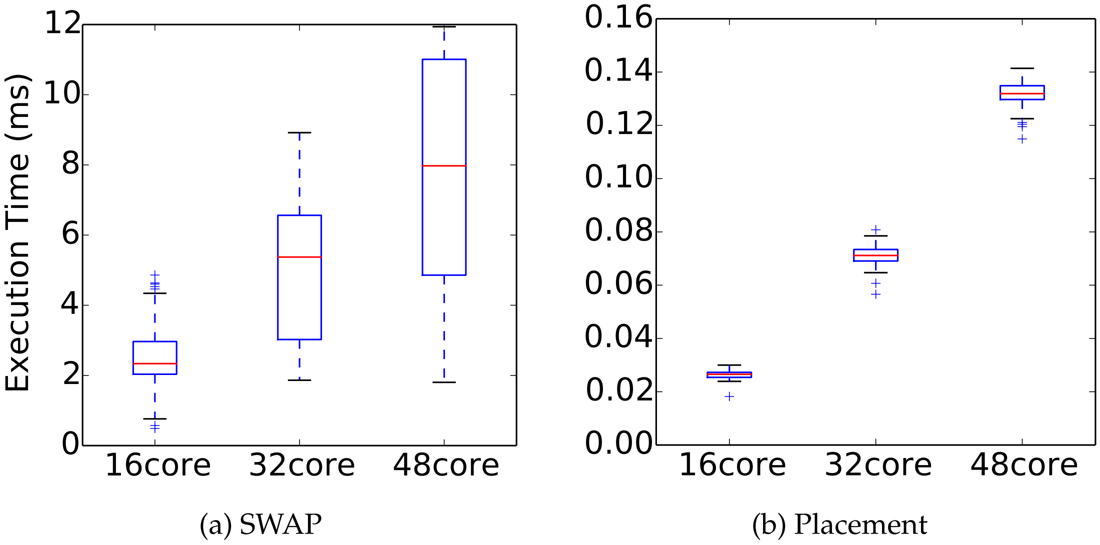
Table 4.3: Comparison of system throughput (weighted speedup normalized to Baseline) for SET, WAY, and SWAP in the dynamic experiment.

Cores	Seq	WAY	SET	SWAP	Avg. Inj interval
16	1	1.04x	1.02x	1.08x	46s
	2	1.11x	1.04x	1.17x	41s
32	1	0.97x	1.04x	1.11x	31s
	2	1.04x	1.02x	1.20x	25s
48	1	0.92x	0.99x	1.11x	34s
	2	1.00x	1.03x	1.15x	25s

the 32-core cases. Although WAY does fairly well in the 16-core sequences, its partition quality drops significantly beyond that. Besides the scalability issue of WAY and SET discussed in the static experiment in Section 4.6.1, another important reason for the poor performance is that application's injection rate is much higher (shown in Table 4.3) with higher core count. An application may be be frequently moved in and out from the "dump" area, which significantly hurts performance. Figure 4.7 shows the real-time throughput (sum of IPC) of SWAP vs. Baseline and SET. It is clear that SWAP outperforms Baseline and SET most of the time, and it runs "ahead" of Baseline and SET.

4.6.3 SWAP Overhead

This section studies the overhead of our SWAP approach. The overhead comes from two sources: (1) the execution time of the SWAP algorithm, which decides the allowable cache region of each core; and (2) the overhead of page recoloring, which involves migrating pages of an application from its old colors to the new ones.



(a) SWAP

(b) Placement

Figure 4.8: Execution time distribution of the overall SWAP, and the placement algorithm in 16-, 32-, and 48-core CMP.

Algorithm Overhead

As is described in Section 4.3.2, SWAP first runs the *lookahead* algorithm [79] to decide the optimal partition size of each core, followed by our proposed placement technique to decide the actual cache region. The complexity of the *lookahead* algorithm is $O(N^2)$, where N is the number of active cores in the chip. Our placement technique, which involves sorting all partitions by their size, has a complexity of $O(Nlog(N))$.

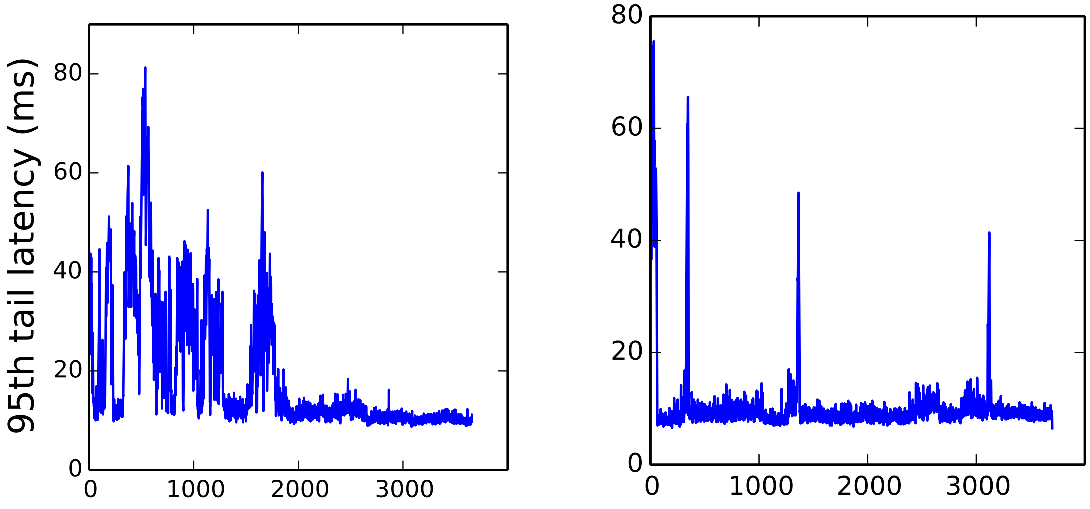
Figure 4.8 shows the distribution of the execution time of SWAP mechanism in 16-, 32-, and 48-core configurations. As is shown in Figure 4.8a, on average, the overall SWAP algorithm consumes 2ms, 6ms, and 8ms for 16-, 32-, and 48-core CMP (12ms in the worst case), which is negligible compared with the 25s repartition interval. Figure 4.8b shows the execution time of our placement technique across different CMP configuration. Although it increases linearly, it took less than 0.15ms even for the 48-core configuration.

Table 4.4: Recoloring Overhead

app	total # page recolored	overhead per repartition (ms)
bwaves	71400	213.00
leslie3d	8000	28.00
bzip2	4900	11.00
gobmk	1350	8.00
gromacs	1100	4.00

Recoloring Overhead

Our SWAP algorithm tries to avoid recoloring by taking the previous color assignment into consideration. However, recoloring is sometimes unavoidable to produce high quality cache partitions, and therefore we study the overhead of recoloring by micro-benchmarking. In the micro-benchmark, we actively recolor 50% of the pages for each SPEC application every 20s, and record the system time of that application. We consider the system time to be the aggregated overhead of page recoloring. [2] Table 4.4 shows the overhead of some sample applications. The overhead per recoloring heavily depends on the number of pages being migrated. For the applications with large memory footprint (e.g., bwaves migrates 70K pages), the overhead is about 200ms. For the applications that migrate thousands of pages, the overhead is negligible. In any case, the overhead is small compared with the 25s application repartition interval in our setup.



(a) Shared cache

(b) Exclusive cache

Figure 4.9: Real time 95th tail latency of memcached co-running with 16-app bundle MP1 over wall clock time (second).

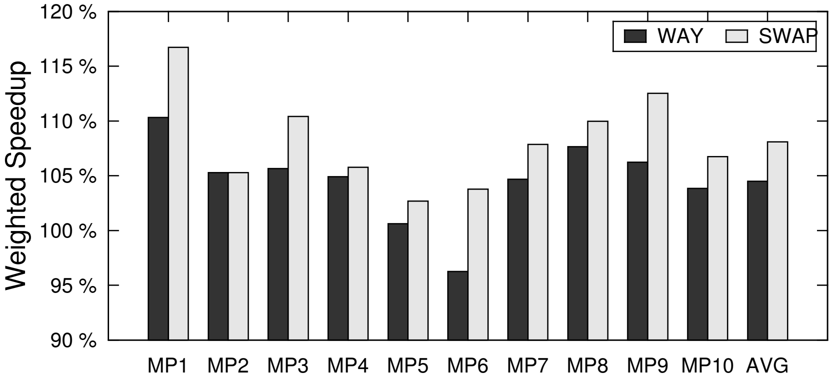


Figure 4.10: Comparison of system throughput (weighted speedup) of the background 16-app bundle for WAY and SWAP, when the QoS of memcached is satisfied. Weighted speedup is normalized with Baseline.

4.6.4 Providing QoS Guarantees

A number of studies have found that the utilization of most datacenter servers are low, and a primary reason is that the load of popular latency-critical (LC)

[2] This is a conservative estimation, because we account all the system time to be the overhead of recoloring.

workloads varies significantly due to diurnal patterns and unpredictable spikes in user accesses [5, 65, 66]. A promising way to improve server utilization is to launch batch workloads on the background to exploit the unused hardware resources [28, 66]. A key to this approach is that the QoS of the LC workloads should not be affected by the batch workloads. In this section, we propose to use SWAP to maximize the background batch workloads, with a prerequisite of guaranteeing the QoS of a popular LC workload memcached.

The memcached setup is described in Section 4.5.3. We use the multi-programmed 16-app SPEC bundles detailed in Table 4.2 as the batch workloads. In order to guarantee QoS of memcached, we allocate an independent cache partition to the memcached server to avoid interference. In addition, the capacity of such partition has to be dynamically adjusted to satisfy the QoS. We adopt a feedback-based mechanism similar to the one proposed by Lo et al. [66], which reads the tail latency every 30s. We start with two cache ways for the memcached server, and when the QoS is not met, we increase the size of its partition by one cache way. When QoS is met for a period of time (10 minutes in our setup), we decrease the partition size to explore whether the QoS can still be met. Figure 4.9 shows the tail latency of memcached over time when the server either shares cache with a sample 16-app SPEC bundle MP1, or owns its exclusive cache partition whose size is dynamically adjusted. We find that QoS (95th tail latency at 10ms) is frequently violated in the case of shared cache, but is satisfied most of the time in the exclusive cache case. A few spikes exist in Figure 4.9b because: (1) the background applications exert higher memory pressure due to phase change, which increases the penalty of L2 misses that is no longer tolerable by memcached; (2) the partition size of memcached is reduced for exploration (discussed above). In either case, our feedback-based mechanism reacts fast enough to reduce the tail latency to normal.

We use SWAP and WAY to partition the remaining cache capacity among the background SPEC applications, and compare them with a Baseline where all the SPEC applications share the cache capacity left by memcached.[3] Note that the partition of memcached server can only be adjusted by cache ways, because the overhead of recoloring its pages is too much to guarantee QoS. As a result, we exclude SET in this study. Figure 4.10 shows the system throughput of ten 16-app bundles managed by WAY and SWAP. Although memcached occupies a non-trivial amount of cache space, SWAP is still able to provide enough granularity to partition the cache space, resulting in 8.10% improvement in system throughput on average over Baseline. This almost doubles the improvement of WAY, which suffers from the limited granularity.

4.6.5 SWAP vs. Probabilistic Cache Partition

Probabilistic cache partition mechanisms [85,101] have been proposed as a scalable cache management technique for large CMPs. However, to the best of our knowledge, all those proposals require non-trivial hardware changes that are currently unavailable on real processors. Therefore, in order to compare against probabilistic cache partition mechanisms, we implement SWAP in architectural simulator SESC [81]. However, we run into a dilemma: on one hand, simulation is multi-order of magnitude slower than real machine execution, and we can only simulate 100M instructions due to the time constraints. This is equivalent to less than 1 second of actual execution, which is almost negligible compared with hours of running in our real machine experiment; on the other hand,

[3] SWAP recolors pages that belong only to SPEC applications.

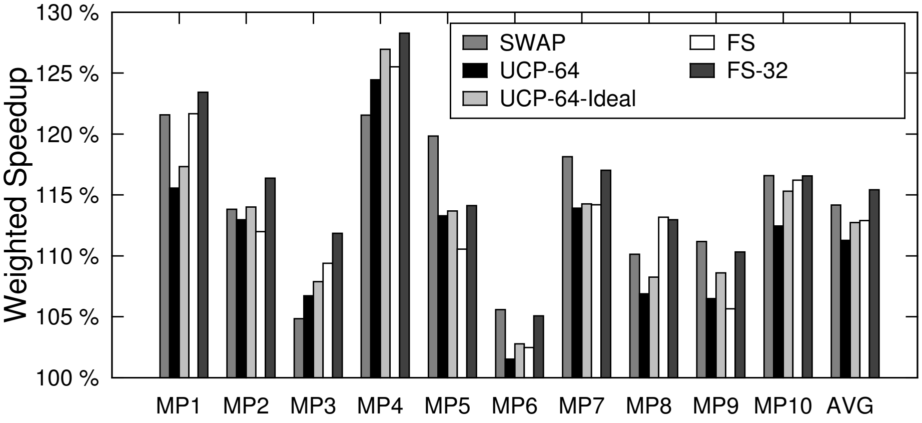


Figure 4.11: SWAP vs. Futility Scaling (FS) and utility-based way partitioning (UCP) with highly associative cache.

the overhead of SWAP is at the order of milliseconds, and we have to simulate long enough to amortize this overhead. As a result, in SESC, we ignore the two sources of SWAP overhead described in Section 4.6.3, and focus on whether SWAP is able to provide the same quality of management as those probabilistic cache partition mechanism. Other overheads, such as cache and TLB flush due to page migration, are faithfully modelled. Note that the results of our real machine studies include all the SWAP overheads.

We compare SWAP with traditional Unmanaged LRU policy; Futility Scaling [101], which is a recently proposed probabilistic cache partition mechanism that maintains fine-grained partition using a feedback control mechanism; and utility-based way partitioning (UCP) [79] with a highly-associative cache. The architectural configuration is the same as ThunderX processor described in Section 4.5.1, with 32 active cores. The shared L2 cache is 16MB with 16 ways, unless specified otherwise. The application bundles are the same as the previous real machine study.

Figure 4.11 shows the system throughput (weighted speedup), normalized to Unmanaged LRU policy. We first compare with UCP mechanism, which partitions the cache by ways. To give an independent partition to each core, we evaluate UCP with 64 cache ways, which Cacti [73] reports a 25% increase in access latency compared to 16 cache ways in ThunderX. UCP-ideal assumes the same access latency. Figure 4.11 shows that SWAP outperforms UCP in 8 out of 10 bundles, which shows that SWAP with 16 cache ways provides a finer granularity than UCP with 64 ways. The reason why UCP slightly outperforms SWAP on bundle MP3 and MP4 is that SWAP constrains the shape of each partition, thus not all partition sizes are allowed (e.g., the partition size is a multiple of its number of colors).

We then compare SWAP with Futility Scaling (FS) [101]. FS maintains a "futility" index of each cache line, and evicts cache lines with the maximum futility among the replacement candidates in the same set. Theoretically, FS is able to maintain the partition size at the granularity of lines. However, we find SWAP outperforms FS for 7 out of 10 bundles. This is because with 16 cache ways, the number of replacement candidate (16) is not large enough to include all the lines with large futility indices. As a result, we evaluate FS with 32 cache ways (FS-32), and we find that SWAP achieves comparable performance improvement. In addition to requiring fewer cache ways, SWAP does not require any extra hardware and is readily available in commercial processors, without giving up any performance improvement.

4.7 Summary

We have proposed SWAP, a fine-grained cache management technique that seamlessly combines set and way partitioning with minimum hardware support. SWAP can successfully provide hundreds of fine-grained cache partitions to achieve effective cache partitioning in the manycore era. We have prototyped SWAP on a 48-core Cavium ThunderX running Linux, and shown average speedups over no cache partitioning that are twice as large as those attained with way partitioning alone.

CHAPTER 5

RELATED WORK

5.1 Resource Allocation in CMPs

In the context of resource allocation in multicore chips, shared cache and power budget are most frequently addressed in the literature, and researchers have shown that using fine-grained management of the available resources to provide optimized utilization is highly desirable as well as practical. Sanchez and Kozyrakis, for example, show that fine-grained shared-cache cache partitioning is feasible in a large-scale CMP system [85], yielding greatly improved utilization. Similarly, multiple power-oriented studies [18, 39, 57] show that fine-grained, per-core DVFS regulation can greatly improve a CMP's energy efficiency. Intel has recently deployed a low-cost, fully-integrated voltage regulator in Haswell [43], and other researchers are making significant advances in supporting per-core DVFS [52, 88].

When it comes to the coordinated resource allocation in CMP systems, a few solutions have been proposed to optimize system throughput by relying on centralized mechanisms such as hill climbing. Choi and Yeung are among the first to perform such hill climbing technique based on trial runs [26]. Since then, a few proposals have been made to improve the performance estimation method, such as artificial neural networks [12], and analytical models [23–25]. However, because the centralized optimization mechanisms essentially explore the large global search space sequentially, which may take too long to converge to an optimal operating point. In addition, they primarily optimize for system throughput, largely ignoring fairness across applications.

5.2 Market-based Resource Allocation

In the computer systems domain, Sutherland is believed to be the first one who created a market to manage Harvard University's computing resources [94]. Since then, a number of proposals have adopted such market-based approaches to manage computer resources, whether it be bandwidth [71], memory allocation [45], or CPU scheduling [100]. For example, Harty and Cheriton integrate a market-based memory allocator in the V++ operating systems [45], so that the processes can trade off between memory capacity vs. time: whether they choose to request large amount of memory over a short period of time, or small amount of memory over a long period of time. They show this approach greatly helps the operating system to enforce its administrative policy in controlling applications' memory allocations.

In recent years, the concept of market has been introduced into distributed systems and data centers [22,40,42,80]. Chase et al. propose a static market [22] to allocate a single resource (compute service units), which is described in details in Section 2.2. Guevara et al. [42] adopt a similar approach to study the optimal configuration of heterogeneous data centers. Because the maximization process is done by the supplier centrally, it may not be efficient enough to deal with a large-scale system.

To address such scalability issue, Lai et al. introduce contentions among the selfish players, by allowing them to adjust their bids in response to the others bids to that resources [60]. As a result, the resource allocation is done in a largely distributed way. However, their study is purely empirical — they do not have any guarantees on the loss of efficiency and fairness. On the other hand, Zahedi and Lee propose an "elasticities-proportional" (EP) mechanism [111], which has game-theoretic guarantees such as Pareto Efficiency, Envy-freeness, etc. However, the limitation of their study is that, they require each player's utility function to be curve-fitted to a Cobb-Douglas function, in order to obtain its elasticities of the resources. In addition, although they prove EP is Pareto Efficient, they do not quantify the efficiency loss compared with global optimality.

5.3 Theoretical Studies of Market Equilibrium

Papadimitriou introduces the concept of *Price of Anarchy* (PoA) [76], which is the lower bound of the efficiency of market equilibrium, compared with optimal allocation. Zhang [112] shows that in a balanced game, where each player is assigned a budget proportional to its maximum utility, i.e., the utility when it owns all the resources, the market equilibrium has a Price of Anarchy $PoA = \Theta(\frac{1}{\sqrt{N}})$. On the other hand, Johari and Tsitsiklis [53] show that in a market without budget constraint,[1] the market equilibrium has a Price of Anarchy $PoA = 3/4$.

5.4 Cache Partitioning in CMPs

Intelligently partitioning a CMP's last-level cache can be an effective way to optimize execution of co-running application bundles. Existing cache partitioning mechanisms generally fall into one of three categories: *(a)* hardware way par-

[1] the players are not constrained in their bids, and they try to to maximize their utility minus the resource costs

titioning; *(b)* software page coloring; or *(c)* probabilistic approaches that tweak cache insertion and eviction.

5.4.1 Way Partitioning

In the single core environment, Albonesi is believed to be the first to propose turning off unneeded cache ways to reduce cache energy [1]. Yang et al. improve such technique by dynamically adjust the active cache capacity to accommodate the changing working set of an application [107,108]. Powell et al. adopt an orthogonal approach, which applies way prediction and direct-mapping to pinpointing the matching ways instead of probing all the ways [78].

In the multicore era, Suh et al. [93] propose to distribute L2 cache ways to minimize the overall miss rate. Qureshi and Patt [79] improves their technique by predicting the marginal utility of additional cache ways.

5.4.2 Page Coloring

Kessler and Hill were among the first to use page coloring to improve the utilization of hardware cache, by distributing the physical pages evenly to different cache sets [56]. Such technique was later adopted by commercial OS such as FreeBSD [33]. A number of follow-up works further improve the cache utilization, and reduce the overhead of page recoloring [86]. For multi-core chips, Lin et al. [62] use page coloring to partition the shared cache among the cores in a dual-core chip to improve system efficiency. There are also proposals to use page coloring to partition memory banks [63,110], or even to manage cache and memory contention cooperatively [64].

5.4.3 Probabilistic Cache Partitioning

Sanchez and Kozyrakis propose Vantage [85], a replacement-based partitioning mechanism that can probabilistically guarantee the size of partitions across applications. Wang and Chen [101] adopt a similar approach to maintaining a partition's cache size, by controlling the eviction priority of cache lines belonging to different cores. Although their simulation-based evaluations show promise, both proposals require a non-trivial amount of additional hardware support. For that same reason, their results cannot be validated by real implementations using commercial processors, where significant discrepancies could arise [62]. Finally, it is unclear whether these probabilistic approaches would be good enough for environments where strict isolation is highly desirable (e.g., to reduce exposure to timing channel attacks).

5.4.4 Cache Partitioning for Tile-based CMPs

In tiled-based architectures, each cache tile constitutes the primary container for the local core, and thus a natural partition exists. Lee et al. propose CloudCache [61], which explores allocating partitions potentially larger than tiles by "borrowing" cache ways from remote tiles. Beckmann et al. improve CloudCache by favoring neighboring tiles, so that the cache access latency via on-chip networks is minimized [7,9].

CHAPTER 6

CONCLUSION

This thesis explores novel approaches to dynamic and scalable resource allocation in multicore systems. Most existing proposals for shared resource management in chip multiprocessors (CMP) are centralized, which has been proven to work reasonably well at small scale (4-8 cores). However, the number of cores on a CMP keeps piling up. Indeed, many microprocessor manufacturers nowadays have large-scale CMPs in their product line: From Intel's Xeon line (10-18 cores), to ARM server processors (Cavium's 48-core Thunder, Qualcomm's 64-core Hydra), and Intel's Xeon Phi, which supports up to 288 threads contending for shared resources simultaneously [4, 13, 20, 36, 96]. At such scale, the global resource management technique is no longer practical, because the super-linear increase in execution time makes global management infeasible to be applied to large-scale CMPs.

Market mechanisms, which are widely used in real life, have been proved to be a great success in allocating shared social resource at scale. We think a market-based approach is also the right solution for scalable resource allocation in large-scale CMPs. Our XChange proposal is believed to be the first to allocate resources in a decentralized manner in large-scale CMP systems, by formulating a purely dynamic, mostly distributed market: Sellers and buyers in the market act individually to pursue their own benefit, and a well-regulated market often produces an outcome where everyone is reasonably happy. We prove the effectiveness and efficiency of the market-based approach by (1) adopting an iterative price discovery process to dynamically reconcile the resource supply and demand; (2) shifting most of the effort to each individual core, so that the technique be scalable to large-scale CMPs; (3) employing hardware monitors to construct application's performance-resource relationship at runtime, so that cores can accurately bid to maximize their utility; (4) providing a "knob," which we call wealth redistribution, to trade off system throughput and fairness effectively.

Our solution can be adopted with very low overhead. In some Linux-based SMP systems, all cores are simultaneously interrupted by an APIC timer every 1 ms to conduct a kernel statistics update routine. We propose to piggyback on this interrupt to incorporate our price discovery mechanism. With this 1ms as the re-allocation interval, our experiments show that the market-based approach can react fast enough to the dynamic behavior of the applications, and significantly improve the system performance and resource utilization. We also show that the market spends less than 0.5% of the time to reach the outcome for CMP systems with fewer than 128 cores. The hardware requirement for workload characterization is also very lightweight.

In addition, we note that our proposed market-based approach is a general framework, which can be applied to multiple shared hardware resources and a variety of computational elements. We show success for general-purpose cores on two arguably very important shared hardware resources, power budget and cache capacity. Moreover, the "knob" we provide to trade off system efficiency and fairness makes market-based mechanism applicable to a wide range of systems with different performance targets. We hope that our work inspires architects to revisit the resource allocation problem in large-scale CMP systems, and apply our approach to systems with more shared resources, as well as other types of cores.

Despite the XChange's scalability, superior system efficiency and fairness against existing proposals, it is purely empirical, however, and thus it does not provide any guarantees on the loss of efficiency and fairness. It is well-known, for example, that market mechanisms in equilibrium can sometimes be highly inefficient—this is known as *Tragedy of Commons* [44]. In such cases, the overall system efficiency is very low ($1/\sqrt{N}$ of the maximum feasible utility, where N is the number of market players). Even worse, because computing the optimal resource allocation (OPT) often involves global optimization which is prohibitively expensive, the system will not even realize it's working in an inferior operating point. In addition, prior market-based approaches are not able to provide a "knob" to control the efficiency vs. fairness trade-off in a systematic manner.

To address those issues, my thesis makes two important contributions to the existing market-based resource allocation approaches: (1) by measuring the Market Utility Range (MUR) and Market Budget Range (MBR) of the current market, we can not only provide a theoretic guarantees of the system efficiency and fairness, but also make a good estimate of these two. If MUR (MBR) indicates a low throughput (fairness), a warning may be sent out to the system administrator, so that appropriate regulation can be applied to the market; (2) backed by the theoretical studies, we provide a practical "knob", budget re-assignment, which the system administrator can use to systematically trade off system efficiency and fairness. Our experimental results using detailed execution-driven simulations shows that our budget re-assignment technique is intuitive, effective, and efficient in practice.

Our budget re-assignment technique is an essential complement for the existing market-based resource allocation approaches, and is general enough to be applied to multiple shared hardware resources and a variety of computational elements. We show success for general-purpose cores on two arguably very important shared hardware resources, power budget and cache capacity. Moreover, the "knob" we provide to trade off system efficiency and fairness makes market-based mechanism applicable to a wide range of systems with different performance targets. With enhanced robustness, monitoring support, and adjustability between efficiency and fairness, we expect our work inspires architects to revisit the resource allocation problem in large-scale CMP systems, and make market-based approach more easily to be adapt in the real system with confidence.

Our market-based approach relies on hardware and software support for shared resource partition. Although fine-grained power partition has already been supported by commercial chips (Intel's RAPL technique [51]), fine-grained cache partition remains to be a hard problem, especially for large-scale CMPs. Two popular cache partition approaches are *(a)* hardware support for way partitioning, or *(b)* operating system support for set partitioning through page coloring. However, neither mechanism is able to scale beyond a handful of colors.

We propose SWAP [102], a fine-grained cache partitioning mechanism that can be readily implemented in existing CMP systems. By cooperatively combining the cache way (hardware) and set (OS) partitioning, SWAP is able to divide the shared cache into literally hundreds of regions, therefore providing sufficiently fine granularity for the upcoming manycore processor generation.

We implement SWAP as a user-space management thread on Cavium's ThunderX, a server-grade 48-core processor with ARM-v8 ISA [97]. To enable SWAP, we introduce small changes to the Linux page allocator, and leverage ThunderX's native architectural support for way partitioning. Our results show that SWAP is able significantly improve system throughput (weighted speedup), by twice as much speedup as what we can obtain by using only way partitioning mechanism. To our knowledge, SWAP is the first proposal of a fine-grained cache partitioning technique that requires no more hardware than what's already present in commercial server-grade CMPs.

CHAPTER 7

FUTURE WORK

7.1 Accurate Online Utility Modeling

Accurate utility—resource modeling is critical to the resource allocation problem in CMPs. Our first attempt in XChange was to adopt a simple linear utility model, as is discussed in Section 2.4. We found that such simple model cannot maximize the potential of the market-based mechanism, and it sometimes may not even outperform the baseline fair-share resource allocation. Similarly, our experiments show that an important reason why REF [111] cannot achieve the same performance and fairness benefit as XChange does is because it curve-fits an application's utility into Cobb-Douglas function. Although recent proposals have improved performance estimation method from trial runs [26], to artificial neural networks [12], and to analytical models [23–25], we think there is still potential for improvement. For example, XChange's utility model is based on two well-known proposals that models cache and power utility. As we show in Section 2.8, although the model's relative error is low most of the time, there are some cases where the model becomes inaccurate. Therefore, we believe more research effort is necessary to further polish the existing utility modeling technique.

In addition, we find that the support for accurate utility modeling is not enough in real commercial processors, despite the existence of some industrial efforts, such as the Intel's Cache Monitoring Technique (CMT) [46]. The shadow tag approach [79], for example, is not readily available in real processors, although it is widely used in the research community to predict an application's miss ratio curve under different cache capacities. Some software techniques have been proposed [30, 95], but they incur high execution overhead, and also rely on hardware counters that is not available on every processor. We believe more industrial effort to incorporate utility modeling hardware into real processors is necessary.

7.2 More CMP Resources

In this thesis, we study two of the most important resources in CMPs: power and cache. There are other shared resources that we haven't explored, such as memory bandwidth, on-chip interconnect network, etc. In addition, in the simultaneous multithreading (SMT) processors, multiple threads run on a single core, and share resources such as instruction fetch unit, reorder buffer, execution units, a core's private L1 cache, etc.

The market-based approach proposed by this thesis is a general framework, which can be applied to all the resources mentioned above, as long as: (1) the resource's utility function can be accurately modeled; (2) the resource can be effectively partitioned among the cores. Besides the importance of accurate utility modeling we discussed in Section 7.1, effective resource partitioning is also essential, especially for large-scale CMPs. This thesis has shown that SWAP, a fine-grained cache partitioning technique, can significantly improve the system throughput in a real 48-core CMP system. However, we find there is a lack of support in the literature to manage some other important resources mentioned above. For example, the ability of partitioning memory bandwidth in a real processor is quite limited. Workarounds have been proposed, where the number of active applications is decreased in the system to reduce memory traffic [66], so that the QoS of the important latency-critical workload is satisfied. This is not ideal, however, because it reduces the utilization of the core resources. A recent proposal by Zhou and Wentzlaff [115] adopts a statistical approach in memory bandwidth partitioning, which is promising. We believe further research in regulating memory bandwidth, and other shared CMP resources, is essential to exploit the full potential for market-based mechanism this thesis has proposed.

7.3 Heterogeneous Architecture and SOCs

This thesis studies the resource allocation problem in homogeneous large-scale CMPs. We observe that there is a recent trend to make CMP heterogeneous, especially in the domain of mobile processors. For example, ARM has proposed big.LITTLE [48], a heterogeneous processing architecture where the little cores are designed for maximum power efficiency, and the big cores are tailored for maximum compute performance. Integrating our market-based resource allocation into such heterogeneous architecture is an interesting topic for future research.

We also observe that many processors today, including mobile, desktop, and server processors, are SOCs (system on-chip). Besides a composition of general-purpose cores, SOCs often include multiple accelerators, which are designed for specific tasks, such as video decoding, encryption and decryption, etc. They can not only offload the cores in the SOC, but also achieve better performance and energy efficiency. However, these accelerators may be heavily contended by the tasks running on different cores, and therefore can be treated as resources in the SOC. Applying our market-based mechanism into the problem of allocating shared accelerators to the competing cores is another important extension of this thesis.

7.4 Theoretic Studies

We find that most of the works in the architectural community are based on the intuitions of human experts. Although it has shown to work well in lots of cases, we find that studying the theoretic properties may often improve the intuitive mechanisms. This thesis shows a good example. XChange, a heuristic market-based approach, is proved to be scalable and effective for the resource allocation problem in large-scale CMPs. In addition, it provides an intuitive "knob", wealth redistribution, to trade off system efficiency and fairness. ReBudget, on the other hand, studies the theoretic lower bound in system efficiency and fairness of the market-based approach. Based on these theoretic properties, ReBudget is able to trade off the efficiency and fairness in a systematic way, and achieves a better throughput or fairness compared with XChange.

We do not overlook the importance of human intuitions. In fact, ReBudget was developed as a combination of human intuitions, and rigorous mathematical proofs. We encourage researchers in our field to spend more times on theoretic studies, especially when today's computer architecture is often times too complicated for human intuitions alone.

APPENDIX A

PROOF FOR REBUDGET

A.1 Proof of Theorem 1

We introduce λ value and budget utilization metric BU into the proof from Johari and Tsitsiklis [53], so that such proof can be generalized to a budget constraint problem. In a market equilibrium allocation $\mathbf{r}^n$ where player i bids $\mathbf{x}_i^n$, there exists $\lambda_i > 0$ for this player, such that for any resources j (recall $y_{ij} = \sum_{i' \neq i} x_{i'j}$):

$$\begin{aligned}
\frac{\partial U_i(\mathbf{r}_i^n)}{\partial x_{ij}} &= \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_{ij}} \\
&= \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}} \left(\frac{C_j}{x_{ij}^n + y_{ij}^n} - \frac{x_{ij}^n C_j}{(x_{ij}^n + y_{ij}^n)^2}\right) \\
&= \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}} \frac{1}{p_j^n} \left(1 - \frac{r_{ij}^n}{C_j}\right) \begin{cases} = \lambda_i & \text{if} \quad x_{ij} > 0 \\ < \lambda_i & \text{if} \quad x_{ij} = 0 \end{cases}
\end{aligned} \tag{A.1}$$

Let $V_i(\mathbf{r}_i) = \sum_j \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}}(r_{ij} - r_{ij}^n) + U_i(\mathbf{r}_i^n)$. We find the equilibrium allocation $\mathbf{r}^n$ for U_i is also the equilibrium allocation for V_i:

$$\frac{\partial V_i(\mathbf{r}_i^n)}{\partial x_{ij}} = \frac{\partial V_i(\mathbf{r}_i^n)}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_{ij}} = \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_{ij}}.$$

Define Nash(U) to be the social welfare (sum of players' utility) in market equilibrium, where players have utility function U_i. Because the market equilibrium allocation for U_i is also the equilibrium allocation for V_i, Nash(U) = Nash(V). Also define OPT(U) to be the maximum social welfare under any feasible resource allocation, where players have utility function U_i. Due to the concavity of the utility function U_i, we have $V_i \geq U_i$ for any resource allocation $\mathbf{r}_i$. Therefore, we have $\mathrm{OPT}(V) \geq \mathrm{OPT}(U)$. As a result:

$$\mathrm{Nash}(U)/\mathrm{OPT}(U) \geq \mathrm{Nash}(V)/\mathrm{OPT}(V)$$

Define $\alpha_{ij} = \frac{\partial U_i(\mathbf{r}_i^n)}{\partial r_{ij}}$, $\beta_i = U_i(\mathbf{r}_i^n) - \sum_j \alpha_{ij} r_{ij}^n$, then we have $V_i(\mathbf{r}_i) = \sum_j \alpha_{ij} r_{ij} + \beta_i$. Let $W_i(\mathbf{r}_i) = \sum_j \alpha_{ij} r_{ij}$, $B = \sum_i \beta_i$. Then we have:

$$\frac{\mathrm{Nash}(U)}{\mathrm{OPT}(U)} \geq \frac{\mathrm{Nash}(V)}{\mathrm{OPT}(V)} \geq \frac{\mathrm{Nash}(V) - B}{\mathrm{OPT}(V) - B} = \frac{\mathrm{Nash}(W)}{\mathrm{OPT}(W)} \tag{A.2}$$

Next, we show the *Price of Anarchy* assuming players have utility W_i. The social welfare $W = \sum_i W_i(\mathbf{r}_i) = \sum_i \sum_j \alpha_{ij} r_{ij} = \sum_j \sum_i \alpha_{ij} r_{ij}$. It is easy to find that the optimal social welfare is achieved if for resource j, all C_j is given to the player i that has maximum α_{ij}:

$$\mathrm{OPT}(W) = \sum_j C_j \max_i \{\alpha_{ij}\}$$

For market equilibrium allocation r^n, we have $\mathrm{Nash}(W) = \sum_j \sum_i \alpha_{ij} r_{ij}^n$. Define $\mathrm{Nash}_j = \sum_i \alpha_{ij} r_{ij}^n$. From Equation A.1, each player has $\alpha_{ij} \frac{1}{p_j^n}(1 - \frac{r_{ij}^n}{C_j}) = \lambda_i$ for any resource j if he submits a non-zero bid to it. Therefore, $\alpha_{ij} \geq p_j^n \lambda_i$. Without loss of generality, we define $\alpha_{1j} = \max_i\{\alpha_{ij}\}$. Then we have:

$$\begin{aligned}\mathrm{Nash}_j \ &= \sum_i \alpha_{ij} r_{ij}^n = \alpha_{1j} r_{1j}^n + \sum_{i \neq 1} \alpha_{ij} r_{ij}^n \\ &\geq \alpha_{1j} r_{1j}^n + \sum_{i \neq 1} p_j^n \lambda_i r_{ij}^n \\ &\geq \alpha_{1j} r_{1j}^n + p_j^n \min_i \{\lambda_i\}(C_j - r_{1j}^n)\end{aligned}$$

From Equation A.1, we have $p_j^n = \frac{\alpha_{ij}}{\lambda_i}(1 - \frac{r_{ij}}{C_j})$, then:

$$\begin{aligned}\mathrm{Nash}_j \ &\geq \alpha_{1j} r_{1j}^n + \alpha_{1j} \frac{\min_i\{\lambda_i\}}{\lambda_1}(C_j - r_{1j}^n)(1 - \frac{r_{1j}^n}{C_j}) \\ &\geq \alpha_{1j} r_{1j}^n + \alpha_{1j} \frac{\min_i\{\lambda_i\}}{\max_i\{\lambda_i\}}(C_j - r_{1j}^n)(1 - \frac{r_{1j}^n}{C_j})\end{aligned}$$

Define BU $= \frac{\min_i\{\lambda_i\}}{\max_i\{\lambda_i\}}$, then:

$$\begin{aligned}\text{Nash}_j \ & \geq \alpha_{1j}(r^n_{1j} + \text{BU}(C_j - r^n_{1j})(1 - \frac{r^n_{1j}}{C_j}) \\ & = \alpha_{1j}[\frac{\text{BU}}{C_j}(r^n_{1j} - (1 - \frac{1}{2\text{BU}}))^2 + C_j(1 - \frac{1}{4\text{BU}})]\end{aligned}$$

Therefore, if BU $\geq \frac{1}{2}$, $\text{Nash}_j \geq \alpha_{1j}C_j(1 - \frac{1}{4\text{BU}})$, and:

$$\begin{aligned}\text{Nash}(W) \ & = \sum_j \text{Nash}_j \geq (1 - \frac{1}{4\text{BU}}) \sum_j \max_i\{\alpha_{ij}\}C_j \\ & = (1 - \frac{1}{4\text{BU}})\text{OPT}(W) \geq \frac{1}{2}\text{OPT}(W)\end{aligned}$$

If BU $< \frac{1}{2}$, $\text{Nash}_j > \alpha_{1j}C_j \cdot \text{BU}$, then:

$$\text{Nash}(W) > \text{BU} \cdot \text{OPT}(W)$$

Combined with Equation A.2, we have:

- If BU $\geq \frac{1}{2}$:
$$\frac{\text{Nash}(U)}{\text{OPT}(U)} \geq \frac{\text{Nash}(W)}{\text{OPT}(W)} \geq (1 - \frac{1}{4\text{BU}}) \tag{A.3}$$
- If BU $> \frac{1}{2}$:
$$\frac{\text{Nash}(U)}{\text{OPT}(U)} \geq \frac{\text{Nash}(W)}{\text{OPT}(W)} \geq \text{BU} \tag{A.4}$$

□

A.2 Proof of Theorem 2

We introduce *budget variation BV* into Zhang's proof of envy-freeness with equal budget, so that our results apply to a market with an arbitrary budget assignment for players.

The idea for proving *envy-freeness* is to prove that at market equilibrium, for any bid vector $\mathbf{z}$ with $0 \leq z_j \leq y_{ij}^n$, and $\sum_j z_j = \max_i\{X_i\}$, $U_i(\mathbf{r}_i^n) \geq c \cdot U_i(\frac{z_j}{x_{ij}^n + y_{ij}^n})$ always stands.

To prove this, we construct the same $V_i(\mathbf{r}_i)$ and $W_i(\mathbf{r}_i)$ utility function for player i as we did in Section A.1. We also define $p_{ij} = \frac{\alpha_{ij} C_j}{x_{ij} + y_{ij}}$, and therefore, $W_i(\mathbf{r}_i(\mathbf{x}_i)) = \sum_j p_{ij} x_{ij}$. We also define the budget variation for player i:

$$\mathrm{BV}_i = \frac{X_i}{\max_j\{X_j\}} \tag{A.5}$$

By adopting the same technique as Zhang does [112], we construct a matrix $\{b_{jk}\}$ for $0 \leq j, k \leq M$, where M is the number of resources. Such matrix satisfies the following three conditions:

$$b_{jj} = \min_j\{x_{ij}, z_j \cdot \mathrm{BV}_i\}, \quad \sum_k b_{jk} = x_{ij}, \quad \sum_j b_{jk} = z_k \cdot \mathrm{BV}_i$$

Because $\sum_j \sum_k b_{jk} = X_i$, and $\sum_k \sum_j b_{jk} = \mathrm{BV}_i \cdot \sum_k z_k = X_i$, such matrix $\{b_{jk}\}$ exists. Therefore, we have:

$$W_i(\mathbf{x}_i) = \sum_j p_{ij} x_{ij} = \sum_j p_{ij} \sum_k b_{jk} = \sum_j (p_{ij} b_{jj} + p_{ij} \sum_{k \neq j} b_{jk})$$

Based on Equation A.1, we define the marginal utility of bid x_{ij} to be:

$$\lambda_{ij} = \frac{\partial U_i}{\partial x_{ij}} = \alpha_{ij} \frac{y_{ij} C_j}{(x_{ij} + y_{ij})^2} \begin{cases} = \lambda_i & \text{if} \quad x_{ij} > 0 \\ < \lambda_i & \text{if} \quad x_{ij} = 0 \end{cases} \tag{A.6}$$

Therefore, $p_{ij}/\lambda_{ij} = \frac{x_{ij} + y_{ij}}{y_{ij}} > 1$, $p_{ij} \geq \lambda_{ij}$, and we have:

$$\begin{aligned} W_i(\mathbf{x}_i) &= \sum_j (p_{ij} b_{jj} + p_{ij} \sum_{k \neq j} b_{jk}) \\ &\geq \sum_j (p_{ij} b_{jj} + \lambda_{ij} \sum_{k \neq j} b_{jk}) \end{aligned}$$

If $b_{jk} > 0$, this implies $x_{ij} > 0$. Then according to Equation A.6, $\lambda_{ij} = \lambda_i \geq \lambda_{ik}, \forall k$. Therefore:

$$\begin{aligned} W_i(\mathbf{x}_i) &\geq \sum_j (p_{ij} b_{jj} + \sum_{k \neq j} \lambda_{ik} b_{jk}) \\ &= \sum_k (p_{ik} b_{kk} + \lambda_{ik} \sum_{j \neq k} b_{jk}) \end{aligned}$$

Let $W_{ik} = p_{ik} b_{kk} + \lambda_{ik} \sum_{j \neq k} b_{jk}$, then:

- If $z_k \cdot \mathrm{BV}_i \leq x_{ik}$, then based on definition, $b_{kk} = z_k \cdot \mathrm{BV}_i$, and $\sum_{j \neq k} b_{jk} = z_k - z_k = 0$. Therefore,

$$W_{ik} = p_{ik} b_{kk} + \lambda_{ik} \sum_{j \neq k} b_{jk} = p_{ik} z_k \cdot \mathrm{BV}_i \tag{A.7}$$

- If $z_k \cdot \mathrm{BV}_i \geq x_{ik}$, then $b_{kk} = x_{ik}$, and $\sum_{j \neq k} b_j k = z_k \cdot \mathrm{BV}_i - x_{ik}$. Therefore, we have:

$$\begin{aligned} W_{ik} &= p_{ik} x_{ik} + \lambda_{ik} \cdot (z_k \cdot \mathrm{BV}_i - x_{ik}) \\ &= p_{ik} x_{ik} + p_{ik} \frac{y_{ik}}{x_{ik} + y_{ik}} (z_k \cdot \mathrm{BV}_i - x_{ik}) \\ &= p_{ik} z_{ik} \cdot (\frac{y_{ik} \cdot \mathrm{BV}_i}{x_{ik} + y_{ik}} + \frac{x_{ik}^2}{(x_{ik} + y_{ik}) z_k}) \\ &\geq p_{ik} z_{ik} \cdot (\frac{y_{ik} \cdot \mathrm{BV}_i}{x_{ik} + y_{ik}} + \frac{x_{ik}^2}{(x_{ik} + y_{ik}) y_{ik}}), \quad z_k \leq y_{ik} \\ &= p_{ik} z_{ik} \cdot \frac{y_{ik}^2 \cdot \mathrm{BV}_i + x_{ik}^2}{(x_{ik} + y_{ik}) y_{ik}} \end{aligned}$$

By fix y_{ik}, for any x_{ik}, we can get:

$$W_{ik} \geq p_{ik} z_{ik} \cdot (2\sqrt{1 + \mathrm{BV}_i} - 2) \tag{A.8}$$

The equality holds only when $x_{ik} = y_{ik} \cdot (\sqrt{1 + \mathrm{BV}_i} - 1)$.

Combining Equation A.7 and Equation A.8, we have:

$$\begin{aligned} W_i(\mathbf{x}_i) &= \sum_k W_{ik} \\ &\geq \min\{\mathrm{BV}_i, 2\sqrt{1 + \mathrm{BV}_i} - 2\} \cdot \sum_k p_{ik} z_{ik} \\ &= (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot W_i(\mathbf{z}) \end{aligned}$$

Recall in Section A.1, $V_i(\mathbf{x}_i) = W_i(\mathbf{x}_i) + \beta_i$. Also, because $\mathrm{BV}_i = \frac{X_i}{\max_j\{X_j\}} \leq 1$, we have $2\sqrt{1 + \mathrm{BV}_i} - 2 \leq 1$. and therefore:

$$\begin{aligned} V_i(\mathbf{x}_i) \; &= W_i(\mathbf{x}_i) + \beta_i \geq (2\sqrt{1 + \mathrm{BV}_i} - 2)(W_i(\mathbf{x}_i) + \beta_i) \\ &= (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot V_i(\mathbf{z}) \end{aligned}$$

Again, recall in Section A.1, for any bid vector $\mathbf{x}_i$, $V_i(\mathbf{x}_i) \geq U_i(\mathbf{x}_i)$, and the market equilibrium bid vector $\mathbf{x}_i^n$ are the same for player with U_i or V_i utility function, and therefore, we have:

$$U_i(\mathbf{x}_i^n) = V_i(\mathbf{x}_i^n) \geq (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot V_i(\mathbf{z}) \geq (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot U_i(\mathbf{z})$$

Finally, for the all the players, define *budget variation* to be:

$$\mathrm{BV} = \frac{\min\{X_i\}}{\max_i\{X_i\}} \leq \mathrm{BV}_i \quad \text{for } \forall i$$

And therefore, for any player i,

$$U_i(\mathbf{x}_i^n) \geq (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot U_i(\mathbf{z}) \geq (2\sqrt{1 + \mathrm{BV}} - 2) \cdot U_i(\mathbf{z}) \quad \square$$

A.3 Proof of Theorem 3

A market with proportional budget is defined as: (1) for each player i, if he is given all the resources $\mathbf{C}$, his utility $U_i(\mathbf{C}) = 1$ (or a player-independent constant); (2) his budget is proportional to $U_i(\mathbf{C}) - U_i(\mathbf{0})$. Without loss of generality, we let budget $X_i = 1 - U_i(\mathbf{0})$. Therefore, the maximum $X_i = 1$, and $\mathrm{BV}_i \geq X_i$. Also we define a utility function for player i: $\bar{U}_i(\mathbf{r}_i(\mathbf{x}_i)) = U_i(\mathbf{r}_i(\mathbf{x}_i)) - U_i(\mathbf{0})$. If players are using $\bar{U}_i$ to participate in the market, then based on Theorem 2, we have:

$$\bar{U}_i(\mathbf{x}_i^n) \geq (2\sqrt{1 + \mathrm{BV}} - 2) \cdot \bar{U}_i(\mathbf{z})$$

It is easy to find that the market equilibrium of $\bar{U}_i$ is also the equilibrium of U_i. Therefore, we have:

$$\begin{aligned}
U_i(\mathbf{x}_i^n) \quad &= U_i(\mathbf{0}) + \bar{U}_i(\mathbf{x}_i^n) = 1 - X_i + \bar{U}_i(\mathbf{x}_i^n) \\
&\geq 1 - X_i + (2\sqrt{1 + \mathrm{BV}_i} - 2) \cdot \bar{U}_i(\mathbf{z}) \\
&\geq 1 - X_i + (2\sqrt{1 + X_i} - 2) \cdot \bar{U}_i(\mathbf{z}) \\
&= U_i(\mathbf{z}) \cdot \frac{1 - X_i + (2\sqrt{1 + X_i} - 2) \cdot \bar{U}_i(\mathbf{z})}{U_i(\mathbf{z})} \\
&= U_i(\mathbf{z}) \cdot \frac{1 - X_i + (2\sqrt{1 + X_i} - 2) \cdot \bar{U}_i(\mathbf{z})}{1 - X_i + \bar{U}_i(\mathbf{z})} \\
&= U_i(\mathbf{z}) \cdot (2\sqrt{1 + X_i} - 2 + \frac{(3 - 2\sqrt{1 + X_i})(1 - X_i)}{1 - X_i + \bar{U}_i(\mathbf{z})}
\end{aligned}$$

Because $\bar{U}_i(\mathbf{z}) \leq X_i$, we have:

$$\begin{aligned}
U_i(\mathbf{x}_i^n) \quad &\geq U_i(\mathbf{z}) \cdot [2\sqrt{1 + X_i} - 2 + (3 - 2\sqrt{1 + X_i})(1 - X_i)] \\
&\geq U_i(\mathbf{z}) \cdot 0.718
\end{aligned}$$

The equality holds only when $X_i = \frac{\sqrt{21}-1}{6} = 0.597$. □

BIBLIOGRAPHY

[1] David H Albonesi. Selective cache ways: On-demand cache resource allocation. In *Intl. Symp. on Microarchitecture (MICRO)*, 1999.

[2] AnandTech. ARM Challenging Intel in the Server Market: An Overview. http://www.anandtech.com/show/8776/arm-challinging-intel-in-the-server-market-an-overview/4, 2014.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, May 2015.

[4] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor. In *HotChips Symp. on High Performance Chips*, 2015.

[5] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[6] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Intl. Conf. on Computing Frontiers (FC)*, 2006.

[7] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[8] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[9] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[10] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[11] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, 2009.

[12] Ramazan Bitirgen, Engin Ipek, and José F. Martínez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Intl. Symp. on Microarchitecture (MICRO)*, 2008.

[13] Bill Bowhill, Blaine Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, et al. The Xeon® processor E5-2600 v3: A 22nm 18-core product family. In *International Solid-State Circuits Conference (ISSCC)*, 2015.

[14] Steven J Brams and Alan D Taylor. *Fair Division: From cake-cutting to dispute resolution*. Cambridge University Press, 1996.

[15] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *Intl. Conf. on Parallel Processing (ICPP)*, 2015.

[16] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Intl. Symp. on Computer Architecture (ISCA)*, 2000.

[17] B. Calder, T. Sherwood, E. Perelman, and G. Hamerley. Simpoint. http://www.cs.ucsd.edu/ calder/simpoint/, 2003.

[18] Alper Buyuktosunoglu Canturk Isci, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[19] Cavium Inc. ThunderX Family of Workload Optimized Processors. `http://www.cavium.com/pdfFiles/ThunderX_PB_p12_Rev1.pdf`, 2013.

[20] Cavium Inc. Cavium Introduces ThunderX™: A 2.5 GHz, 48 Core Family of Workload Optimized Processors for Next Generation Data Center and Cloud Applications. `http://www.cavium.com/newsevents_Cavium_Introduces_ThunderX_A_2.5_GHz_48_Core_Family_of_Workload_Optimized_Processors_for_Next_Generation_Data_Center_and_Cloud_Applications.html`, 2014.

[21] Pedro Chaparro, José González, and Antonio González. Thermal-effective clustered microarchitectures. In *Workshop on Temperature-Aware Computer Systems*, 2004.

[22] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2001.

[23] Jian Chen and Lizy Kurian John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *Intl. Conf. on Supercomputing (ICS)*, 2011.

[24] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. In *Intl. Conf. on Measurement and Modeling of Computer Systems (Sigmetrics)*, 2011.

[25] Jian Chen, A Nair, and L John. Predictive heterogeneity-aware application scheduling for chip multiprocessors. *IEEE Trans. on Computers*, 63, 2012.

[26] Seungryul Choi and Donald Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Intl. Symp. on Computer Architecture (ISCA)*, 2006.

[27] George Chrysos. Intel xeon phi coprocessor (codename knights corner). In *IEEE Symp. on High Performance Chips (HOT CHIPS)*, 2012.

[28] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Intl. Symp. on Computer Architecture (ISCA)*, 2013.

[29] Christina Delimitrou and Christos Kozyrakis. The netflix challenge: Datacenter edition. *Computer Architecture Letters (CAL)*, 2013.

[30] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[31] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[32] DigitalOcean. Transparent huge pages and alternative memory allocators: A cautionary tale.
`https://www.digitalocean.com/company/blog/`

transparent-huge-pages-and-alternative-memory-allocators/.

[33] Matthew Dillon. Page coloring. http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html.

[34] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[35] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A Joao, Onur Mutlu, and Yale N Patt. Parallel application memory scheduling. In *Intl. Symp. on Microarchitecture (MICRO)*, 2011.

[36] EE Times. Cavium Thunder Rattles Xeon: 48-core SoCs gun for mainstream servers. http://www.eetimes.com/document.asp?doc_id=1322565&_mc=sm_eet, 2015.

[37] EPFL. Cloudsuite. http://cloudsuite.ch/datacaching/.

[38] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE MICRO*, 28(3):42–53, 2008.

[39] Stijn Eyerman and Lieven Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Trans. on Architecture and Code Optimization (TACO)*, 8(1), 2011.

[40] Michal Feldman, Kevin Lai, and Li Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Intl. Conf. on Electronic Commerce (EC)*, 2005.

[41] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[42] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Navigating

heterogeneous processors with market mechanisms. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[43] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[44] Garrett Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, 1968.

[45] Kieran Harty and David Cheriton. A market approach to operating system memory allocation, market-based control: a paradigm for distributed resource allocation. *Market-Based Control: A paradigm for distributed resource allocation*, 1996.

[46] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: from concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2016.

[47] IBM Inc. 64KB pages on Linux for Power systems. `https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Welcome+to+High+Performance+Computing+(HPC)+Central/page/64KB+pages+on+Linux+for+Power+systems`, 2012.

[48] ARM Inc. big.LITTLE technology: The future of mobile. `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`, 2013.

[49] Micron Technology Inc. 2Gb DDR3 SDRAM component data sheet: MT41J256M8. http://www.micron.com/parts/dram/ddr3-sdram/mt41j256m8da-125, July 2012.

[50] Red Hat Inc. Huge pages and transparent huge pages. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html`.

[51] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manuals.

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2014.

[52] Ruzica Jevtic, Hanh-Phuc Le, Milovan Blagojevic, Stevo Bailey, Krste Asanovic, Elad Alon, and Borivoje Nikolic. Per-core dvfs with switched-capacitor converters for energy efficiency in manycore processors. *IEEE Trans. on Very Large Scale Integration (TVLSI) Systems*, 2014.

[53] Ramesh Johari and John N Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[54] Vasileios Karakostas, Osman S Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2014.

[55] Frank Kelly. Charging and rate control for elastic traffic. *European Trans. on Telecommunications*, 8(1):33–37, 1997.

[56] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 1992.

[57] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2008.

[58] Kirk Kirkconnell. Often overlooked linux os tweaks. `http://blog.couchbase.com/often-overlooked-linux-os-tweaks`.

[59] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Intl. Symp. on Microarchitecture (MICRO)*, 2003.

[60] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1, 2005.

[61] Hyunjin Lee, Sangyeun Cho, and Bruce R Childers. Cloudcache: Expand-

ing and shrinking private caches. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2011.

[62] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2008.

[63] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[64] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, and Chengyong Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[65] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[66] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Intl. Symp. on Computer Architecture (ISCA)*, 2015.

[67] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (PriSM). In *Intl. Symp. on Computer Architecture (ISCA)*, 2012.

[68] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Intl. Symp. on Microarchitecture (MICRO)*, 2011.

[69] Andreu Mas-Colell, Michael Dennis Whinston, Jerry R Green, et al. *Microeconomic theory*, volume 1. Oxford University Press New York, 1995.

[70] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Intl. Symp. on Microarchitecture (MICRO)*, 2012.

[71] Mark S Miller, David Krieger, Norman Hardy, Chris Hibbert, and E Dean Tribble. An automated auction in atm network bandwidth. *Market-Based Control: A paradigm for distributed resource allocation*, 1996.

[72] MongoDB. Disable transparent huge pages (thp). `https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/`.

[73] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[74] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[75] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*. Cambridge University Press, 2007.

[76] Christos Papadimitriou. Algorithms, games, and the internet. In *Intl. Symp. on Theory of computing*, 2001.

[77] Binh Pham, Ján Veselỳ, Gabriel H Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Intl. Symp. on Microarchitecture (MICRO)*, 2015.

[78] Michael D Powell, Amit Agarwal, TN Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Intl. Symp. on Microarchitecture (MICRO)*, 2001.

[79] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[80] Ori Regev and Noam Nisan. The popcorn market. online markets for computational resources. *Decision Support Systems*, 28(1):177–189, 2000.

[81] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[82] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):0020–27, 2012.

[83] Rafael H Saavedra, R Stockton Gaines, and Michael J Carlton. Micro benchmark analysis of the ksr1. In *ACM/IEEE Conf. on Supercomputing*, 1993.

[84] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Intl. Symp. on Microarchitecture (MICRO)*, 2010.

[85] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Intl. Symp. on Computer Architecture (ISCA)*, 2011.

[86] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing cache misses using hardware and software page placement. In *Intl. Conf. on Supercomputing (ICS)*, 1999.

[87] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, HP Western Research Labs, 2001.

[88] Abhishek A Sinkar, Hamid Reza Ghasemi, Michael J Schulte, Ulya R Karpuzcu, and Nam Sung Kim. Low-cost per-core voltage domain support for power-constrained high-performance processors. *IEEE Trans. on Very Large Scale Integration (TVLSI) Systems*, 22(4):747–758, 2014.

[89] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. on Architecture and Code Optimization (TACO)*, 1(1):94–125, 2004.

[90] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. A. and C. Black, 1863.

[91] Standard Performance Evaluation Corporation. SPEC CPU2000. http://www.spec.org/cpu2000/, 2000.

[92] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/, 2006.

[93] G Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[94] Ivan Edward Sutherland. A future market in computer time. *Communications of the ACM*, 11, 1968.

[95] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[96] tbreak Media. Qualcomms Prepping A 64-core ARM Server CPU Called Hydra. http://www.tbreak.com/qualcomm-working-on-hydra-its-64-core-arm-server-processor/, 2015.

[97] The OVH group labs. ARM Cloud. `https://www.runabove.com/armcloud.xml`, 2016.

[98] Dean M Tullsen and Jeffery A Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Intl. Symp. on Microarchitecture (MICRO)*, 2001.

[99] Hal R Varian. Equity, envy, and efficiency. *Journal of economic theory*, 9(1):63–91, 1974.

[100] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Intl. Symp. on Operating System Design and Implementation (OSDI)*, 1994.

[101] Ruisheng Wang and Lizhong Chen. Futility scaling: High-associativity cache partitioning. In *Intl. Symp. on Microarchitecture (MICRO)*, 2014.

[102] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F. Martínez. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2017.

[103] Xiaodong Wang and José F. Martínez. Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[104] Xiaodong Wang and José F. Martínez. ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[105] Fang Wu and Li Zhang. Proportional response dynamics leads to market equilibrium. In *Intl. Symp. on Theory of computing*, 2007.

[106] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Intl. Symp. on Computer Architecture (ISCA)*, 2009.

[107] S Yang, Michael D Powell, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2001.

[108] Se-Hyun Yang, Michael D Powell, Babak Falsafi, and TN Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[109] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[110] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[111] Seyed Majid Zahedi and Benjamin C Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[112] Li Zhang. The efficiency and fairness of a fixed budget resource allocation game. In *Automata, Languages and Programming*, pages 485–496. Springer, 2005.

[113] Li Zhang. Proportional response dynamics in the fisher market. *Theoretical Computer Seience*, 412(24):2691–2698, 2011.

[114] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *European conference on Computer systems (EuroSys)*, 2009.

[115] Yanqi Zhou and David Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *Intl. Symp. on Computer Architecture (ISCA)*, 2016.